Análise de Performance na Localização de Bugs apoiada pela Dissecção de Conjuntos de Dados

Victor Sobreira

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2022

Victor Sobreira

Análise de Performance na Localização de Bugs apoiada pela Dissecção de Conjuntos de Dados

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2022

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S677
2022

Sobreira, Victor, 1981-
Analysis of Bug Localization Performance Supported by Dataset Dissection [recurso eletrônico] / Victor Sobreira. - 2022.

Orientador: Marcelo de Almeida Maia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2022.60
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Maia, Marcelo de Almeida,1969-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO – FACOM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO – PPGCO

The undersigned hereby certify they have read and recommend to the PPGCO for acceptance the dissertation entitled **"Analysis of Bug Localization Performance Supported by Dataset Dissection"** submitted by **"Victor Sobreira"** as part of the requirements for obtaining the **PhD's degree in Computer Science**.

Uberlândia, January 24, 2022.

Supervisor: ______________________________
Prof. Marcelo de Almeida Maia, Ph.D.
Universidade Federal de Uberlândia

Examining Committee Members:

____
Prof. Eduardo Figueiredo, Ph.D.
Universidade Federal de Minas Gerais

____
Prof. Fabiano Azevedo Dorça, Ph.D.
Universidade Federal de Uberlândia

____
Prof. Flávio de Oliveira Silva, Ph.D.
Universidade Federal de Uberlândia

____
Prof. Uirá Kulesza, Ph.D.
Universidade Federal do Rio Grande do Norte

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 1/2022, PPGCO				
Data:	24 de janeiro de 2022	Hora de início:	08:30	Hora de encerramento:	12:50
Matrícula do Discente:	11423CCP007				
Nome do Discente:	Victor Sobreira				
Título do Trabalho:	Análise de Performance na Localização de Bugs apoiada pela Dissecção de Conjuntos de Dados				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Fabiano Azevedo Dorça - FACOM/UFU, Flávio de Oliveira Silva - FACOM/UFU, Eduardo Magno Lages Figueiredo - DCC/UFMG, Uirá Kulesza - DIMAp/UFRN e Marcelo de Almeida Maia - FACOM/UFU orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Eduardo Magno Lages Figueiredo - Belo Horizonte/MG; Uirá Kulesza - Natal/RN; Fabiano Azevedo Dorça, Flávio de Oliveira Silva e Marcelo de Almeida Maia - Uberlândia/MG. O discente participou da cidade de Uberlândia/MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 24/01/2022, às 17:08, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Flávio de Oliveira Silva**, **Professor(a) do Magistério Superior**, em 25/01/2022, às 07:13, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Uirá Kulesza**, **Usuário Externo**, em 25/01/2022, às 10:11, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Usuário Externo**, em 25/01/2022, às 10:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Fabiano Azevedo Dorça**, **Professor(a) do Magistério Superior**, em 03/02/2022, às 09:57, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3322808** e o código CRC **77301D0B**.

À minha querida e amada Alice.

Agradecimentos

Agradeço a Deus e a todos que direta ou indiretamente contribuíram para o desenvolvimento e conclusão deste trabalho.

Agradeço ao meu orientador Marcelo de Almeida Maia pela oportunidade, solicitude e por acreditar, em alguns momentos até mais que eu, que esta "parte" da pesquisa pudesse ser finalizada. Agradeço aos membros da banca de qualificação, Celso Gonçalves Camilo Júnior e Marcelo Keese Albertini, pelas sugestões e contribuições para a sequência do trabalho naquele momento. Neste sentido, também agradeço aos membros da banca de defesa, Eduardo Figueiredo, Fabiano Azevedo Dorça, Flávio de Oliveira Silva e Uirá Kulesza, pela atenção, dedicação e cuidado que culminaram em contribuições bem vindas para a lapidação do texto final da tese.

Agradeço ao PPGCO, aos membros do colegiado, à coordenação antiga e atual por toda a compreensão e por ter permitido que pudesse me manter vinculado ao programa até a conclusão do trabalho. Agradeço especialmente ao técnico Erisvaldo Fialho por sempre ser solicito e prestativo em todas as minhas requisições e pedidos e também ao técnico Alessandro Gonçalves da Silva que salvou minha pele várias vezes em questões técnicas com os servidores onde rodei meus experimentos. Agradeço também à FACOM e à UFU que permitiram a licença para a qualificação, fundamental para o desenvolvimento de grande parte do trabalho e, sem a qual, teria sido quase impossível uma conciliação com as atividades docentes regulares.

Agradeço aos colegas, especialmente do LASCAM, que em muitos momentos serviram de incentivo, fizeram boas colaborações ou simplesmente propiciaram boas conversas, reflexões e discussões. Que esses momentos possam se repetir mais. Parabenizo aqueles que, nessa janela comum de passagem pelo programa, já haviam concluído seus trabalhos (Eduardo, Klérisson, Allyson, Fernanda e Rodrigo) e desejo boa sorte e boa

conclusão aqueles que no momento dessa escrita ainda estavam desenvolvendo ou finalizando (Adriano e Carlos Eduardo).

Agradeço especialmente aos meus colaboradores no trabalho com o "dissection" do Defects4J cujo tema, naquele momento da colaboração, ainda parecia um pouco distante do tema principal de meu projeto de tese, mas que acabou sendo fundamental para a conclusão, inclusive pelos desdobramentos do segundo artigo encabeçado pela Fernanda e com as importantes contribuições de Thomas. Nesses momentos, vemos o quanto a colaboração deveria ser mais valorizada, incentivada e estimulada para trabalhos que busquem fazer a diferença em suas áreas. Muito obrigado Fernanda Madeiral, Thomas Durieux e Martin Monperrus.

Agradeço a meus pais, Izilda e Tadeu, por terem sempre me incentivado e apoiado nos estudos. As sementes plantadas muito cedo foram o que permitiram enraizar valores como a responsabilidade, o compromisso, a dedicação, a disciplina e a persistência. Agradeço a minha irmã Vickele, que pôde estar presente junto a meus pais, nos muitos momentos que estive distante e que, em sua maioria, estava focado no trabalho. Neste ínterim, o nascimento e presença da Helena, ajudou a alegrar, preencher lacunas e transpor os vários percalços que a vida trouxe a todos.

Agradeço à minha esposa, Mirella, que esteve ao meu lado todo esse tempo. Vivemos momentos de todo tipo, dificuldade e intensidade. Vivemos e superamos desafios e situações que nem imaginávamos. Agradeço também por ter sido forte, resistido e acreditado que toda dor é passageira e que com fé, esperança e amor verdadeiro podemos transpor os maiores obstáculos. Agradeço especialmente por ter me dado o meu maior presente, minha filha Alice, que é minha fonte de inspiração, força e alegrias. À Alice agradeço por todos os momentos que vivemos, pelos aprendizados, pela nova perspectiva de vida e pelos vários sopros de esperança, persistência e fé que me inspirou. Contem sempre comigo em suas caminhadas e que sejam repletas de momentos felizes e memoráveis.

Agradeço aos amigos verdadeiros, aos irmãos e todos que, com uma palavra sábia, um incentivo ou com um simples gesto, colaboraram e continuam colaborando na caminhada e na superação do desafio diário de sempre seguir em frente, aparar nossas asperezas, desenvolver nossas virtudes e, com isso, persistir no ideal de busca da felicidade humana.

"Tudo passa, tudo passará [...]
Temos muito ainda por fazer
Não olhe pra trás
Apenas começamos
O mundo começa agora
Apenas começamos"
(Metal contra as nuvens - Russo, R.; Bonfá, M.; Villa-Lobos, D.; Neto, E. S.)

Resumo

Encontrar e corrigir a causa de falhas em *software* continua sendo um grande desafio. Tais tarefas exigem dos desenvolvedores esforço e experiência equivalentes as necessárias para o desenvolvimento de novas funcionalidades. Nas últimas décadas, a comunidade de pesquisa esteve ativa na produção de abordagens para apoiar a depuração de software. A tarefa de Localização de Faltas (LF) é um passo essencial, independente da abordagem utilizada para reparo de programas (automática ou manual). Entretanto, as abordagens automatizadas de localização são críticas para tornar o processo mais eficaz e eficiente. Existem muitas abordagens para a LF automática e todas têm um alvo comum: melhorar a precisão do ranqueamento de componentes de *software* suspeitos de conter uma falta. Uma questão recorrente é a indefinição sobre as razões do sucesso ou fracasso das abordagens sobre o conjunto de dados de faltas avaliado, uma vez que a maioria dos métodos não considera a natureza e as características intrínsecas das faltas. A discussão ainda é muito focada em ganhos de desempenho nos comparativos com o estado da arte. Este trabalho visa apoiar as tarefas de reparo de *software*, com foco primário no suporte automatizado à LF. Primeiro, investigamos as características associadas as faltas comumente utilizadas na avaliação de estratégias de LF (o que se estende também ao reparo automático de programas). Então, analisamos as relações entre essas características e como influenciam a performance da LF. Partimos de uma abordagem estática de LF, baseada em algoritmos de aprendizado de rankings, Learning to Rank (LtR), e tendo relatórios de bugs como entrada do processo. Inicialmente, analisamos um conhecido conjunto de dados de faltas, Defects4J, de onde extraímos várias características das faltas. Posteriormente, analisamos tais características em um conjunto de dados maior, o qual referenciamos como *LR–dataset*. Então, levantamos várias estratégias e alternativas para a melhoria dos rankings de arquivos suspeitos de falta

e gerados por abordagens de LF. Por exemplo, o uso de novas características (como a Entropia do Código), o ajuste de hiper-parâmetros e o balanceamento de dados para treinamento em abordagens de aprendizado de máquina e, finalmente, a amostragem de falhas guiada pela análise de códigos de reparo. Para isso, testamos as alternativas para melhoria dos rankings de componentes suspeitos por meio de um ambiente construído para experimentação e reprodução de estratégias para a LF. Mostramos que as estratégias de pré-processamento de relatórios de bugs e dos conjuntos de dados, além do ajuste de diferentes algoritmos de LtR, podem produzir resultados diferentes para os rankings mesmo usando abordagens prévias de LF. Além disso, as características das falhas amostradas para a avaliação podem influenciar significativamente o ranqueamento dos arquivos suspeitos, por exemplo, dependendo do tipo de padrões e ações de reparo necessários para a correção das falhas envolvidas. Este é o caso do padrão de reparo *Missing Not-Null Check* cuja presença em uma das amostras experimentais gerou um ranking de arquivos suspeitos marcando 27.22 pontos percentuais acima da linha base, ou seja, quando nós não consideramos a presença (ou ausência) do padrão. Esses resultados apontam para oportunidades de revisão das abordagens prévias de LF sob as lentes da dissecção dos conjuntos de dados utilizados na avaliação, com potencial de novos entendimentos, interpretações e composições de estratégias para LF.

Palavras-chave: Localização de Bug, Reparo Automático de Software, Análise de Reparos, Dissecção de Conjuntos de Dados de Bugs, Depuração de Software, Aprendizado de Rankings.

Abstract

Finding and fixing software bugs still is a big challenge. These tasks demand developers as much effort and experience as required to develop new functionality. Last decades, the research community actively produced approaches to support the debugging process. The Bug Localization (BL) task is an essential step, wherever is the applied software repair approach (automated or manual). However, automated techniques for BL are critical in turning the process more effective and efficient. There are many approaches to automated BL, and all of them have one frequent goal: to improve accuracy performance in classifying software components suspected of containing bugs. One recurrent issue is the lack of clarity about the reasons for the success or failure of the approaches on the assessed bug dataset since most methods do not consider the nature and intrinsic characteristics of the bugs. The discussion is still too focused on performance gains compared to the previous state-of-the-art. This work aims to contribute to software repair tasks, primarily focusing on supporting the automated BL. First, we explored characteristics of bugs usually applied in the assessment of the localization strategies (also extended to automated program repair). Then, we analyze the relationships between these bug characteristics and their influence on the performance of localization strategies. We start from a static information-based BL approach, based in LtR algorithms, having bug reports as input to the localization process. Initially, we analyze a well-known bug dataset, Defects4J, from where we extract various bugs' characteristics. Next, we analyzed these characteristics in a larger dataset referred to as LR-dataset. Then, we raise various strategies and alternatives to improve the ranking of suspect buggy files and generated by BL approaches. Some examples are the use of new features (e.g., Code Entropy), the tuning of hyperparameters and the data balance for training in Machine Learning (ML) based approaches, and, finally, bugs' sampling

guided by patch analysis. For that, we tested the alternatives to improve the ranking of suspected components with an environment built for experimenting with and reproducing the BL strategies. We show that pre-processing strategies on bug reports and also on the dataset, besides the tuning of different LtR algorithms, can produce different ranking results even with past BL approaches. Still, characteristics of the bugs sampled for assessment can influence ranking scores of buggy suspected files, e.g., depending on the type of associated repair patterns and repair actions required to fix the bugs. For example, this is the case for the Missing Not-Null Check repair pattern whose presence in an experimental sample produces a suspicious score ranking 27.22 percentual points above the baseline when we do not consider the presence (or absence) of the pattern. These results point to opportunities to review the BL past approaches under the lens of dataset dissection applied in the assessment and with a potential to new insights, interpretations, and compositions of strategies for BL.

Keywords: Bug Localization, Automatic Program Repair, Patch Analysis, Bugs' Dataset Dissection, Debugging, Learn-to-Rank.

List of Figures

Figure 1 – Chapters Roadmap. 28
Figure 2 – *Bug Report* for the bug LANG-552 from Apache Commons Lang project. 31
Figure 3 – *Patch* for the bug LANG-552 from Apache Commons Lang project. . 32
Figure 4 – GumTree generated AST representing the patch for the bug LANG-552 shown in Figure 3. 34
Figure 5 – *Patch* for the bug Math-58 from Apache Commons Math project. . . 34
Figure 6 – *Patch* for the bug Math-41 from Apache Commons Math project. . . 35
Figure 7 – *Patch* for the bug Closure-13 from Closure Compiler project. 35
Figure 8 – *Patch* for the bug Lang-17 from Apache Commons Lang project. . . . 36
Figure 9 – Static approaches for BL until 2021. 62
Figure 10 – Dynamic approaches for BL until 2021. 62
Figure 11 – Hybrid approaches for BL until 2021. 63
Figure 12 – Distribution of the number of lines in each patch of Defects4J projects. 69
Figure 13 – Distribution of the chunks composing each patch of Defects4J projects. 70
Figure 14 – Spreading distribution on each patch of Defects4J projects. 70
Figure 15 – Repair actions incidence in patches from Defects4J projects. 71
Figure 16 – Distribution of the number of repair patterns by patch of Defects4J projects. 71
Figure 17 – BL process overview and associated modules of the experimental package. 80
Figure 18 – SVM-light format. 83
Figure 19 – Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), Normalized Discounted Cumulated Gain (NDCG)@10 of AspectJ and Eclipse Standard Widget Toolkit (SWT) features. 95

Figure 20 – the percentual of success on finding relevant items in the top positions of a ranking limited to N items (Top-N) of AspectJ and SWT features 96
Figure 21 – MAP, MRR, Top-{1,5,10} results on 300 bug reports of SWT (200 for training and 100 for testing) 98
Figure 22 – NDCG@10 for MART (Lambda and Obliviuos) and Random Forest algorithms from QuickRank and RankLib tools. 100
Figure 23 – NDCG@10 for Stochastic, ListNet, DART, Linear Regression, Coordinate Ascent, RankNet, RankBoost, AdaRank from QuickRank and RankLib tools). 101
Figure 24 – Tuning LambdaMART (RankLib) on 300 bug reports of SWT: MAP and NDCG@10 performance changing Shrinkage {0.05, 0.5, 0.8} and Number of Leaves (NL) {1, 5, 10}. 106
Figure 25 – Tuning LambdaMART on SWT (RankLib): MAP performance changing Number of Trees (NT) from 32 doubling until 512 and Number of Leaves (NL) = {1, 10}. 107
Figure 26 – Average computing time distribution per bug report for AspectJ, SWT, and Tomcat. 108
Figure 27 – Distribution of the number of lines of the patches limited to a maximum size. 114
Figure 28 – Distribution of the number of code lines by type in 20,138 patches of Learning to Rank approach from Ye, Bunescu e Liu (2016) (LR)–dataset. 115
Figure 29 – Overall patches distribution according to the type of code lines affected.116
Figure 30 – Patches according to the type of code lines affected in each project. . 116
Figure 31 – Distribution of the number of chunks of the patches. 118
Figure 32 – Distribution of chunks spreading of the patches. 119
Figure 33 – Repair actions found in the LR–dataset. 121
Figure 34 – Grouped repair actions found in LR–dataset. 124
Figure 35 – Grouped repair actions incidence on each project. 125
Figure 36 – Repair patterns found in Defects4j Dissection. 126
Figure 37 – Repair patterns with variations found in Defects4j Dissection. 126
Figure 38 – Repair patterns found in LR–dataset. 128
Figure 39 – Grouped repair patterns found in LR–dataset. 128
Figure 40 – Grouped Repair Patterns incidence on LR–dataset projects. 129

Figure 41 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) AspectJ and in b) Eclipse Business Intelligence and Reporting Tools (BIRT). 132
Figure 42 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) Eclipse and b) Eclipse Java Development Tools (JDT). 133
Figure 43 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) SWT and b) Tomcat. 134
Figure 44 – Repair Patterns variations: a) Conditional Block Return Add (1..5); b) Conditional Block Return Add (6..10); c) Conditional Block Exception Add; d) Conditional Block Removal. 136
Figure 45 – Repair Patterns variations: a) Conditional Block Others Add (1..4); b) Conditional Block Others Add (5..8). 137
Figure 46 – Repair Patterns variations: a) Missing Null-Check; b) Constant Change; c) Code Moving. 137
Figure 47 – Repair Patterns variations: Single Line a) 1 to 11; b) 12 to 22; c) Expression Fix. 138
Figure 48 – Repair Patterns variations: a) Wraps-with; b) Unwraps-with. 139
Figure 49 – Repair Patterns variations: Wrong Method Reference a) 1 to 5; b) 6 to 10; c) Wrong Variable Reference. 140
Figure 50 – Bug Report for the bug 7861 from Eclipse, with a snippet of the patch matching the *Wrong Variable Reference* repair pattern. 141
Figure 51 – Bug Report for the bug 187445 from BIRT, with a snippet of the patch matching the *Logic Expression Expansion* and *Copy Paste* repair patterns. 142
Figure 52 – Patterns co-occurrence without outliers in: a) AspectJ and b) BIRT. 143
Figure 53 – Patterns co-occurrence without outliers in a) Eclipse Platform UI and b) JDT. 144
Figure 54 – Patterns co-occurrence without outliers in a) SWT and b) Tomcat. . 145
Figure 55 – Applications of Defects4J Dissection study by research area until November of 2021. 146
Figure 56 – Bug reports categories: 1) Functional vs Non-Functional; 2) With or Without LR-Results. 153
Figure 57 – *Bug Report* for the bug 117526 from Eclipse Platform UI. 154
Figure 58 – Patch for the bug 117526 in files from Eclipse project. 155

Figure 59 – Score rankings differences for *Wraps with If* repair pattern in AspectJ (Functional Code (FC)+Non-Functional Code (NFC)). 177
Figure 60 – Score rankings differences for *Wraps with If* repair pattern in Tomcat (FC+NFC). 178
Figure 61 – Score rankings differences for *Wrong Method Reference* repair pattern in AspectJ (FC+NFC). 179
Figure 62 – Score rankings differences for *Wrong Method Reference* repair pattern in Tomcat (FC+NFC). 180
Figure 63 – Score rankings differences for *Wrong Var Reference* repair pattern in AspectJ (FC+NFC). 181
Figure 64 – Score rankings differences for *Wrong Var Reference* repair pattern in Tomcat (FC+NFC). 182
Figure 65 – Score rankings differences for *Missing Not-Null Check Reference* repair pattern in AspectJ (FC+NFC). 183
Figure 66 – Score rankings differences for *Missing Not-Null Check Reference* repair pattern in Tomcat (FC+NFC). 184

List of Tables

Table 1 – Summary of Static approaches for BL. 59
Table 2 – Summary of Dynamic approaches for BL. 60
Table 3 – Summary of Hybrid approaches for BL. 60
Table 4 – Imbalanced data in LR–dataset. 72
Table 5 – Original features in (YE; BUNESCU; LIU, 2016). 90
Table 6 – Best features per project according with (YE; BUNESCU; LIU, 2016) 90
Table 7 – Entropy features computed in exploratory experiments. 91
Table 8 – Selection and weighting of features from Table 6. 92
Table 9 – Performance variation of SVMRank on SWT by tuning the algorithm with capacity parameter in LR-All setting. 99
Table 10 – Performance statistics from the tuning of LambdaMART on SWT using baseline and $\phi_{20.1}$ entropy settings. 103
Table 11 – Parameters found in the best performance settings while tuning LambdaMART on SWT bug reports. 104
Table 12 – Descriptive statistics for 21,177 bug patches. 120
Table 13 – Descriptive statistics for 20,138 bug patches, without outlier patches (more than 60 lines). 120
Table 14 – Repair actions acronyms and full names. 122
Table 15 – Repair actions acronyms and grouping names. 123
Table 16 – Repair patterns, acronyms and groups. 127
Table 17 – LR–dataset with and without outliers. 156
Table 18 – LR–dataset sampling candidates. 157
Table 19 – LR–dataset samples representativeness for AspectJ and Tomcat. . . . 158

Table 20 – *Wraps with If* in AspectJ and Tomcat, H0 result for Mann-Whitney (MW) test. 164
Table 21 – *Wrong Method Reference* in AspectJ, Tomcat and BIRT: H0 result for Mann-Whitney (MW) test. 165
Table 22 – *Wrong Variable Reference* in AspectJ, H0 result for Mann-Whitney (MW) test. 166
Table 23 – *Missing Not-Null Check* in AspectJ, H0 result for Mann-Whitney (MW) test. 167
Table 24 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the MAP scores. 169
Table 25 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the MRR scores. 170
Table 26 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@1 scores. 171
Table 27 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@1 scores. 172
Table 28 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@10 scores. 173
Table 29 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-1 scores. 174
Table 30 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-1 scores. 175
Table 31 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-10 scores. 176

Acronyms list

ADD Automatic Diff Dissection

API Application Programming Interface

APR Automated Program Repair

AST Abstract Syntax Tree

BIRT Eclipse Business Intelligence and Reporting Tools

BL Bug Localization

BLUiR **B**ug **L**ocalization **U**sing **i**nformation **R**etrieval

DL Deep Learning

DNN Deep Neural Networks

FL Fault Localization

FC Functional Code

NFC Non-Functional Code

GB Giga Bytes

HITS Hyperlink-Induced Topic Search

IDE Integrated Development Environment

IR Information Retrieval

IRBL Information Retrieval-based Bug Localization

JDT Eclipse Java Development Tools

JSON JavaScript Object Notation

JUNG Java Universal Network/Graph Framework

LDA Latent Dirichlet Allocation

LF Localização de Faltas

LM Language Models

LR Learning to Rank approach from Ye, Bunescu e Liu (2016)

LSI Latent Semantic Index

LtR Learning to Rank

MAP Mean Average Precision

ML Machine Learning

MRR Mean Reciprocal Rank

NDCG Normalized Discounted Cumulated Gain

NLTK Natural Language Toolkit

ORM Object Relational Mapping/Mapper

PPD Patch Pattern Detector

RQ Research Question

SBFL Spectrum-Based Fault Localization

SUM Smoothed Unigram Model

SWT Eclipse Standard Widget Toolkit

Top-N the percentual of success on finding relevant items in the top positions of a ranking limited to N items

VSM Vector Space Model

Contents

1 INTRODUCTION **21**
1.1 Motivation **21**
1.2 Objectives **23**
1.2.1 General Objectives 23
1.2.2 Specific Objectives 23
1.3 Thesis Declaration Proposal **24**
1.4 Research Summary, Assumptions, and Questions **24**
1.5 Contributions **26**
1.6 Thesis Organization **27**

2 BACKGROUND **29**
2.1 Essential concepts about Bug Localization **29**
2.1.1 What is a bug? 29
2.1.2 Bug Reports 30
2.1.3 Where is a bug located? What is it found? 33
2.1.4 How to find bugs? 34
2.2 Bug datasets **36**
2.2.1 Defects4J 37
2.2.2 LR-dataset 37
2.3 Performance metrics in Bug Localization **38**
2.3.1 Precision, Recall and F-measure 38
2.3.2 Precision@k 39
2.3.3 Top-N 40
2.3.4 MAP 40

2.3.5 MRR . . . 41
2.3.6 NDCG . . . 41
2.3.7 Other metrics . . . 43
2.4 Machine Learning Approaches . . . 43
2.4.1 Learning to Rank . . . 43
2.4.2 Language Models . . . 45
2.5 Final Considerations . . . 45

3 STRATEGIES FOR BUG LOCALIZATION . . . 47
3.1 Static Information-Based Approaches . . . 47
3.1.1 LR . . . 48
3.1.2 AmaLgam+ . . . 49
3.1.3 DNNLoc . . . 50
3.1.4 ConCodeSe . . . 51
3.1.5 NSGA-II . . . 52
3.1.6 Locus . . . 53
3.1.7 BLIA . . . 53
3.1.8 **B**ug **L**ocalization **U**sing **i**nformation **R**etrieval (BLUiR) . . . 54
3.1.9 BugLocator . . . 55
3.2 Dynamic Information-Based Approaches . . . 55
3.2.1 Tarantula . . . 55
3.2.2 D* . . . 56
3.3 Hybrid approaches . . . 56
3.3.1 EnSpec . . . 57
3.3.2 AML . . . 57
3.4 Reference and chronology of BL approaches . . . 58
3.5 Final considerations . . . 58

4 ON THE INFLUENTIAL FACTORS FOR BUG LOCALIZATION EXPLORATORY ASSESSMENT . . . 65
4.1 Bug Reports' Pre-processing . . . 65
4.2 Dataset Quality Assessment and Source Code Filtering . . . 66
4.3 Bug Classification Schemes . . . 67
4.4 Handling imbalanced data . . . 72
4.5 Data Splitting Strategies . . . 73
4.6 Source Code Representation . . . 74

4.7 New Features for Bug Localization . . . 75
4.7.1 Entropy . . . 75
4.7.2 Word2Vec, Glove and ConceptNet . . . 76
4.7.3 Commits and Patches . . . 77
4.7.4 Other features . . . 78
4.8 LtR Tools and Models for BL . . . 78
4.9 Experimental Package for Bug Localization Assessment . . . 79
4.9.1 Experimental Package Overview . . . 79
4.9.2 Input . . . 81
4.9.3 Feature Extraction . . . 82
4.9.4 LtR Input . . . 82
4.9.5 LtR Output . . . 83
4.10 Final Considerations . . . 86

5 STRATEGIES FOR LEARNING-TO-RANK BUG LOCALIZATION IMPROVEMENT . . . 87
5.1 Evaluation Method . . . 87
5.1.1 Dataset . . . 88
5.1.2 Data Preparation and Cleaning . . . 88
5.1.3 Experiment Configurations . . . 89
5.1.4 Metrics Extracted . . . 93
5.1.5 Runtime Environment . . . 93
5.2 Results . . . 93
5.2.1 RQ1: What is the performance of the entropy features compared to other features? . . . 93
5.2.2 RQ2: The use of entropy feature can improve the results obtained by past learning approaches to BL? . . . 95
5.2.3 RQ3: What is the impact of data balance strategies in the learning process? . . . 97
5.2.4 RQ4: How does the tuning of LtR algorithms impact the BL performance? . . . 99
5.2.5 RQ5: How long does it take to conclude each step in the process (feature extraction, ranking generation, training, validation, and testing)? 105
5.3 Final Considerations . . . 109

6 ANALYSIS OF REPAIR ACTIONS AND PATTERNS 111
6.1 The role of patches on Bug Localization 111
6.2 Understanding the nature of the bugs through their patches . 112
6.3 Analysis dimensions of a bug patch for Bug Localization . . . 112
6.3.1 Size dimensions 113
6.3.2 Spreading 114
6.3.3 Size and Spreading Dimensions' Statistics 118
6.3.4 Repair actions 120
6.3.5 Repair patterns 124
6.4 Patterns composition 129
6.4.1 Repair Actions 130
6.4.2 Patterns Co-occurrences 135
6.5 Actual applications for the Defects4J Dissection study 141
6.6 Related Work 142
6.7 Final Considerations 148

7 INFLUENCE OF REPAIR PATTERNS ON BL APPROACHES149
7.1 Research Questions 149
7.2 Evaluation Method 150
7.2.1 Dataset preparation and cleaning 150
7.2.2 Selected Settings 157
7.2.3 Metrics Extracted and Hypothesis Tests 159
7.2.4 Runtime Environment 160
7.3 Results 160
7.3.1 Screening of Repair Patterns 161
7.3.2 Differences on ranking scores correlated to the repair pattern presence (or absence) and with statistical significance 163
7.3.3 Impact of the differences correlated to the repair patterns 163
7.3.4 Variation of the differences correlated to the repair patterns 168
7.4 Analysis 169
7.4.1 Wraps with If Repair Pattern 170
7.4.2 Wrong Method Reference Repair Pattern 185
7.4.3 Wrong Variable Reference Repair Pattern 185
7.4.4 Missing Not-Null Check Repair Pattern 186
7.5 Answers for the Research Questions 187

7.5.1 RQ6: on the existence of differences correlated to the presence versus absence of repair patterns in patches 187
7.5.2 RQ7: on the type of impact correlated to the presence versus absence of repair patterns in patches 188
7.5.3 RQ8: on the degree of the impact correlated to the presence versus absence of repair patterns in patches 188
7.6 Threats to Validity 190
7.7 Limitations and Future Work 190
7.8 Related Work 191
7.9 Final Considerations 192

8 CONCLUSION 195
8.1 Final considerations on the research 196
8.2 Main Contributions 198
8.3 Future Work 199
8.4 Bibliographical Production 200

BIBLIOGRAPHY 203

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

student name and signature

Chapter 1

Introduction

Even after notable and world-class faults like the Millennium Bug (Y2K), the Software Engineering area still steps slowly and spends many resources trying to fix software. Studies referenced by MITCHELL (2009) have shown the spences of companies to deal with the Y2K reached around U$ 308 billion, perhaps, one of the most representative registered cases of huge budgets spent on repairing software. The lack of support to dates in four digits format is the bugs' root cause. Therefore, people reviewed adapted and fixed software applications worldwide to avoid unexpected consequences and disasters.

Y2K is a typical case of a known bug where the problem is well defined; however, the actual demand is on selecting the best strategy to localize the bug and apply a patch to the source code. Therefore, the localization strategy was crucial in Y2K since the chosen approach would directly impact the effort demanded to find the right point to fix the bug.

1.1 Motivation

Bug Localization (BL) is a typical Software Engineering maintenance activity. It consists of finding where to fix the source code, starting from a bug report describing the observed misbehavior in the software functionality. Then, the developer can proceed with the localization process with or without some automation support. This work concentrates on the automated strategies for BL that tries to rank the most suspicious piece of code, where the developer should point out to fix the bug.

BL precedes a buggy source code's effective patching (a.k.a. fixing or repair). Initially, strategies to localize a bug mainly were ad hoc. Even today, we can find developers that do not know anything about a specific or well-known method to proceed with the

BL. In these scenarios, developers rely on the most basic strategies like searching and reviewing the code repository, consulting and keeping in touch with the code "owners", or looking for similar bugs or situations to get some insight. However, many prototype tools and approaches are in development to facilitate and help automate this essential step in software maintenance and debugging tasks. Unfortunately, these solutions' popularity and extensive use are still far from most developers' reality. Nevertheless, the Automated Program Repair (APR) area (MOTWANI et al., 2018; MONPERRUS, 2018; GAZZOLA; MICUCCI; MARIANI, 2019) has shown some evolution and brings hope for a stage where fixing a software bug is behind a simple click in the development environment.

A good BL approach is critical to guide the developer towards the right point to fix a bug, reducing the time and effort to debug, whether manually or with automatic program repair approaches (LIU et al., 2019). Nevertheless, developers can quickly reject the strategy if it is inaccurate and unreliable. Kochhar et al. (2016) reports a minimum success rate (or trustfulness) around 75% for a BL approach satisfy the needs of most of the professional developers participating in the study (KOCHHAR et al., 2016). This minimum threshold is still far to be reached by most of the approaches for BL still reporting the best results on the range of 20% to 70% success rate (PEARSON et al., 2017; SHI et al., 2018; KHATIWADA; TUSHEV; MAHMOUD, 2020; HUO et al., 2019). To increase the challenge and even with the evolution of the debugging research and practices, many developers still do not receive formal education in debugging area and have to learn by doing, with pairs, or by self-teaching (SIEGMUND et al., 2014). On the tools' side, we are currently far from the "killer" tool to support developers in debugging activities, which is even more severe to BL approaches and tools (PARNIN; ORSO, 2011). Asking for a good IDE for development would end with default answers like Eclipse, IntelliJ, Netbeans, VS Code, and many already popular and widely used tools. However, asking for an excellent tool for BL, it is no surprise that many developers do not even know about this kind of tool. Most of what we find in this area is still a work in progress and is not mature enough to become a universal and broadly applicable solution. So, a long journey of maturity remains to satisfy the high demands of a professional environment, considering not only improvements in suspicious ranking precision but also requirements related to scalability, efficiency, IDE integration, and others (KOCHHAR et al., 2016).

The studies on BL started some decades ago (COUSOT; COUSOT, 1977) and employ a diverse set of techniques such as the classical Information Retrieval models as LSI and LDA (POSHYVANYK et al., 2007; NICHOLS, 2010; ZHOU et al., 2012). Later, and

some years from now, researchers began to explore Machine Learning (ML) techniques to improve the performance of BL approaches (BRIAND; LABICHE; LIU, 2007; JEFFREY et al., 2009; NAMIN, 2015; HUO; LI; ZHOU, 2016; LAM et al., 2017) and also in more recent works (BARBOSA et al., 2019; LI; WANG; NGUYEN, 2021; LOU et al., 2021; HUO et al., 2019). However, state-of-the-art performance is far from a sound performance, e.g., Ye, Bunescu e Liu (2016) reaches around 0.5 with MAP measure, while more recent work is still far from a perfect score, e.g., Huo et al. (2019) reach 0.64 with the same MAP measure and using recent Deep Learning strategies. Moreover, even considering that BL approaches are "potentially" promising to support developers in the search for buggy source code files from a bug description (i.e., usually through bug reports), the possible false negatives would derail the widespread use of such approaches. Thus, the understanding of many factors related to the BL approaches' performance assessment would help to explain the actual results.

1.2 Objectives

1.2.1 General Objectives

The main objective of this work is to contribute to software repair activities with a focus on automated approaches for Bug Localization (BL)[1] through the analysis of bugs characteristics in assessment datasets, primarily those related to bug's patches.

1.2.2 Specific Objectives

1. Explore alternatives for the improvement on the accuracy of rankings of suspicious software components produced by automated BL approaches (Chapters 4 and 5);

2. Show how different BL approaches based on Machine Learning techniques behave with diverse parameter tuning configurations (Chapter 5);

3. Show how the patch analysis can help in the characterization of a bug dataset, exposing characteristics that would help to guide experimentation to evaluate BL approaches (Chapter 6);

4. Define taxonomy and criteria for characterization of bug dataset through its patches (Chapter 6 with the extension of (SOBREIRA et al., 2018));

[1] a.k.a. Fault Localization (FL) or Fault Location

5. Show the dissection of a large bug dataset and compare with the original results from a smaller one, used as the base for our initial findings (Chapter 6);

6. Show correlation between the characteristics of bugs in a dataset and the observed performance of the BL approaches (and also on the features scores they are based on) (Chapter 7);

7. Reproduce previous BL approaches and guide the experiment sampling with specific bug patch characteristics, for example, repair actions and repair patterns (Chapter 7).

1.3 Thesis Declaration Proposal

The characteristics of bugs in specific datasets influence the accuracy of Bug Localization (BL) techniques, so dataset dissection supported by patch analysis and a well-defined taxonomy help to: 1) guide the technique configuration, 2) interpret the results, and 3) shed light on future research.

1.4 Research Summary, Assumptions, and Questions

The performance analysis of a BL approach would require the execution of experiments to produce data either with the use of a reproduction package (when available and up to date) or with the implementation of a new experimental package (especially when we plan for new settings, alternatives, and other customizations). Since our work involves many customizations and the test of many alternative settings, we opted to develop our experimental package. Still, we choose the work of Ye, Bunescu e Liu (2016) for our experiments baseline and also as a starting point for the experimental package implementation. Some reasons for the choice were the combination of ideas from other approaches, including the work of Saha et al. (2013), a reasonable number of extracted features, and the use of a composition mechanism based on LtR algorithms that gives some flexibility to introduce new features. So, we assume this context provides a good test-bed for experimenting with BL strategies involving many information sources.

We first enumerated factors of influence on a BL approach. As a proof of concept, we can 1) apply different LtR algorithms to observe the impact of parameter tuning on the ranking scores, 2) include new features (e.g., Code Entropy) for comparisons,

3) compare the impact and the role of individual and groups of features on the final ranking. Therefore, we conduct some preliminary experiments while developing first version of our experimental package to answer the following Research Questions (RQs):

RQ1 What is the performance of the entropy feature compared to other features?

RQ2 The use of entropy feature can improve the results obtained by past learning approaches to BL?

RQ3 What is the impact of data balance strategies in the learning process?

RQ4 How does the tuning of LtR algorithms impacts the BL performance?

RQ5 How long does it take to conclude each step in the process (feature extraction, ranking generation, training, validation, and testing)?

After observing the potential influence of the bugs' characteristics composing the assessment datasets of BL approaches, we conduct some studies to analyze, define, and propose a taxonomy for these characteristics. We refer to these kinds of studies as a *dataset dissection*, first made with Defects4J (JUST; JALALI; ERNST, 2014), and the subsequent study with LR–dataset (YE; BUNESCU; LIU, 2014), detailed in this thesis. From these studies, we observed that many types of bugs with different characteristics are present in a dataset used for the assessment of BL approaches. These characteristics are common, frequent, recurrent, and prevalent even between different projects. Therefore, knowing the dataset composition can support more informed decisions regarding many aspects of dataset usage. Between the possible applications for research of our dissection analysis framework, we can mention 1) BL, 2) APR, 3) dataset comparisons and benchmarking, 4) Source Code, Bug, and Patch Analysis, 5) Software Testing, 6) Debugging, and 7) Program Synthesis. Aligned with the thesis declaration proposal (our main hypothesis), we would assume the composition of a sample from a bug dataset can influence the assessment measures of the target task (e.g., BL approaches) depending on the selected bugs' nature and characteristics. Additionally, this influence would impact 1) the selection of techniques to improve BL scores accuracy, 2) on the understanding of the performance variations, 3) on the obtaining of more practical insights to improve the past approaches, and 4) on the proposition of new BL approaches with more informed decisions. So, the combination of the study dissection study with extensions to the experimental package after the preliminary experiments helped to answer additional research questions related to the effective influence of the sampled bugs on the BL strategies, and that helped to confirm some of these assumptions:

RQ6 When we compare a sample of bugs where the respective patches match a given repair pattern against another sample of bugs where this pattern is not present, is there any difference in the measured metrics targeting the ranking of bug suspects? Are these differences statistically significant?

RQ7 What type of impact is associated with the evaluated metric's score rankings by the presence of a repair pattern in the patches of a bug sample? Moreover, when the repair pattern is absent?

RQ8 What is the degree of the impact correlated to the repair pattern's presence or absence on the metrics measured?

1.5 Contributions

1. A proof of concept showing the application of a new feature based on Code Entropy and LtR to produce BL scores;

2. A proof of concept showing the application of different data balance strategies on the training with LtR algorithms to produce BL scores;

3. A proof of concept showing the influence of parameters tuning on LtR–based BL scores;

4. A new approach to deal with the assessment of BL methods using bug datasets and benchmarks guided by bug characteristics;

5. A taxonomy to characterize bug datasets in terms of their patches composition (in collaboration);

6. A tool to extract patch characteristics from a bug dataset, e.g., repair action, repair patterns, and size dimensions (in collaboration);

7. A proof of concept showing the influence of the bugs types from a dataset for the assessment of research approaches on typical software development tasks like BL;

8. An experimentation package prototype for BL approaches that would progress in future works for an experimentation framework to support assessment of past and new approaches.

1.6 Thesis Organization

This chapter briefly described the motivation, objectives, hypothesis, and expected contributions. Chapter 2, *Background*, presents the essential concepts related to the BL research, including bug and faults, bug reports, bug datasets, evaluation metrics, and some ML techniques applied in experiments with BL approaches. Chapter 3, *Related Work*, presents some of the published works in BL, split into Static, Dynamic, and Hybrid approaches. From Chapter 4 to Chapter 7, we mainly detail the developed work to answer the proposed hypothesis in this thesis. Chapter 4, *On the Influential Factors for Bug Localization Exploratory Assessment*, raises many ideas to apply in the construction of an environment to experiment with BL approaches, started in this work as an experimental package, briefly described, that support all the experimentation in this thesis. Chapter 5, *Strategies for Learn-to-Rank Bug Localization Improvement*, presents some proof of concept to assess alternatives in BL with the preliminary results obtained, experimenting ideas related to the application of Code Entropy feature, optimum selection of features, tuning of ML hyper-parameters in LtR algorithms and the use of data balancing strategies. Chapter 6, *Analysis of Repair Actions and Patterns*, takes a step back in the dataset issues and presents an analysis on a regularly applied bug dataset extending previous work on Defects4J, smaller popular bug dataset, and benchmark, to analyze LR–dataset, a larger dataset applied in BL context. We searched for understanding how the patches associated with bugs in a dataset are classified, their characteristics, the existence of patterns, and how prevalent these characteristics are. Chapter 7, *Influence of Repair Patterns on Bug Localization approaches*, continues the exploratory analysis and presents additional experiments considering the selection of sampling data based on the bug characteristics presented in Chapter 6, and showing how this would impact the BL assessment, guided by bug patches characteristics, especially, repair actions and repair patterns. Chapter 8, *Conclusion*, summarizes and highlights the main points in this work, reinforcing the thesis declaration proposal.

Figure 1 show a brief thesis roadmap. After the discussion in Chapter 4 about the influential factors, we present two alternative tracks to explore the BL problem. The first in Chapter 5 show some experiments related to LtR BL considering the datasets of bugs as usually done in previous approaches. The next track starts in Chapter 6 where we present an extension of a dataset dissection analysis, first defined for Defects4J, and then applied to LR–dataset. Following in Chapter 7, we present results of experiments with BL applying the dissection analysis ideas. We consider this last track one of our main innovative contributions to the research community since it opens a new lens to

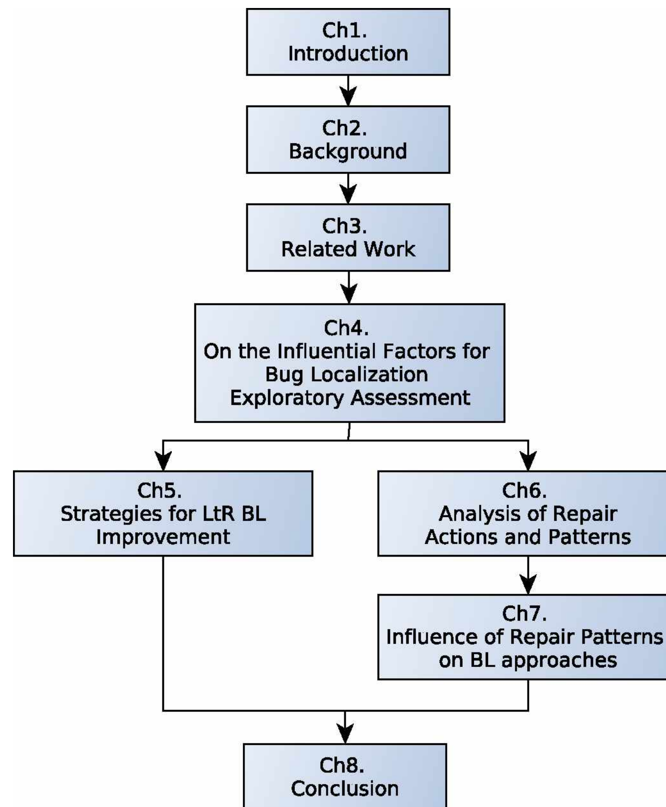


Figure 1 – Chapters Roadmap.

review and improve the previous approaches for BL, and potentially other related areas such as APR.

Chapter 2

Background

Terms as *bugs* have different meanings depending on the context. Furthermore, even considering a specific knowledge area as Software Engineering, sometimes these terms are abstract and confusing. Therefore, before discussing *how to find bugs?*, we need first to define 1) what we consider a *bug*?, 2) how to measure how successful we would be with a given strategy to find a bug?, 3) what are our ground truths to assess a BL strategy? (i.e., what database we would apply?). This chapter discusses these and other essential concepts. Additionally, we briefly present some concepts and ML techniques applied to BL and, more specifically, applied in some of the experiments described in further chapters.

2.1 Essential concepts about Bug Localization

This section introduces basic concepts and definitions related to the BL context for the subsequent chapters.

2.1.1 What is a bug?

Thomas Edison helped to coin the term *bug*, referring to a technical problem in hardware engineering, while dealing with real (or imaginary) bugs that disturbed him during the working on his inventions in the 19th-century (MAGOUN; ISRAEL, 2013). The term gained the computer world in 1946, after the discovery of a “real” bug in the circuits of the electromechanical computer, Mark II, built and programmed by Howard Aiken and Grace Hopper (KIDWELL, 1998). Operators found the bug and the cause of Mark II’s errors, a moth, removed from the circuits and now taped in the logbook for the History. The event was referenced as the “First actual case of bug being found”. The

term *debug* has a similar etymology, first used in 1945 in the context of aircraft engines, according to with Oxford English Dictionary. The software development world has the *debugging* process well established, widely used, and associated with tasks of removing bugs or faults from a software system.

ISO/IEC/IEEE 24765 standard defines the term *fault* as "1. a manifestation of an error in software. 2. an incorrect step, process, or data definition in a computer program. 3. a defect in a hardware device or component." (ISO/IEC/IEEE, 2010). The term *fault* is also a synonym to the term *bug*. Thus it is usual to find works using both terms to express the same meaning. In this work, the choice for the *bug* term reflects the more widely and popularized use against *fault*, which also has a diverse meaning in other contexts and areas.

2.1.2 Bug Reports

A Bug Report is a formal registration of an issue found in software. Many bug-tracking platforms help users and developers to create and manage these reports. Some popular platforms in use today are: Bugzilla[1], JIRA[2], GitHub Issues[3], and FogBugz[4]. Beyond the differences, these platforms share common and essential resources to store critical information about the found issue, allowing the developer to understand and, preferably, reproduce a bug, then proceed to localization and fixing tasks.

Figure 2 shows an example of a bug report for the bug LANG-552 from the project Apache Commons Lang. The report describes the observable bug behavior according to the user (or developer) perspective, aiming to help the maintainer search and understand the problem and consequently proceed to the code fixing. Furthermore, this type of bug report facilitates finding the fixing location since the reporter points to the method triggering the error. However, few bug reports suggest the bug localization with good precision and providing precise and complete information in practice.

The code listing in Figure 3 shows the patch applied to fix the LANG-552 bug[5]. The fixing consists of adding three lines of code (highlighted in green) with a missing null check in the buggy code. The patch makes the program continue to the next loop step if the conditional testing expression is satisfied.

---

1 Bugzilla: <www.bugzilla.org>

2 JIRA: <br.atlassian.com/software/jira>

3 GitHub Issues: <guides.github.com/features/issues>

4 FogBugz: <www.fogcreek.com/fogbugz>

5 The patch for LANG-552 is available in Defects4J Dissection website: <http://program-repair.org/defects4j-dissection/#!/bug/Lang/39>

Bug-ID: LANG-552
Title: StringUtils replaceEach - Bug or Missing Documentation
Description: The following Test Case for replaceEach fails with a null pointer exception. I have expected that all StringUtils methods are "null-friendly". The use case is that i will stuff Values into the replacementList of which I do not want to check whether they are null. I admit the use case is not perfect, because it is

unclear what happens on the replace. I outlined three expectations in the test case, of course only one should be met. If it is decided that none of them should be possible, I propose to update the documentation with what happens when null is passed as replacement string

```
import static org.junit.Assert.assertEquals;
import org.apache.commons.lang.StringUtils;
import org.junit.Test;

public class StringUtilsTest {
        @Test
        public void replaceEach(){
                String original = "Hello World!";
                String[] searchList = {"Hello", "World"};
                String[] replacementList = {"Greetings", null};
                String result = StringUtils.replaceEach(original, searchList,
                    replacementList);
                assertEquals("Greetings !", result);
                //perhaps this is ok as well
                //assertEquals("Greetings World!", result);
                //or even
                //assertEquals("Greetings null!", result);
        }
}
```

Figure 2 – *Bug Report* for the bug LANG-552 from Apache Commons Lang project.

Some studies aim to define the essential information for a report that directly impacts its usefulness to solve a problem. For example, according to (SASSO; MOCCI; LANZA, 2016), the elapsed time between the bug report creation and the bug fixing is directly related to the fulfilling quality of the following fields: summary, description (including stack traces and screenshots), due date, and people involved (report creator, allocated developer to fix, who found the issue). The same study considers a bug report change between the states: new/open, not confirmed, in progress/assigned, patch available, verified, resolved, reopened, and closed. To the authors, customization resources, project, and platforms specificities do not contribute too much in practice to the usefulness of a bug report and are also poorly used.

Bettenburg et al. (2008) have found mismatches between what users fulfill in a bug

src/java/org/apache/commons/lang3/StringUtils.java CHANGED

```
          @@ -3673,6 +3673,9 @@ private static String replaceEach(String text, String[] searchList, String[] rep
3673 3673
3674 3674          // count the replacement text elements that are larger than their corresponding text being replaced
3675 3675          for (int i = 0; i < searchList.length; i++) {
     3676 +            if (searchList[i] == null || replacementList[i] == null) {
     3677 +                continue;
     3678 +            }
3676 3679              int greater = replacementList[i].length() - searchList[i].length();
3677 3680              if (greater > 0) {
3678 3681                  increase += 3 * greater; // assume 3 matches
```

Figure 3 – *Patch* for the bug LANG-552 from Apache Commons Lang project.

report and what developers consider as helpful. To developers, the essential items are steps to reproduce, stack traces, test cases, and observed behavior. Information such as hardware, bug severity, component, and the operating system is rarely used, even as mandatory fields in many tracking platforms. Errors in reproduction steps and incomplete information are critical problems for developers. Usually, developers and users agree on the top-3 most useful and fulfilled fields in the reports (steps to reproduce, observed and expected behavior). The disagreement starts from the fourth item (stack trace versus product, test cases versus version, and others). Except for the reproduction steps, there is a significant mismatch between what developers consider most important and what users provide in practice through the report. When there is an agreement between developers' and users' usefulness notion, the lack of some fields in a bug report is more related to the difficulty of obtaining these kinds of information (e.g., stack trace and test cases) than user carelessness. Therefore, developers highlight the importance of clear, correct, and complete information in bug reports. Following factors are used to measure the quality of a bug report: the use of itemization, keywords related to important categories (action items, observed/expected behavior, steps to reproduction, build, and user interface elements), code samples, stack trace, patches, screen captures, and readability (based on standard measures such as SMOG Grade). Finally, the readability, stack traces, and code samples correlate to less time to close a bug report (after fixing the reported bug).

The bug report is one of the possible starting points to proceed with bug localization, whether done manually or automated through a tool (especially in static approaches based on Information Retrieval (IR) and ML). Thus, it is essential to carefully consider the information present in a bug report, including its content quality, because it can be determinant in the success of the bug localization.

2.1.3 Where is a bug located? What is it found?

Lucia et al. (2012) preliminary study presents a notion of locality for a bug related to the spreading level of the buggy components in the codebase. The authors define four locality levels: L_{D1}, the number of faulty lines; L_{D2}, the number of faulty methods; L_{D3}, the number of faulty files; L_{D4}, a score based on the sum of lines between the first and the last faulty line found for each buggy file. The analysis is done over 374 manually selected bugs from three Java systems (AspectJ, Rhino, and Lucene). Most of the analyzed bugs are localizable in the sense the most of them is concentrated in few lines ($L_{D1} <= 10$ lines in more than 80% of the bugs, and $L_{D1} = 1$ lines in 33% of the bugs), few methods ($L_{D2} <= 6$ method in more than 83% of the bugs, and $L_{D2} = 1$ lines in 44% of the bugs), and few files ($L_{D3} <= 2$ lines in more than 88% of the bugs, and $L_{D3} = 1$ lines in 73% of the bugs). The fourth dimension shows a good spreading of the buggy lines since the score $L_{D4} < 1000$ for around 90% of the bugs.

We can analyze the composition of a bug from the perspective of repair actions required to fix the bug. Liu et al. (2018) define a repair action as a combination of code entities and change operators. The code entities are the nodes found in an Abstract Syntax Tree (AST) as parsed by Eclipse JDT AST Parser[6]. Three main categories are highlighted in the paper: statements (22 types, e.g., ReturnStatement), declarations (total not informed in the paper, e.g., TypeDeclaration), and expressions (35 types, e.g. InfixExpression). The change operators follows the GumTree tool definition (FALLERI et al., 2014), applied to produce the AST diff between the buggy and fixed source code versions. GumTree change operators considered were: update, insert, delete and move. The Figure 4 illustrate the AST for the patch fixing the LANG-552 bug (shown in Figure 3) and generated by GumTree. This patch represents the insertion of a Block-IfStatement including a series of child insertion actions for the following code elements: IfStatement (e.g., if), InfixExpression (e.g., $searchList[i] == null$), ArrayAccess (e.g., $searchList[i]$), SimpleName (e.g., $seachList$), InfixExpressionOperator (e.g., $==$, NullLiteral (e.g., $null$), Block (e.g., $\{continue;\}$), and ContinueStatement (e.g., $continue;$).

The addition of code as illustrated for the bug LANG-552 is one of the possible scenarios for patches applied to fix bugs. The Defects4J Dissection website[7] gives us easy access to view the patches like LANG-552 (or Lang-39 in Defects4J), so we refer to Defects4J bug identifiers for illustrations. The other common scenarios are: patches removing some buggy code as in Figure 5 (Math-58), patches modifying buggy code

[6] Eclipse JDT AST Parser: <https://help.eclipse.org/kepler/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html>

[7] Defects4J Dissection website: <http://program-repair.org/defects4j-dissection/>

```
IfStatement [149033,149131]
    InfixExpression [149037,149088]
        InfixExpression [149037,149058]
            ArrayAccess [149037,149050]
                SimpleName: searchList [149037,149047]
                SimpleName: i [149048,149049]
            INFIX_EXPRESSION_OPERATOR: == [149051,149053]
            NullLiteral [149054,149058]
        INFIX_EXPRESSION_OPERATOR: || [149059,149061]
        InfixExpression [149062,149088]
            ArrayAccess [149062,149080]
                SimpleName: replacementList [149062,149077]
                SimpleName: i [149078,149079]
            INFIX_EXPRESSION_OPERATOR: == [149081,149083]
            NullLiteral [149084,149088]
    Block [149090,149131]
        ContinueStatement [149108,149117]
```

Figure 4 – GumTree generated AST representing the patch for the bug LANG-552 shown in Figure 3.

as in Figure 6 (Math-41), patches moving some code to other positions as in Figure 7 (Closure-13), and patches mixing all these scenarios as in Figure 8 (Lang-17). These selected scenarios (and bugs) have illustrative purposes. Still, the bug universe goes far beyond these examples, and we can think of them as building blocks for bugs requiring more complex patches and with even more repair actions than those shown.

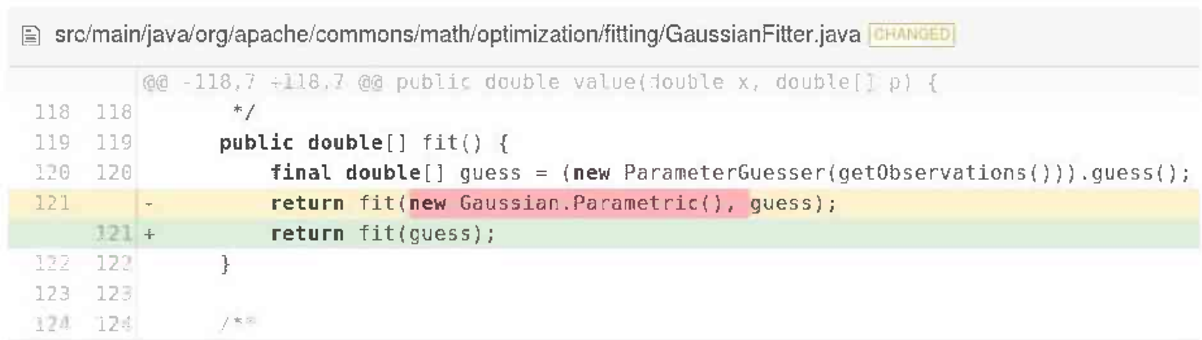


Figure 5 – *Patch* for the bug Math-58 from Apache Commons Math project.

2.1.4 How to find bugs?

The Bug Localization (BL) is one of the fundamental steps in the software fixing process (PARNIN; ORSO, 2011). BL contributes to the considerable time, and effort demanded in this process (HAMILL; GOSEVA-POPSTOJANOVA, 2017), and also to the challenge to conduct and to complete this process with success, especially for the novices (MCCAULEY et al., 2008). The BL consists in the identification of places of

```
src/main/java/org/apache/commons/math/stat/descriptive/moment/Variance.java CHANGED
@@ -517,7 +517,7 @@ public double evaluate(final double[] values, final double[] weights,
517 517             }
518 518
519 519             double sumWts = 0;
520     -           for (int i = 0; i < weights.length; i++) {
    520 +           for (int i = begin; i < begin + length; i++) {
521 521                 sumWts += weights[i];
522 522             }
523 523
```

Figure 6 – *Patch* for the bug Math-41 from Apache Commons Math project.

```
src/com/google/javascript/jscomp/PeepholeOptimizationsPass.java CHANGED
@@ -123,8 +123,8 @@ private void traverse(Node node) {
123 123     do {
124 124       Node c = node.getFirstChild();
125 125       while(c != null) {
126     -         traverse(c);
127 126         Node next = c.getNext();
    127 +         traverse(c);
128 128         c = next;
129 129       }
130 130
```

Figure 7 – *Patch* for the bug Closure-13 from Closure Compiler project.

bugs (faults or errors) in the source code and causing the system to fail. The failure usually manifests as a wrong behavior that differs from the expected behavior by the users and developers, implying the system breakdown and significant losses.

There is still no standard way widely used to deal with BL, although there are lots of approaches and different proposals (WONG et al., 2016). Ultimately, it is up to the developer to locate and fix identified faults. In many environments, this is done essentially manual way with only general-purpose tool support provided by an Integrated Development Environment (IDE). However, with the increase in the size and complexity of software, it is not enough to rely only on the developers' experience, judgment, intuition, and familiarity to locate the bugs. Although it is a good resource (i.e., when an expert is always available to help), relying solely on the human ability to perform this task is not always feasible. There is then a strong demand for the production of new techniques and tools that support and even automate BL.

Since 1970's, research community has been studying and developing techniques to support BL (COUSOT; COUSOT, 1977). Traditional approaches involves: logging (EDWARDS, 2003), assertions (ROSENBLUM; ROSENBLUM, 1995), breakpoints (COUTANT et al., 1988) and profiling (BALL; LARUS, 1994). Due to the amount of analysis and

src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java CHANGED

```
        @@ -80,26 +80,20 @@ public final void translate(CharSequence input, Writer out) throws IOException {
 80  80                 return;
 81  81             }
 82  82             int pos = 0;
 83     -           int len = Character.codePointCount(input, 0, input.length());
     83 +           int len = input.length();
 84  84             while (pos < len) {
 85  85                 int consumed = translate(input, pos, out);
 86  86                 if (consumed == 0) {
 87  87                     char[] c = Character.toChars(Character.codePointAt(input, pos));
 88  88                     out.write(c);
     89 +                   pos+= c.length;
     90 +                   continue;
 89  91                 }
 90     -               else {
 91  92    //           // contract with translators is that they have to understand codepoints
 92  93    //           // and they just took care of a surrogate pair
 93  94                 for (int pt = 0; pt < consumed; pt++) {
 94     -                   if (pos < len - 2) {
 95  95                     pos += Character.charCount(Character.codePointAt(input, pos));
 96     -                   } else {
 97     -                       pos++;
 98     -                   }
 99     -               }
100     -               pos--;
101  96                 }
102     -               pos++;
103  97             }
104  98         }
105  99
```

Figure 8 – *Patch* for the bug Lang-17 from Apache Commons Lang project.

manual work, such techniques become ineffective given the increase in software complexity and size (WONG et al., 2016). This requires the production of more advanced techniques, able to deal better with these issues. These techniques can be based on: program slicing (WANG et al., 2014), spectrum (WONG et al., 2014b), statistics (CHILIMBI et al., 2008), program states (SUMNER; ZHANG, 2013), machine learning (BRIAND; LABICHE; LIU, 2007), data mining (CELLIER et al., 2011), program models (BAAH; PODGURSKI; HARROLD, 2011), and other models (SAHA et al., 2013). The next chapter details some of the most recent and remarkable approaches to BL.

2.2 Bug datasets

The BL literature references many bug datasets. Some of these datasets are collections of projects and software repositories, initially selected to serve evaluation purposes of specific works but recurrent applied a posteriori to make possible comparisons. Other datasets are conceived and designed for reuse, becoming base references or benchmarks

to facilitate comparisons between the approaches. LR-dataset is an example of the first case, while Defects4J is the second case. The following subsections present a brief description of these datasets.

2.2.1 Defects4J

Defects4J (JUST; JALALI; ERNST, 2014) is a benchmark of bugs developed to support research, especially in the areas of software testing, automated software repair, and bug localization. The benchmark comprises a dataset of bugs and a command-line interface to facilitate the exploration of this dataset (e.g., query information about the bugs such as: affected classes/tests cases, buggy/fixed source code versions, and checkout of the associated projects). The 395 real bugs (version 1.1) initially extracted from 6 Java open-source projects are Apache Commons Lang (65 bugs), Apache Commons Math (106 bugs), Closure Compiler (133 bugs), JFreeChart (26 bugs), Joda Time (27 bugs), and Mockito Testing Framework (38 bugs). Moreover, Defects4J bugs are 1) related to source code (i.e., excluding fixes within the build system, configuration files, documentation, or tests), 2) reproducible (each bug contains at least one test that exposes the bug), and 3) isolated (patches do not include unrelated changes to the bugs such as features or refactorings). Many works on BL employed Defect4J to evaluate their approaches (LE et al., 2016; PEARSON et al., 2017; PEREZ et al., 2017; LI; ZHANG, 2017; JUST et al., 2018; CHAKRABORTY et al., 2018).

2.2.2 LR-dataset

Ye et al. (YE; BUNESCU; LIU, 2014; YE; BUNESCU; LIU, 2016) propose a new approach for BL based on LtR and a companion new benchmark dataset to evaluate the approach. The dataset maps a total of 22747 bug reports (that implies the same amount of bugs) based on 6 Java open-source projects: AspectJ (593 bugs), BIRT (4178 bugs), Eclipse Platform UI (6495 bugs), JDT (6274 bugs), and SWT (4151 bugs) and Tomcat (1056 bugs). The amount of bugs in this dataset is far more than the amount found in Defects4J, turning LR-dataset a better target to ML approaches. The criteria to collect bug reports and to consider the respective bugs in the dataset were: 1) bug reports have the status *resolved fixed*, *verified fixed*, or *closed fixed*; 2) there are explicit mentions to terms like *bug <bug-id>* or *fix for <bug-id>* in the project changelogs; 3) the bug reports are associated to a single fixing Git or revision commit, not shared with other bug reports; 4) the inclusion of only functional bug fixings (i.e., at least one fixed file should exist).

The selected bug reports from each project are available in XML files containing the fields: bug id, summary, description, report date/time (and timestamp), status, commit (and timestamp), files (fixed), and ranking position obtained in the approach. All the projects repositories are in GitHub, allowing to obtain the associated versions/revisions through the commit information. Many authors in BL field applied LR-dataset in their approaches' evaluation (LAM et al., 2015; UNENO; MIZUNO; CHOI, 2016; LAM et al., 2017; YE; BUNESCU; LIU, 2016; ZHAO et al., 2015; ALMHANA et al., 2016).

2.3 Performance metrics in Bug Localization

Many metrics apply to the performance assessment in BL approaches. Some of the more often found are Precision@k, the percentual of success on finding relevant items in the top positions of a ranking limited to N items (Top-N), Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and Normalized Discounted Cumulated Gain (NDCG) that are someway complementary measures. Other metrics appear less frequently, but it is important to be clarified. Many of these metrics are classical IR evaluation measures and are associated to the notion of *relevance* of a *document* picked from a larger *collection* given an input *query* with the desired information needs (MANNING; RAGHAVAN; SCHÜTZE, 2008). In the BL context of this work, a relevant document given a query is equivalent to a buggy file given a bug report, while an irrelevant document is a non-buggy file for the same bug report. The document collection is equivalent to the source code files considered for the given bug report and extracted from a specific project version or revision.

2.3.1 Precision, Recall and F-measure

Although *precision*, *recall* and *F-measure* are very common IR measures, these are set-based measures and are computed using an unordered set of documents (MANNING; RAGHAVAN; SCHÜTZE, 2008). Then, it is not so common to find these measures in the evaluation for ranking problems. Besides this, some of the ranking measures, such as MAP, extend set-based measures to support the evaluation of ranking-based IR techniques.

Precision metric, defined in Equation 1, is the fraction of relevant documents in a retrieved set r, given a query q. *Precision* increases if the number of relevant documents increases and also if the number of retrieved documents decreases. For example, considering a retrieval set of just a single document, the *Precision* is maximal (1) if the

document is relevant and minimal (0) if the document is not relevant. Otherside, for a larger retrieval set, if there is only one relevant document retrieved, but a huge number of not relevant documents retrieved, the *Precision* can approximate to zero, even considering that the ideal target for the query is in the retrieved set of documents. So, *Precision* receives a strong influence from the number of retrieved docs.

$$Precision(q,r) = \frac{\#relevant\ retrieved\ docs}{\#retrieved\ docs} \tag{1}$$

Recall metric, defined in Equation 2, is a complementary measure to *Precision*, with more emphasys in the number of relevant documents retrieved when considered all the existent relevant documents.

$$Recall(q,r) = \frac{\#relevant\ retrieved\ docs}{\#relevant\ docs} \tag{2}$$

Taken isolated, *precision* and *recall* can be misleading since it is possible to have a high performance of one, while the other shows poor performance, and vice versa. A metric that trades off between these measures is the *F-measure*, defined in Equation 3. *F-measure* represents the weighted harmonic mean between *precision* and *recall*. The α parameter weights how much the final value tends to the precision or the recall.

$$F\text{-}measure(q,r) = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \tag{3}$$

where, $P = Precision(q,r)$, and $R = Recall(q,r)$.

2.3.2 Precision@k

The *precision* metric defined in Equation 1 is not used directly in ranking problems, especially in the BL problem, since the number of considered files depends on the size of the project and the user is not interested in analyzing all the files or a big list of files. The *precision* metric is often limited to a small number of the top K retrieved files. The frequently found values for K are 1, 5, 10, or 20. Equation 4 is an adapted version of Equation 1, considering the BL context, a single bug report br, and a given ranking of files r.

$$P@k(br,r) = \frac{\#retrieved\ buggy\ files}{K} \tag{4}$$

In practice, P@k is computed for all the bug reports of interest and averaged to provide the final evaluation measure. Given a set of bug reports B, from where we can

extract tuples containing bug reports *br* and their respective rankings r limited to K files, Equation 5 defines *Precision@k*.

$$Precision@k = \frac{1}{|B|} \sum_{br \in B} P@k(br, r) \tag{5}$$

2.3.3 Top-N

a.k.a.: Accuracy@k, Recall at Top-N

Considered as in Equations 4 and 5, the *Precision@k* measure could worsen, merely increasing the k value (remembering that most of the bugs restrict to a few files, typically less than 5). Therefore, a common assumption adopted by most of the approaches is to consider an alternative definition, shown in Equation 6. Thus, the bug is found for a given bug report *br* if at least one buggy file is between the N files in the retrieved rank r.

$$top(br, r) = \begin{cases} 1, & \text{if found buggy file is in the top } n \text{ files} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

As in *Precision@k*, the percentual of success on finding relevant items in the top positions of a ranking limited to N items (Top-N) is computed for all the bug reports and averaged. The final Top-N measure, given a set of bug reports B, is shown in Equation 7.

$$\textit{Top-N} = \frac{1}{|B|} \sum_{br \in B} top(br, r) \tag{7}$$

The idea behind Top-N for BL is to measure how accurate a tool is in presenting at least one buggy file using a Top-N ranking of files.

2.3.4 MAP

Mean Average Precision (MAP) is a commonly found measure for ranking approaches. The ranking positions of all the buggy files are accounted for and expressed as the *Average Precision* (AP) for each bug report. Then, we compute the mean based on the AP for all the bug reports. Given a bug report *br* and a ranking of files r, Equation 8 defines the associated AP.

$$AP(br, r) = \frac{\sum_{k=1}^{L} P@k(br, r) * is_buggy(k, br, r)}{\#buggy\ files} \tag{8}$$

The parameter L indicates the last position of a buggy file that should be found for the bug report br. $P@k(br, r)$ is the precision at k^{th} position, given br and r. The function $is_buggy(k, br, r)$ is defined in Equation 9.

$$is_buggy(k,br,r) = \begin{cases} 1, & \text{if the doc position } k \text{ in } r \text{ is a buggy file (relevant)} \\ 0, & \text{otherwise (not relevant)} \end{cases} \tag{9}$$

Given a set of bug reports B, from where we can extract tuples containing bug reports br and their respective rankings r, Equation 10 defines MAP.

$$MAP = \frac{1}{|B|} \sum_{br \in B} AP(br, r) \tag{10}$$

The idea in the context of BL is to give a notion of effort a developer would have to examine all the buggy files, given their positions in the retrieved ranking of files. If the files near the top of the ranking are buggy, the effort is lower, and the MAP score is high. Otherwise, the more the buggy files are distant from the top positions, the more effort the developer has, and the lower is the MAP score.

2.3.5 MRR

While similar to MAP, Mean Reciprocal Rank (MRR) have some particularities: in the retrieved ranking, MRR considers only the first best-positioned buggy file; and as in Precision@k, MRR uses just one buggy file in the computing, but there is no K-limit for the rank of retrieved files. Given a set of bug reports B, from where we can extract tuples containing bug reports br and their respective rankings r, Equation 11 define MRR.

$$MRR = \frac{1}{|B|} \sum_{br \in B} \frac{1}{rank(br, r)} \tag{11}$$

Where $rank(br, r)$ returns the position of the first buggy file for the bug report br and ranking r.

As occurs in MAP, MRR also gives a notion of effort to examine buggy files. The main difference is that MRR only accounts for the first buggy file found in the ranking.

2.3.6 NDCG

a.k.a.: NDCG@k

Normalized Discounted Cumulated Gain (NDCG) is a measure defined especially for multi-graded ranking problems (JÄRVELIN; KEKÄLÄINEN, 2002), i.e., problems where the relevance of the documents can receive more levels than merely relevant or irrelevant. The first component in NDCG, the *Cumulative Gain* (CG), represents the sum of the grades, G, attributed to each file in a ranking. For binary grades, as in BL, it is equivalent to the number of retrieved buggy files in the ranking (where $G = 1$ for buggy files and $G = 0$ for non-buggy files). The Equation 12 shows a recursive function to compute the CG, for a given position i in the ranking.

$$CG(i) = \begin{cases} G(1), & \text{if } i = 1 \\ CG(i-1) + G(i), & \text{otherwise.} \end{cases} \tag{12}$$

The *Discounted Cumulative Gain* (DCG) represents a penalized version of CG with a log component to decrease the score of $G(i)$ as the position of relevant files increase in the ranking. The farthest the buggy files are from the top positions of the rank, the higher is the discount. The Equation 13 show the recursive function for DCG.

$$DCG(i) = \begin{cases} CG(i), & \text{if } i < b \\ DCG(i-1) + G(i)/log_b\ i, & \text{otherwise.} \end{cases} \tag{13}$$

Finally, the *Normalized Discounted Cumulative Gain* ($NDCG$) applies a normalization to DCG, considering the DCG of an ideal ranking as reference ($IDCG$). The ideal ranking places the most relevant files in the top positions and gives the best possible performance for this set of files. The NDCG is given by the Equation 14. The truncated version of NDCG is frequently found and is indicated by $NDCG@k$, where k represents the maximum number of files considered in the rankings.

$$NDCG(i) = \frac{DCG(i)}{IDCG(i)} \tag{14}$$

For the BL problem, the binary labeling used in the previous measures is enough since a file can merely be considered buggy or non-buggy. On the other hand, NDCG is a good measure since it leverages two ideas: give a higher score to rankings whose buggy files near the top positions and also to discount rankings farthest from the ideal (when all buggy files are in the top positions).

2.3.7 Other metrics

Many other metrics apply in the context of BL, such as EXAM, Expense, AUC, AUCEC, Effort@k, MAE, and MFE. For this work, the more popular metrics described in previous sections are enough, and we can find more details about the other metrics elsewhere (ZHAO et al., 2015; RAY et al., 2016; CHAKRABORTY et al., 2018; RAHMAN et al., 2014; DIGIUSEPPE; JONES, 2015).

2.4 Machine Learning Approaches

Since this work proposes the use of Machine Learning in the BL problem, this section gives a very brief overview of ML techniques applied and discussed in the following chapters.

2.4.1 Learning to Rank

Methods using ML to solve ranking problems are generally called "learning-to-rank" methods or LtR (LIU, 2009). There are many algorithms to LtR usually classified as *Pointwise*, *Pairwise* and *Listwise*.

2.4.1.1 Pointwise

In Pointwise algorithms, the idea is to transform the ranking problem in a regression problem of each item (or document) in the rank. For example, in BL context, an item would be a source code file suspected to be buggy. We compute an individual score for each item to produce the ranking based on its features. We briefly describe some traditional Pointwise LtR algorithms next:

Random Forest (BREIMAN, 2001) is a bagging and inherently parallel algorithm based on CART (Classification and Regression Trees) to build decision trees.

MART (FRIEDMAN, 2001), Multiple Additive Regression Trees, is an ensemble model of boosted regression trees, producing a linear combination of the outputs of a set of regression trees.

2.4.1.2 Pairwise

In Pairwise algorithms, we reduce the problem to the classification of pairs of documents, and the algorithm should decide which document from the pair should be on the

top. While Pointwise algorithms are concerned with finding the degree of relevance of a document, Pairwise is more concerned about finding the proper relative order between the documents in a pair. Examples of Pairwise algorithms are:

RankNet (BURGES et al., 2005) is a probabilistic algorithm based on neural networks and applying cross-entropy, back-propagation, and gradient descent in training.

RankBoost (FREUND et al., 2003) is a boosting algorithm, training weak rankers in rounds and generating as output a linear combination of these weak rankers.

RankSVM (JOACHIMS, 2006) is a variant of an SVM (Support Vector Machine), adapted to the LtR problem, and targeting to optimize the number of correct classified pairs of documents.

2.4.1.3 Listwise

In Listwise algorithms, the learning process considers the whole list of documents as an instance for training (CAO et al., 2007). An instance is composed of 1) a query (a bug report, for BL problem), 2) a list of documents (the source code files to consider for that bug report), 3) a list of feature vectors associated to each document based on the query, and 3) a ground truth with the relevance of these documents to the queries (a mapping telling which file is buggy and which has not the bug described in the associated bug report). Examples of Listwise are:

AdaRank (XU; LI, 2007) is a boosting algorithm, inspired in AdaBoosting, adapted to the LtR ranking problem. AdaRank constructs weak learners as a boosting algorithm by re-weighting training data and forming an ensemble to boost the final performance.

Coordinate Ascent (METZLER; CROFT, 2006) is a multivariate objective optimization technique that optimizes each dimension (or feature) sequentially.

LambdaMART (WU et al., 2010) is the boosted tree version of LambdaRank (or a combination of MART and LambdaRank).

LambdaRank (BURGES; RAGNO; LE, 2006) is a gradient-based on NDCG as cost function and is smoothed by the RankNet loss.

ListNet (CAO et al., 2007) was the first Listwise proposed approach. Based on Neural Networks and Probability Models, that accounts for the possible permutations

between the documents in the rank and uses Gradient Descent as an optimization algorithm.

2.4.2 Language Models

Language Models are statistical models assigning the probability of occurrence to a sequence of words (or tokens) (TU; SU; DEVANBU, 2014). For example, given a sequence of tokens $S = t_1\ t_2\ ...\ t_N$, a language model allows computing the probability of occurrence of this sequence as a product of conditional probabilities for each token composing the sequence, as shown in Equation 15.

$$P(S) = P(t_1) * \prod_{i=2}^{N} P(t_i|t_1, ..., t_{i-1}) \quad (15)$$

It is not practical to compute the probability considering all the tokens in the sequence. Thus a *Markov assumption* is applied, and the calculus considers only the $n-1$ more recent tokens. This computing approach is the n-gram model, where n defines the number of tokens considered to form the sentence.

Since software programs have highly repetitive and predictable structures and contents, language models can detect unnatural code. We can use the cross-entropy metric to measure the level of naturalness of a piece of code. For example, the cross-entropy of a sequence S with probability $p_M(S)$ estimated by the language model M, is computed according to the Equation 16.

$$H_M(S) = -\frac{1}{N} log_2\ p_M(S) = -\frac{1}{N} \sum_{1}^{N} log_2 P(t_i|h) \quad (16)$$

With these ideas in mind, Ray (RAY et al., 2016) has shown that the pieces of code with high entropy relate to buggy code. When comparing a buggy piece of code with the associated fixed piece of code, the entropy tends to decrease.

2.5 Final Considerations

We provide in this chapter a very brief introduction to essential concepts related to the BL task, i.e., bug, bug reports, patches, and bug localization. We also introduce some of the common performance measures applied in BL evaluations (MAP, MRR, NDCG, and Top-N) and also in our experimentation. Between the many datasets applied in research, we focus our work on two of them: Defects4J and LR-dataset. Our experimentation is conducted and detailed in: Chapter 5 exploring some of the LtR algorithms

from Section 2.4.1, the introduction of new features based on language models briefly introduced in Section 2.4.2 (with some additional implementations details discussed in 4.7.1), and other strategies trying to improve the BL rankings (presented further in Chapter 4); and Chapter 7 experiment with a new approach to evaluate BL based on the characterization of bug datasets, especially through the analysis of its patches (also detailed in Chapter 6).

Chapter 3

Strategies for Bug Localization

Since the first efforts to automate the BL task, the research community has produced many approaches. While there are many perspectives to analyze these approaches, one accepted alternative usually found in the literature is to divide BL strategies according to the type of information they process, naming them static information-based, dynamic information-based, and hybrid or multi-modal approaches. Static information sources are available before the localization process starts and do not require re-execution. Examples of static information are source code, bug reports, official documentation, change history (commits), static metrics, and other kinds of related documentation (discussion forums, questions and answers sites, complementary documentation, and more). Usually, dynamic information requires the system re-execution to produce the input data. Examples of this kind of data are execution traces, stack traces, dynamic metrics, spectrum, coverage, and any information extracted from test case running. Finally, hybrid approaches combine these two kinds of information to point out the ranking of suspects. This chapter presents state of art with some proposals from the many available in these three lines.

3.1 Static Information-Based Approaches

This section presents some of the static-based approaches to BL, the main focus of this work, and the source for some experimental baselines. We started the construction of an experimental package to evaluate BL approaches with the reproduction of the LR approach (YE; BUNESCU; LIU, 2014; YE; BUNESCU; LIU, 2016), presented first in this chapter. We detail the experiments with the initial reproduction in Chapter 5. Another approach, BLUiR (SAHA; SAHA; PERRY, 2013), was also applied in some

experiments, especially in Chapter 7. The other works in this chapter illustrate different alternatives to deal with the problem of BL.

3.1.1 LR

A learning model for locating bugs is proposed in (YE; BUNESCU; LIU, 2016), combining a total of 19 features including previously applied features (SAHA et al., 2013; YE; BUNESCU; LIU, 2014), and also some new features (such as measures of code complexity, PageRank, and Hyperlink-Induced Topic Search (HITS)). Since the approach applies LtR algorithms, we call Ye, Bunescu e Liu (2016) approach as LR to avoid confusion to the broader LtR acronym. The sources of information used are source code, bug reports, Application Programming Interface (API) documentation, change history, and dependency graph between files. The applied models were:

VSM The vector representation model applies for the generation of ϕ_1 (Superficial Lexical Similarity) and ϕ_2 (API Enriched Lexical Similarity). While ϕ_1 represents a classic similarity comparison between terms in the bug report and the source code, ϕ_2 aggregates terms found in the API documentation of methods and classes used in the source code file. The feature score returns the maximum similarity to the bug report between the whole file and its methods.

Collaborative Filter The ϕ_3 feature refers to the similarity between the content of the target bug report and the summary of previous reports associated with each source file, based on the usual idea of Collaborative Filters in Recommender Systems.

Class Name Similarity ϕ_4 feature matches class names found in the summary of the target bug report and the source code files. Thus, the calculus considers the length of the matched class names.

File Review History The revision history (or changes) of files provide two features: ϕ_5, the recency of the change, proportional to the difference of months between the creation of the target report and the last change made to each file evaluated; ϕ_6, the frequency of changes, counting the number of changes the file had undergone before the report creation.

Structured Information Retrieval Eight features are derived based on the retrieval of structured information originally proposed in (SAHA et al., 2013). The essential idea is to extract class names, methods, variables, and comments from the source

code and compare them to the terms in the summary and description fields of the bug report. Each combination represents one of eight features ϕ_7 to ϕ_{14}.

Dependency Graph between Files Based on the extraction of the dependency graph between source code files, the last five features are defined, being: ϕ_{15} the number of dependencies of a file s to other files; ϕ_{16} the number of files that depend on s; ϕ_{17} the PageRank of each file; ϕ_{18} the Hub degree and ϕ_{19} the Authority degree of the file, according to the algorithm HITS.

The feature normalization process put values between 0 and 1, limiting their original values to the maximum and minimum found in the training data. One of the main objectives of the training is to learn the weights to perform the linear combination of features to define the ranking of the files for each bug report. Ye, Bunescu e Liu (2016) also conducted a study on the relevance of the features, and the main conclusions were: 1. through an automatic selection algorithm, it is possible to use a subset of features whose result only with the six most important characteristics represents more than 90% of that obtained with all the characteristics; 2. There are no irrelevant features, each variable's importance, depending on the target system; 3. In general, the features of greatest impact on the ranking were ϕ_1, ϕ_3 and ϕ_4, while the least important were ϕ_6, ϕ_{10}, ϕ_{15} and ϕ_{19}.

The authors created a dataset for evaluating the proposal with more than 22,000 bug reports in their previous work (YE; BUNESCU; LIU, 2014). The idea was to solve a series of reported problems, especially the bias caused by using single versions of a codebase and ignoring the timestamps from the bug reports creation. This situation can cause potential bias in the ranking results because of bug fixes (or patches) in the code from the report creation. The systems used to create this dataset were: AspectJ, BIRT, Eclipse Platform UI, JDT, SWT, and Tomcat. The main results obtained were Accuracy@k: for k = 1 from 13% to 42%, for k = 5 from 29% to 71%, for k = 10 from 39% to 80%; MAP from 0.16 to 0.49; MRR from 0.21 to 0.55. Tomcat gave the best results in all metrics, while the worst results came from BIRT. Additionally, a replication of the fixed-version dataset containing AspectJ, Eclipse, and SWT allowed comparing other works.

3.1.2 AmaLgam+

AmaLgam+ (WANG; LO, 2016) leverages five sources of information (version history, similar bug reports, structure, stack trace, and reporter information) to improve the

performance of the authors' previous work (former AmaLgam approach applies only three of these sources).

The authors applied the dataset in the assessment of BugLocator, and BLUiR approaches. The obtained results were: MAP from 0.36 to 0.62; MRR from 0.47 to 0.71; Hit@1 from 0.36 to 0.63; Hit@5 from 0.60 to 0.82; Hit@10 from 0.69 to 0.90.

3.1.3 DNNLoc

DNNLoc (LAM et al., 2017) combines the rVSM information retrieval model proposed in (YE; BUNESCU; LIU, 2014) with Deep Learning to solve the lexical mismatch problem between terms in the source code and terms in the bug report. The lexical mismatch is related to the lack of direct correspondence between the terms in the bug report and the terms in the source code and negatively impacts the BL strategies. To avoid the mismatch, strategies based on vector representations of terms apply since they go beyond textual similarity and consider, for example, common contexts of use of these terms. With the use of DL, DNNLoc is able to map a term such as *context* present in a bug report to related terms in the source code such as *authorization*, *ctx*, *envCtx* textitasyncContext.

DNNLoc separates the features used for the calculation of suspicious files in different vector spaces, being: 1) relevance of terms between the bug report and source code; 2) textual similarity between report and code; 3) collaborative filtering based on similar reports changing common source files; 4) nominal similarity between entities in the report and classes in the code; 5) recency of fixing; 6) fixing frequency. Despite the similarity with some features used in (YE; BUNESCU; LIU, 2014), the combination for calculating the ranking of suspicious files is done non-linearly through a DL network. Another DL model (*Auto-Encoders*) is used to reduce dimensionality, given the large number of features extracted from bug reports and source code.

The basis for extracting the characteristics comes from pairs containing a bug report and a source code file. For the training of the networks, positive examples (files that have received corrections pointed out by the report) and negative examples (sample of files textually similar to the first but not associated with the report of bug and correction) are selected. The processing of the bug reports terms is the usual (space separation, removal of *stop words*, division of compound words in the style *CamelCase* with the maintenance of the original terms as well, extraction of radicals *Porter Stemming*, relevance calculation based on *tf-idf*). Four features are extracted from the source code: 1. identifier names in the source code; 2. API elements used in the code (names of classes,

interfaces, and external methods); 3. comments in the source code; 4. comments/descriptions associated with API elements. Comments on the source code are handled similarly to the terms in the bug report.

The systems used in the experiments are the same as in (YE; BUNESCU; LIU, 2014), available online [1]: AspectJ, BIRT, Eclipse UI, JDT, SWT, and Tomcat. Even after contact, the authors did not provide experimental data or scripts for reproduction. Based on the article alone, it would be difficult to obtain a precise reproduction since much important information about the architecture used for implementation is implied or omitted (mainly related to DL models) and would have to be discovered by trial and error (e.g., number of nodes used, cell types, number of layers, number of entries, and others). The main results obtained with DNNLoc were: Top-1 from 25.2 to 53.9%, Top-5 from 42.2 to 72.9%, Top-10 from 50.9 to 85.0%, MAP from 0.2 to 0.52, MRR from 0.28 to 0.60.

3.1.4 ConCodeSe

The main focus of ConCodeSe (**Con**textual **Code** **Se**arch) (DILSHENER, 2016) is the search for suspect source file names at specific points in a bug report. Unlike other proposals, ConCodeSe does not use historical information and search for similarities between bug reports. The information is extracted only from the buggy version of the source code and the bug report in question. In addition, the treatment is different from each field in the report, considering the distinct nature of its content (more technical, containing information such as execution stack/calls and code elements, or less technical, containing the vocabulary of the application domain itself). Suspicious files are ranked based on two scoring models: 1. Probabilistic; 2. Lexical Similarity.

Lucene resources [2] and the vector model (VSM) applies in the probabilistic model. The base queries for generating probabilistic scores use different combinations of terms (complete and *stemmed*) extracted from three sources: bug report, comments in the code, and the source code file itself.

The semantic similarity model applies three criteria to rank a file: 1. the key position of the file name in the bug report (KP score); 2. position of the file name in the stack trace (ST score); 3. Matching terms in the report with terms in the file (TT score). The rankings definitions are:

[1] DNNLoc Datasets: <http://dx.doi.org/10.6084/m9.figshare.951967>
[2] Apache Lucene: <https://lucene.apache.org/core/>

KP score scores files whose name occurs on the first (10 pts), second (8 pts), penultimate (6 pts), or last position of the bug summary (4 pts).

ST score scores files whose name occurs on the first (9 pts), second (7 pts), third (5 pts), and fourth (3 pts) stack trace position.

TT score applies punctuation algorithm for a) exact match (+2 pts) or b) partial match (+0.025 pts) between terms in the report and file name; c) exact match between terms in the report and terms in the file (+0.0125 pts).

Unlike other approaches that combine the scores obtained in each model/calculation with weight adjustment, ConCodeSe assumes only the best result for ranking the suspicious file.

The study was done on open systems: ArgoUML, AspectJ, Eclipse, SWT, Tomcat and ZXing. The executable of ConCodeSe [3] and references to datasets [4] are available online, although the authors have not made available a set of pre-processed data. The main results were: MAP from 0.30 to 0.68, MRR from 0.55 to 0.94, Top-1 from 31.9% to 72.4%, Top-5 from 61.2% to 89.8 %, Top-10 from 65.9% to 92.9%.

3.1.5 NSGA-II

Almhana et al. (2016) proposes the first approach to locating bugs based on a genetic algorithm of multi-objective optimization. This work aims to transform the problem of locating bugs into a search problem, with the following objectives: 1. Maximize the lexical similarity of bug reports with the source code and its method APIs; 2. Maximize the historical similarity between bug reports and classes in the source code, based on the number of bug fixes received by the class, recency of fixes/changes, and consistency in which classes are changed together in previous patches; 3. Minimize the number of suspicious recommended classes, aiming to reduce the developer's effort while proceeding with the final localization and repair.

The evaluation of the proposal is made on the systems: AspectJ, BIRT, Eclipse UI, JDT, SWT and Tomcat. There is no mention in the work on making the implementation available online. The dataset is based on (YE; BUNESCU; LIU, 2014). The main results were (means of the experiments): Precision@k of 89% (k = 5) and 82% (k = 10); Recall@k of 72% (k = 5) and 81% (k = 10); Accuracy@K of 68% (k = 5) and 86% (k = 10).

[3] ConCodeSe Tool: <http://www.concodese.com/?cat=7>
[4] Experimental data ConCodeSe: <http://www.concodese.com/?cat=9>

3.1.6 Locus

Locus (LOcates bugs from software Change hUnkS) is proposed in (WEN; WU; CHEUNG, 2016) aiming to rank two levels of suspicious components, i.e., files and changes, using the hunks (or continuous sequence of lines changed) found in the project commits from change histories. Beyond the reduced granularity compared to files, the idea behind the ranking of changes is to provide a better context to facilitate the developer's work while dealing with bug localization and fixing tasks. Three models are used to produce the rankings: 1. based on a Natural Language (NT) corpus, that computes the similarity between terms in hunks with terms in bug reports; 2. based on Code Entity (CE) corpus, that computes the similarity between code entity names in the hunks and those found in bug reports; 3. The Boosting model measures elapsed time between the bug report creation and the commits time. The information is extracted from change histories (commits) and bug reports to build these ranking models. The final score comes from a weighted sum from the scores of each model. These weights are defined experimentally.

The evaluation is proposed on: AspectJ, JDT, SWT, PDE, Tomcat and ZXing. Locus source code repository[5] and dataset[6] are available online. Obtained results are Top-1 from 25% to 64%, Top-5 from 56.6% to 84.7%, Top-10 from 63.9% to 91.8%, MAP from 0.32 to 0.64, and MRR from 0.381 to 0.725.

3.1.7 BLIA

BLIA, **B**ug **L**ocation with **I**ntegrated **A**nalysis (YOUM et al., 2015), combines four score models in a weighted/parametrized way to produce a final suspect score for the target source code files. The information comes from source code files, bug reports, and change history.

The first score model, SimiBugScore, is based on the similarity of the queried bug report to previous fixed bug reports in the project and relies on the BugLocator approach in (ZHOU et al., 2012) with classical Vector Space Model (VSM) similarity. The second score model, StructVSMScore, is based on the VSM similarity between the bug report fields (summary and description) to the source code structured information (identifiers for classes, methods, and variables, plus source code comments), and also relies on BugLocator extended with the BLUiR approach in (SAHA et al., 2013). The third score model, STraceScore, is based on the scoring of files appearing in the stack trace

[5] Locus repository: <https://github.com/justinwm/Locus/>
[6] Locus dataset: <http://home.cse.ust.hk/~mwenaa/Locus.html>

informed in the queried bug report, including files with names explicit in the stack trace and the imported files by the formerly found files. STraceScore relies on the BRTracer approach in (WONG et al., 2014a) for the processing and scoring of the stack trace information. The last base score model, CommScore, is based on the extraction of the last commits related to bug fixing and previous to the queried bug report. The computed score correlates to the recency of the fixing pointed by the selected commits. CommScore relies on the AmaLgam approach in (WANG; LO, 2014). The final BLIA score integrates all these scores in a formula parametrized by α and β, balancing the influence of each base score, and k, which defines the recency range in days to consider in CommScore.

The evaluation was done over AspectJ, SWT, and ZXing from (ZHOU et al., 2012) and (WONG et al., 2014a). The results were: Top-1 from 37.7 to 68.4, Top-5 from 60 to 82.7, Top-10 from 73.2 to 89.8, MAP from 0.323 to 0.506, and MRR from 0.491 to 0.746.

3.1.8 BLUiR

BLUiR is proposed in (SAHA et al., 2013), based on Indri IR open-source toolkit and in a structured information retrieval approach. The main idea behind BLUiR is to leverage a classical TF-IDF model by distinguishing the bug report fields (query) and the source code fields (document and collection corpus) to proceed with the bug localization (information retrieval task). The fields summary and description from the bug report fields are the query side information targets. BLUiR extracts the fields (classes, methods, variables, and comments) to compose the documents corpus from the source code. Each query versus document field generates a document score, and adding the individual scores produces a final score.

The approach is evaluated with the same dataset from the BugLocator approach (ZHOU et al., 2012), described in the following subsection. The obtained results were: MAP from 0.17 to 0.56; MRR 0.33 to 0.65. The authors also present results for Top-1, Top-5, and Top-10 metrics, but as the absolute number of localized bugs. We do not present the results for these Top-N metrics to avoid confusion. Finally, the BLUiR authors claim to outperform BugLocator, and BugScout (NGUYEN et al., 2011), while the last comparison is indirect and based on the relative results obtained by each approach on different evaluation datasets.

3.1.9 BugLocator

Proposed in (ZHOU et al., 2012), BugLocator was the first to use similar bug reports and source code file size in the suspicious score computing, beyond the classical similarity comparison of the queried bug report to the target source code files. The information sources are the bug reports and the source code file.

Two score models are applied. The rVSM score model is an adaptation of the classical VSM model, with a weight component defined by a formula based on the source code file size ($\#terms$) because the authors assume the big files as more buggy-prone. The SimiRank score model has two steps: first, to find the previous bug reports similar to the queried bug report and compute the similarity of the queried bug report to the fixed files associated with similar bug reports. A weight factor α defines the balance between the contributions of each score in the final weight sum of rVSM and SimRank scores.

The evaluation is done over ZXing, SWT, AspectJ and Eclipse. BugLocator is compared to other BL approaches based on: 1. the classical VSM model; 2. Latent Dirichlet Allocation (LDA); 3. Smoothed Unigram Model (SUM); 4. Latent Semantic Index (LSI). The obtained results were: Top-1 from 22.3 to 40.0; Top-5 from 40.91 to 65.31; Top-10 from 55.59 to 77.55, MRR from 0.33 to 0.48; MAP from 0.17 to 0.41.

3.2 Dynamic Information-Based Approaches

Dynamic approaches to BL represent another path to solve the localization problem. However, unlike static approaches, the input data requires the execution of the target system with some instrumentation to collect execution traces. This section illustrates a few works in this line.

3.2.1 Tarantula

Tarantula (JONES; HARROLD, 2005) is one of the frequently cited Spectrum-Based Fault Localization (SBFL) approaches. The suspiciousness level for a component c computes as in Formula 17. Tarantula overcomes other approaches such as Set-Union, Set-Intersection, Cause-Transition, and Nearest-Neighborhood techniques. Tarantula's authors use the Siemens package (consisting of C programs with few lines of code) and analyze 122 bugs. They report that in 55.7% of bugs, less than 10% of the code requires examination to localize bugs, while others require 10% to 90% code analysis. They also point out other experiments performed with a larger program (Space, 6,218 lines),

indicating that in 40% of the bugs, it would be necessary to examine less than 1% of the code for LB. Even with better results indicative, remember that the increase in program size implies examining more lines of code when considering percentage data. Therefore, even if the values seem better in percentage terms, the effort can be greater. Finally, the approach is also limited to programs containing a single fault in each version (one bug at a time).

$$Susp(c)_{Tarantula} = \frac{\frac{n_f}{n_{tf}}}{\frac{n_s}{n_{tf}} + \frac{n_f}{n_{tf}}} \quad (17)$$

where,
n_f = number of failed test cases covering c;
n_{tf} = total number of failed test cases;
n_s = number of successful test cases covering c;
n_{ts} = total number of successful test cases.

3.2.2 D*

D* overcomes other SBFL approaches in an analysis involving 24 programs and 38 techniques for locating bugs (WONG et al., 2014b). Formula 18 shows the metric proposed in D*, which is a variation of Kulczynski's formula (CHOI; CHA, 2010). The coefficient * allows for varying the weight of the coverage of components by failed test cases. When we have * = 1, the formula reduces to Kulczynski. The performance of D* improves with greater values for *, reaching a threshold. D* uses only coverage information, not depending on prior information on the program's structure and semantics. While testing more extensive programs like Ant (75 KLOC), the question remains whether the technique can apply to even larger programs.

$$Susp(c)_{D*} = \frac{(n_f)*}{n_{uf} + n_s} \quad (18)$$

where,
n_f = number of failed test cases covering c;
n_{uf} = number of failed test cases uncovering c;
n_s = number of successful test cases covering c;
* = D* coeficient, greater than or equal 1.

3.3 Hybrid approaches

Hybrid approaches combine the static and dynamic approaches illustrated in the last sections. Hybrid approaches are recent since most previous alternatives avoid mixing

static and dynamic strategies. One reason is the difficulty obtaining both kinds of data input, generating an extra overhead. Another reason is the difficulty validating the approach since it is not usual to have bug datasets that contemplate static and dynamic info needs with shelf ground truths. This section illustrates some of the few works published using this strategy.

3.3.1 EnSpec

EnSpec (CHAKRABORTY et al., 2018) combine static Language Models (LM) with SBFL dynamic approaches to produce the ranking of suspicious lines of code. Two models apply: the LM model from (TU; SU; DEVANBU, 2014) is used to compute the entropy of lines in source code ($gram model with syntax-sensitive normalization (RAY et al., 2016)), considering the likelihood of tokens sequences forward, backward, and the average; 25 metrics from previous SBFL models (e.g., Tarantula, Ochiai, and others) applies to compute suspiciousness of executed lines for test cases. The entropy measures apply to learn weights that relate buggy/non-buggy lines in the source code to failed/passed test cases. An Ensemble Learning technique applies to compute the final suspiciousness. RankBoost and Random Forest are the learning-to-rank algorithms used.

The evaluation occurs with the projects of the Defects4J and ManyBugs benchmarks. There is no mention on the paper to make the system and post-processed dataset available online since it is an ongoing work pre-printed in Arxiv. The obtained values for $AUCEC_{100}$ metric goes from 0.864 to 0.961 and represents gains of 0.708% to 19.13% compared to SBFL approaches alone. The experiments show that the average entropy feature has a major role in the improvement of the bug localization, especially to differentiate buggy lines from non-buggy lines executed by fail test cases and also in a cross-project setting (where the learning derived from a project is used to locate bugs in a different one).

3.3.2 AML

AML (LE; OENTARYO; LO, 2015) proposes the combination of IR with SBFL. The main idea of the proposal (classified as multimodal by the authors) is to combine information extracted from the bug reports (used in IR) and coverage of test cases (used in the SBFL). According to the authors, AML was one of the first multimodal proposals for BL in the literature. Previously there were only a few proposals for the

different domains of feature localization. In addition, it is the first approach to use suspicious words associated with bugs and able to adapt individually to each type of bug (achieved through ML and a probabilistic-based optimization process). The experiments considered 157 real bugs from AspectJ, Ant, Lucene, and Rhino applications. The runtime for the bug localization was between 20 and 80 seconds approximately, which makes it possible to apply the technique interleaved with the developer activities (the experiments run on an Intel (R) Xeon Linux Server E5-2667 2.9GHz). The results showed that the proposal surpasses the other approaches considered in state of the art (based on IR, SBFL and multimodal adapted from feature localization context to BL), reaching 92 of the 156 bugs (improvement of 27.78% and 47.62% for Top-N metric) and MAP of 23.7% (surpasses in 28.8% the other approaches). However, even with the good results compared to the other techniques, it is notable that there is still a considerable margin for improvements in the accuracy.

3.4 Reference and chronology of BL approaches

Figure 9 presents a chronology with some of the more recent static approaches, including those summarized in previous sections. The arrows show the major references between approaches and illustrate their comparisons and inter-connections over the years. Since we choose BLUiR (SAHA et al., 2013) and LR (YE; BUNESCU; LIU, 2014; YE; BUNESCU; LIU, 2016) approaches as our baselines for the experimental package and experiments, we highlighted it with a white background. Figure 10 and Figure 11 show the equivalent information for dynamic and hybrid-based approaches, respectively. Table 1 to 3 indexes the references for the presented approaches in the figures. Even considering we present a reasonable number of approaches, we did not proceed with a systematic review, and, certainly, some works were left out. For a more in-depth and complete survey in BL, the work of (WONG et al., 2016) is an excellent reference to papers until 2016.

3.5 Final considerations

This chapter presented some of the published approaches for BL. We can observe that this is an active research area, and most of the works concentrate on static and dynamic data-source approaches. Static approaches start from previously available data to provide rankings of suspecting buggy components, mainly targeting the mapping of bug re-

Year	Approach / Reference
.. 2012	BugLocator (ZHOU et al., 2012), BugScout (NGUYEN et al., 2011), Sinha et al. (SINHA; MANI; MUKHERJEE, 2012), Sisman et al (SISMAN; KAK, 2012)
2013	BLUiR (SAHA et al., 2013), NB-TwoPhase (KIM et al., 2013)
2014	AmaLgam (WANG; LO, 2014), BRTracer (WONG et al., 2014a), $LR_{(1)}$ (YE; BUNESCU; LIU, 2014), Lobster (MORENO et al., 2014)
2015	BLIA (YOUM et al., 2015), BugWalker (WANG; PARNIN; ORSO, 2015), HyLoc (LAM et al., 2015), PartOfSpeech (TIAN; LO, 2015)
2016	ConCodeSe (DILSHENER; WERMELINGER; YU, 2016), DrewBL (UNENO; MIZUNO; CHOI, 2016), Locus (WEN; WU; CHEUNG, 2016), $LR_{(2)}$ (YE; BUNESCU; LIU, 2016), LR-WE (YE et al., 2016), NP-CNN (HUO; LI; ZHOU, 2016), NSGA-II (ALMHANA et al., 2016)
2017	AmaLgam+ (WANG et al., 2016), DeepLocator (XIAO et al., 2017), DNNLoc (LAM et al., 2017), LS-CNN (HUO; LI, 2017)
2018	Bench4BL (LEE et al., 2018), Blizzard (RAHMAN; ROY, 2018), CNN_Forest (XIAO et al., 2018), EBRo (ARCEGA; FONT; CETINA, 2018), Loyola et al. (LOYOLA; GAJANANAN; SATOH, 2018), Orca (BHAGWAN et al., 2018), Rath&Mader (RATH; MÄDER, 2018), TraceScore (RATH; LO; MÄDER, 2018)
2019	BLiM2 (ARCEGA et al., 2019), CAST (LIANG et al., 2019), Chaparro et al. (CHAPARRO; FLOREZ; MARCUS, 2019), DeepLoc (XIAO et al., 2019), D&C (KOYUNCU et al., 2019), Kim & Lee (KIM; LEE, 2019), Polisetty et al. (POLISETTY; MIRANSKYY; BA\CSAR, 2019), Zhang et al. (ZHANG et al., 2019a)
2020	Ackbar & Kak (AKBAR; KAK, 2020), BugPecker (CAO et al., 2020), DependLoc (YUAN et al., 2020), Khatiwada et al. (KHATIWADA; TUSHEV; MAHMOUD, 2020), KGBugLocator (ZHANG et al., 2020), Scaffle (PRADEL et al., 2020), Yang et al. (YANG; MIN; LEE, 2020)
2021	Arcega et al. (ARCEGA et al., 2021), DreamLoc (QI et al., 2022), IncBL (YANG et al., 2021) TRANP-CNN (HUO et al., 2019)

Table 1 – Summary of Static approaches for BL.

Year	Approach / Reference
.. 2012	Ample (DALLMEIER; LINDIG; ZELLER, 2005), Barinel (JANSSEM; ABREU; GEMUND, 2009), Carrot (PYTLIK et al., 2003), GZoltar (CAMPOS et al., 2012), Jaccard (ARTZI et al., 2012), Kulczynski (ABREU; ZOETEWEIJ; GEMUND, 2007), Ochiai (ABREU et al., 2009), Op2 (NAISH; LEE; RAMAMOHANARAO, 2011), SBI (ASKARUNISA; MANJU; BABU, 2011), Tarantula (JONES; HARROLD, 2005),
2013	Xie et al. (SHI et al., 2013)
2014	Dstar (WONG et al., 2014b), MULTRIC (XUAN; MONPERRUS, 2014), MUSE (MOON et al., 2014)
2015	Metallaxis (PAPADAKIS; TRAON, 2015)
2016	Savant (LE et al., 2016), Zheng et al. (ZHENG et al., 2016)
2017	FLUCCS (SOHN; YOO, 2017), Pearson et al. (PEARSON et al., 2017), PRFL (ZHANG et al., 2017), TraPT (LI; ZHANG, 2017)
2018	Delta Debug (CHRISTI et al., 2018), Wang et al. (WANG et al., 2018)
2019	CNN-FL (ZHANG et al., 2019b), DeepFL (LI et al., 2019), Raselimo & Fischer (RASELIMO; FISCHER, 2019)
2020	Deuslirio et al. (SILVA-JUNIOR et al., 2020), Kuma et al. (KUMA et al., 2020), ProFL(a) (LOU et al., 2020), ProFL(b) (THOMPSON; SULLIVAN, 2020)
2021	Alloy (KHAN; SULLIVAN; WANG, 2021), DeepRL4FL (LI; WANG; NGUYEN, 2021), GRACE (LOU et al., 2021), Sohn et al. (SOHN et al., 2021)

Table 2 – Summary of Dynamic approaches for BL.

Year	Approach / Reference
2015	AML (LE; OENTARYO; LO, 2015)
2017	Dao et al. (DAO; ZHANG; MENG, 2017)
2018..2021	EnSpec (CHAKRABORTY et al., 2018), NetML (HOANG et al., 2018)

Table 3 – Summary of Hybrid approaches for BL.

ports (typically, describing the faulty software behavior) to source code files that are more likely to contain the bug (or receive the fixing patch). In this context, the works usually employ different IR and ML techniques, like VSM (e.g., BugLocator and BLUiR) and LtR–based algorithms (e.g., LR), but with a growing interest in the application of Deep Neural Networks (DNN) models in recent works (e.g., NP-CNN, DeepLocator, DNNLoc, LS-CNN, CNN_Forest, DeepLoc, TRANP-CNN). Dynamic approaches require a system re-execution, generally guided by test-cases reproducing scenarios exhibiting the faulty behavior and producing trace data (or spectrum) containing information about all the source code components participating in the execution. Researchers propose formulas (e.g., Tarantula and D*) for ranking the components according to buggy likelihood. Since we have an instrumented execution, a more fine granularity (method or line level) to point out buggy components is commonly possible with SBFL approaches. Even with the common target of producing a good ranking of buggy suspects, the way the data is acquired and processed differs between static and dynamic approaches, making the merging a challenge. The merging difficulty and the mismatch of the approaches are possible reasons for the few available Hybrid approaches that combine both strategies and data sources. We can also observe that the precision and reliability of the produced rankings are still far from the ideal, and the research should advance so that automatic BL would become commonplace in a professional environment.

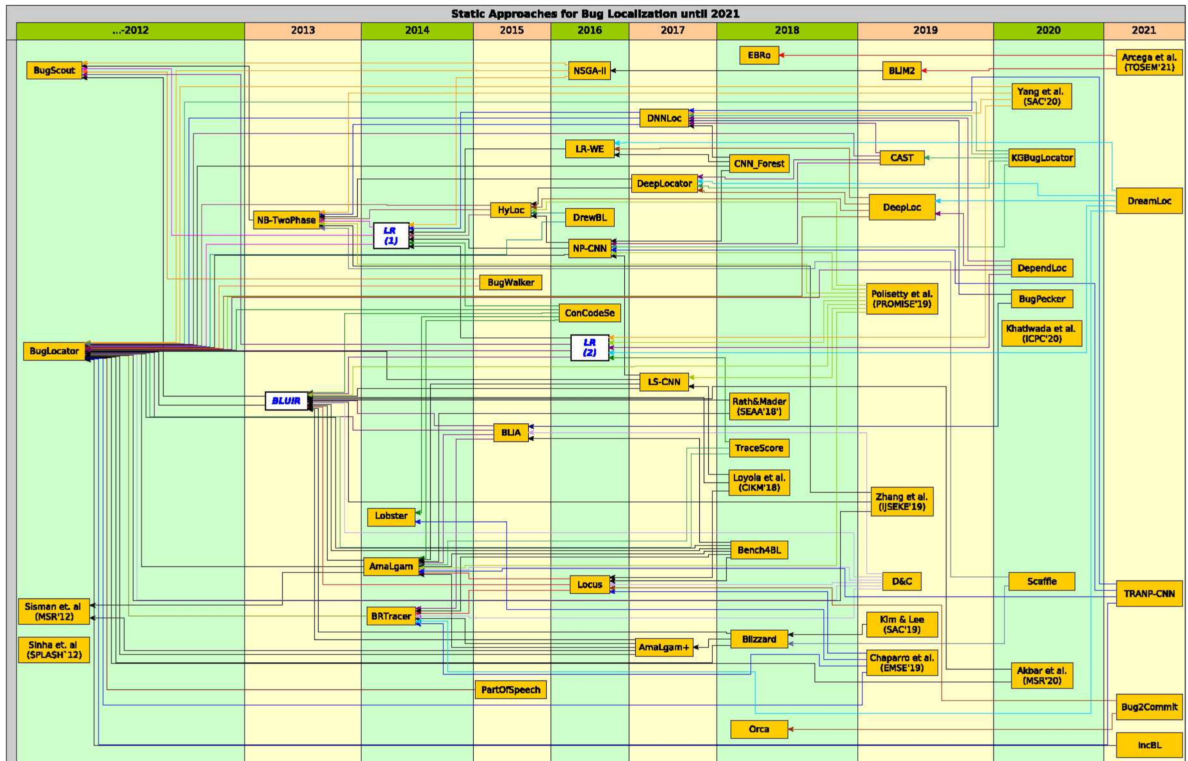


Figure 9 – Static approaches for BL until 2021.

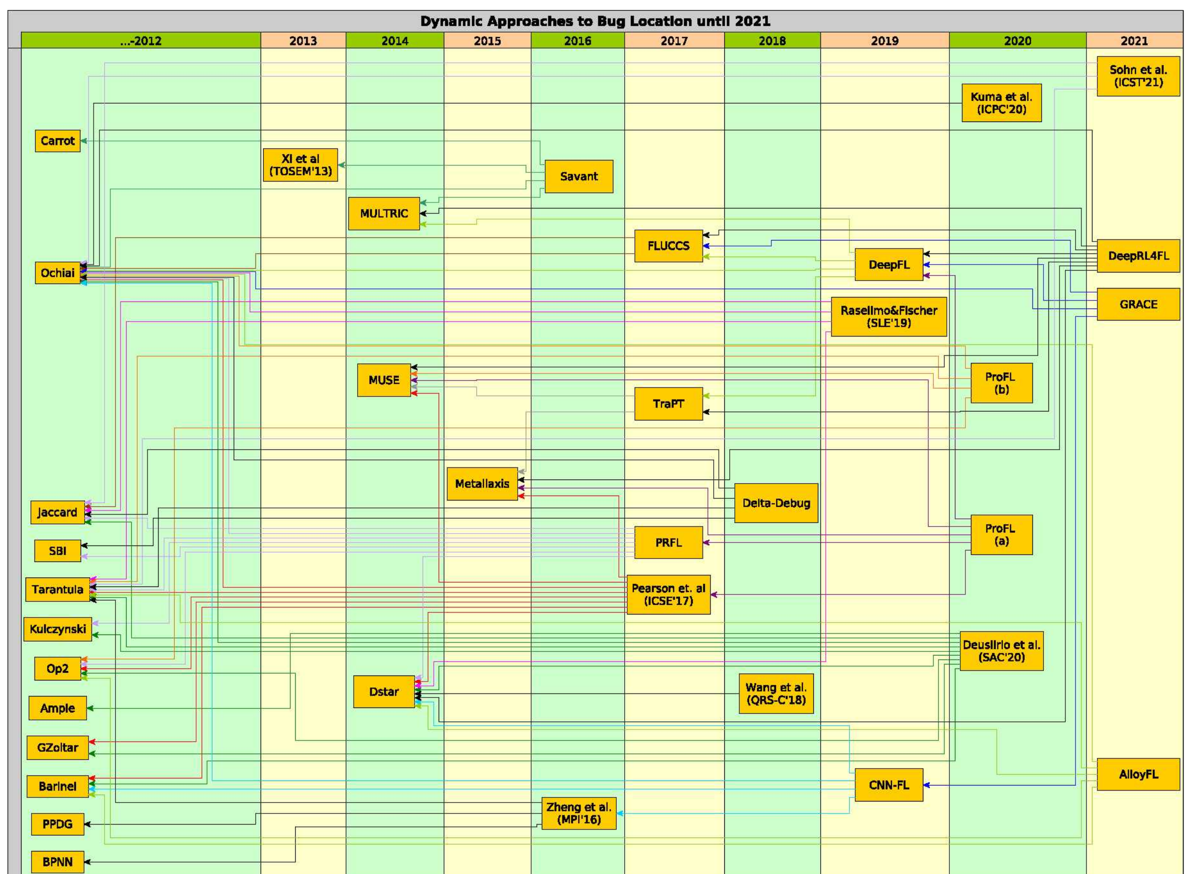


Figure 10 – Dynamic approaches for BL until 2021.

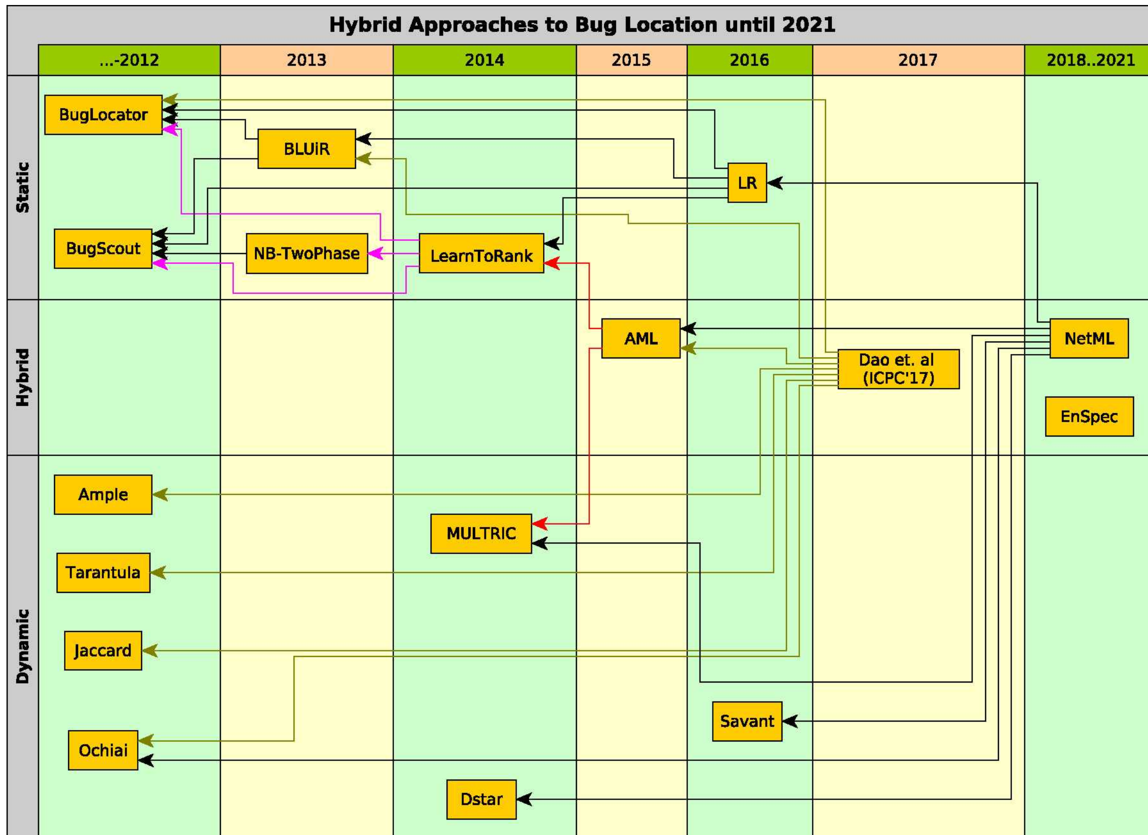


Figure 11 – Hybrid approaches for BL until 2021.

Chapter 4

On the Influential Factors for Bug Localization Exploratory Assessment

This chapter raises several of the many influential factors on the assessment of BL approaches, including the impact on performance measures. First, we enumerate and describe the factors. Then, we briefly present the experimentation package developed and applied in subsequent chapters while exploring and comparing different approaches for BL.

4.1 Bug Reports' Pre-processing

Studies on bug reports show the importance of adequate pre-processing before their use as input in BL techniques. Thus, the report quality measure should consider the existence (or absence) of critical information, relevance, and content correctness. For example, some bug reports may not refer to a bug but to users' questions or claims for requirements (BETTENBURG et al., 2008). These cases clearly would imply noise, especially in a ML approach. Therefore, a filter or weighting module for the bug reports is essential to compute how confident we should be about results based on a given report. Another study (MILLS et al., 2018) presents evidence that the bug report content has enough to improve the text retrieval based BL approaches, even without localization hints (i.e., specific information about code elements that would leverage the performance and that is not present in all bug reports). Finally, Chaparro, Florez e Marcus (2019) work points to query rewriting techniques as the next agenda for BL research. It provided a public curated dataset containing near-optimal queries generated from the bug report *title* and *description* fields through a Genetic Algorithmic approach

and applied in their experiments.

The LR study (YE; BUNESCU; LIU, 2016) that we based on has two types of features: 1. query dependent, i.e., features using the information of the bug reports to compute the ranking (e.g., all features based on similarity to the bug report summary or description); 2. query independent, i.e., features relying on other sources, without using information from the bug report as an input query (e.g., features based on PageRank or HITS algorithms, extracting information only from source code). Thus, these pre-processing approaches should only impact rankings generated from query-dependent features.

Next, we enumerate some strategies to deal with bug report pre-processing:

1. Discard bug reports that do not contribute to similarity-based features or has a null score in respective rankings, especially in rankings interfering directly in the generation of inputs to the ML process;

2. Assign a weight to each bug report according to their quality assessment, so the ranking generation focuses on bug reports with relevant information and reduces the impact of bug reports contributing little or nothing.

3. Discard bug reports related to patches over source code containing test suites or test cases exclusively. Since test suites are artifacts used to maintain code quality and have no direct influence on functionalities, there is no reason to treat a test case as a bug source like a source code containing functionalities.

4.2 Dataset Quality Assessment and Source Code Filtering

When running BL experiments, the applied dataset can influence the obtained results because of many variables like dataset size, type, and corpus characteristics, including the bugs collected, bug reports, target projects or systems, source code base, development period, and many others.

Focusing on just one aspect, as the mixing of functional and testing code, we can observe situations requiring attention and care. In LR–dataset (YE; BUNESCU; LIU, 2014), specifically in AspectJ project input data, test suites are indistinctly enumerated together with functional source code as part of the fixing patches for many bug reports. For example, the file Ajc150Tests.java (in folder /tests/src/org/aspectj/systemtest/ajc150) has 153 references in the XML input file containing all the AspectJ bug

reports data. These references are in the fixed file list, the ranking results enumeration, and the bug reports content fields (Summary or Description). For many bug reports, the rank of the non-functional file is above the position of the functional source code files (usually and ideally, the real target to fix the reported functional bug). More specifically:

1. Bug report 321 (BugId: 115235): Ajc150Tests.java is the first fixed file found ranked in the fourth position by LR approach; the other two fixed files are functional files ranked in fifth and ninth position, respectively;

2. Bug report 323 (BugId: 112756): only Ajc150Tests.java is enumerated as the fixing target and is in the ranking first position;

3. Bug report 325 (BugId: 114005): Ajc150Tests.java is the first fixed file, ranked in 2nd position; the other is a functional file ranked in 60th position;

4. Bug report 326 (BugId: 90143): Ajc150Tests.java is the first fixed file and is in the first ranking position; the other patch is on a functional file in the 124th ranking position.

With just these examples, it is clear that the results obtained for the performance of a tool based on the raw LR–dataset are not very trustful (without filtering testing files) or at least require additional considerations. Test suites like Ajc150Test.java contain calls for test cases, and even considering their change as a side effect of a patch, rarely test cases may be directly related to the cause of a bug. The most surprising is that this kind of file can be ranked in top positions while the actual functional source code files patched are poorly ranked (as in bug report 326). This situation is entirely misleading and invalidates someway the obtained results of LR experiments, and it is worthy to note that Kim e Lee (2018) already reported about this issue.

An experimentation package would require filtering options to exclude folders containing test cases or test suites and exclude bug reports patched exclusively by testing files to avoid the previous situation. These also can be considered a kind of ground truth quality assessment for the target dataset.

4.3 Bug Classification Schemes

The classification of bug datasets according to the bug characteristics is not so usual, and it is not explicitly available in bug datasets or benchmarks. Moreover, few studies proposed the classification of bugs or patches as a complement to improve the analysis

while using these datasets as proposed by Sobreira et al. (2018), Nayrolles e Hamou-lhadj (2018). Consequently, most studies and approaches consider the dataset like a black box, and a more in-depth performance analysis becomes extremely difficult, mainly because the nature and characteristics of each bug in the dataset can variate a lot. Another consequence is that these studies concentrate on showing numerical improvements obtained over past approaches but not explaining why and where these improvements happen and when the approach fails, according to the bugs' nature.

We start to fill the gap regarding the need to classify and discriminate the bugs' nature in a dataset with the first dissection study (SOBREIRA et al., 2018). After a broad review of the patches (or bug fixes) in the Defects4J dataset, it was possible to determine a series of characteristics previously not explicitly available about their bugs. We have concentrated on four main dimensions of the bug patches: the size of a patch (in lines of code); the spreading of the patches (again in lines of code and measuring the quantity and distance between chunks of code in each patch); the repair actions (the type of changes applied in a patch, in terms of syntactic constructions such as if conditionals, assignments, and loops); and finally, the repair patterns (more abstract structures or shared constructions repeatedly found in various patches). This first work was essential to provide insights into the bug patches' nature and how to automate the extraction of these characteristics. So, we have continued the work and proposed Automatic Diff Dissection (ADD)[1], a tool to support the automated extraction of bug patches' features defined in our Defects4J dissection study. Madeiral et al. (2018) details the Patch Pattern Detector (PPD), a module of ADD for detection of repair patterns in patches.

Figure 12 shows the distribution of the number of lines composing each patch (or code fixings) in Defects4J projects. According to the silhouette of the distribution, there are no considerable differences in the patches size between the projects. Only the Joda Time project diverges a little bit more in the size distribution, but they all have a high concentration of patches involving up to 9 lines, and 95% of the bugs have patches involving no more than 22 lines (SOBREIRA et al., 2018). This data would suggest a careful approach while training for BL, since most lines of a file are not affected in practice by a patch, and many approaches take the whole file given equal importance for all its lines and possibly feeding noisy data to the learning algorithm.

Figure 13 shows the distribution of chunks (or sequential block of code lines) that compose the patches in Defects4J. Again, the distribution is similar between the projects

[1] ADD repository: <https://github.com/lascam-UFU/automatic-diff-dissection>

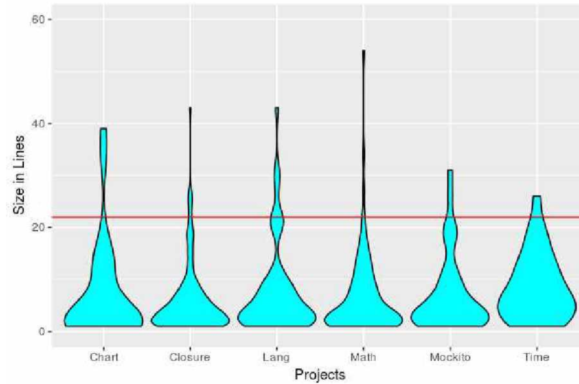


Figure 12 – Distribution of the number of lines in each patch of Defects4J projects.

(the Joda Time project is the one that diverges a little more still). The chunks' number is the first measure of patch spreading in the source code file (or between files). 25% of patches have only one chunk (i.e., no spreading). The majority (75%) have at most three chunks. Almost all the patches (95%) have no more than eight chunks. Complementing the analysis of spreading, Figure 14 shows the spreading distribution of patches. This spreading measure represents the number of lines between the chunks composing a patch. Many patches have no spreading at all (at least 25%), half have no more than just one line separating the chunks, while almost all (95%) have no more than 19 lines. These data suggest that even considering some separation between the code lines of a patch, these lines are close to each other, on average. Finally, considering the distribution of patches between Files (or Classes that have similar results) and Methods, most patches from Defects4J projects are restricted to a few Files/Classes (90% just one, and 95% at most 2), and Methods (90% at most two methods and 95% at most three methods). These highlights suggest that if we get Defects4J as a benchmark for BL, approaches and tools may be successful if they can handle bugs that require small patches and with a low spreading on the source code. By the way, it is essential to define how representative is Defects4J when compared to other projects of interest so that this kind of analysis and insights would also be applicable for other projects.

Beyond the basic dimensions of a patch, the insights related to repair actions (basic operations with syntactic constructs like conditionals, assignments, and others) and repair patterns (more abstract and repetitive constructs, like the adding of conditional blocks, missing of a null check, and others) can also be interesting for BL. Figure 15 shows the incidence of repair actions on the patches of Defects4J. In green are shown the actions involving adding code, in yellow are the actions that modify an existing code, and in red are the actions that remove existing code. While expecting that most of the

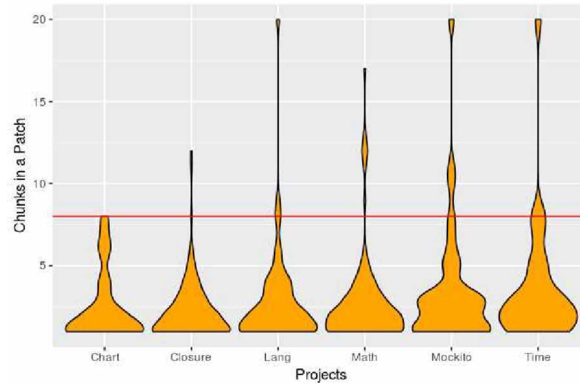


Figure 13 – Distribution of the chunks composing each patch of Defects4J projects.

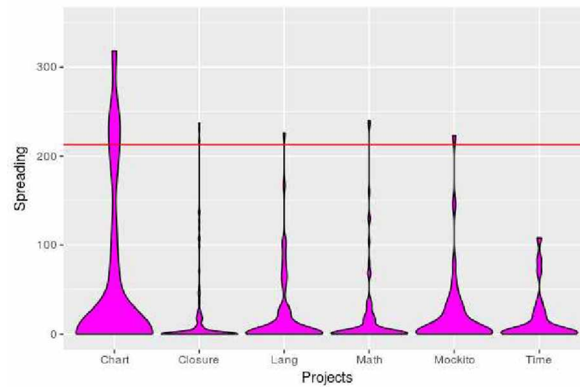


Figure 14 – Spreading distribution on each patch of Defects4J projects.

approaches for BL should be more successful in the handling of buggy code that should be modified or removed, it is not clear how these approaches would behave trying to find a bug that requires the adding of new code to fix it. It is evident in the chart that most of the actions in patches involve the adding of new code, e.g., the top-3 repair actions found in patches are adding of method calls (mcA), adding of conditionals (cndA), and adding of assignments (asgnA). Thus, the capacity to find bugs because of the lack of code is an authentic concern.

Figure 16 was extracted from our website "Defects4J Dissection"[2] and shows how many repair patterns are generally found in the patches of Defects4J. Most patches have between one and four repair patterns. A segmentation of the evaluated dataset according to the repair patterns found in the patches would best explain why the approach/tool succeeds in some bugs classes and fails in others.

The insights from the dissection study (SOBREIRA et al., 2018) supported by

[2] Defects4J Dissection: companion website of the published work (SOBREIRA et al., 2018) <http://program-repair.org/defects4j-dissection/>

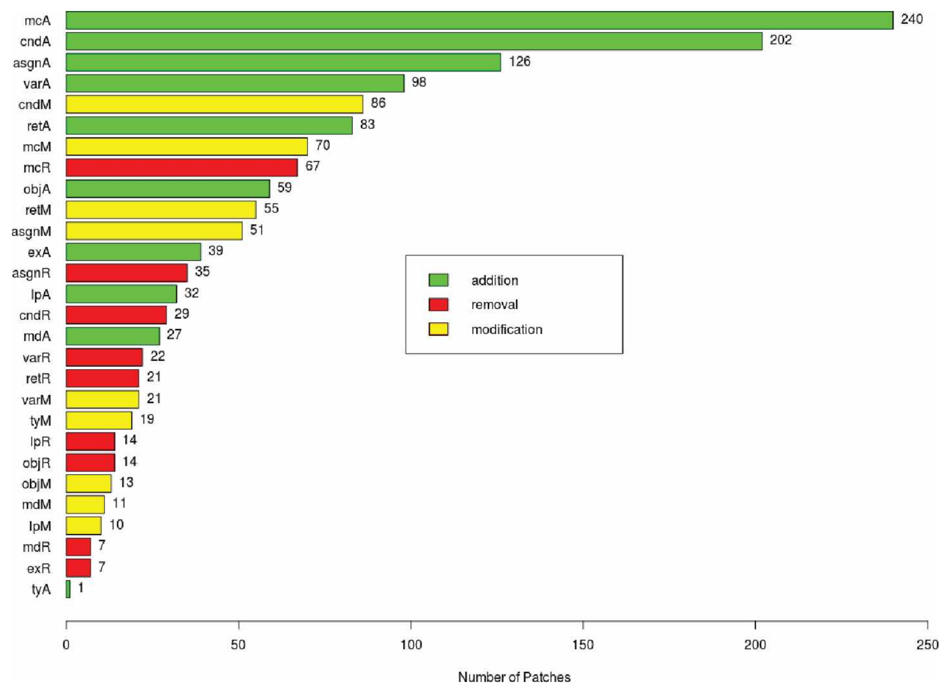


Figure 15 – Repair actions incidence in patches from Defects4J projects.

Figure 16 – Distribution of the number of repair patterns by patch of Defects4J projects.

ADD/PPD tool (MADEIRAL et al., 2018) can be applied to enrich the performance analysis of BL approaches. Knowledge on tools behavior against the bug nature would benefit and make analysis go beyond simple performance measure comparisons. Also, the why, how, and when an approach has better performance than the others would be more clearly justified.

The information about bug nature would also be helpful to leverage ML strategies for bug localization, for example, grouping or clustering bug reports to tune BL tools towards a specific type of bug or even to obtain better results in the learning process.

The ADD tool applied to the LR–dataset for comparison purposes and to contrast how the results obtained from Defects4J would prevail in other datasets and contexts is a natural unfolding and continuation of the first two papers to define the generalizability of ADD.

4.4 Handling imbalanced data

One of the main difficulties in tuning a ML model is the problem with the imbalanced dataset. Studies as in (GONG et al., 2012) have shown the negative impact caused by imbalanced test cases used in Spectrum-Based Fault Localization (SBFL) approaches. For the static BL approaches, this issue is also present since there is a big difference between the small number of source code files affected by a bug (positive examples) compared to the high number of non-buggy source code files (negative examples) in the search space for a given project snapshot. Table 4 shows this relation for the projects in LR–dataset.

Many strategies apply to deal with the imbalance of the datasets in ML classification tasks (HE; GARCIA, 2009), such as Undersampling, Oversampling, Smote, CBO, and Boosting.

The LR approach (YE; BUNESCU; LIU, 2016) has adopted the strategy of limiting the number of source code files input for model training (a kind of undersampling). Only the top k (k = 200 for optimal results) negative examples (non-buggy files) ranked according to the surface lexical similarity feature (F1) are inputted to the training. The

Table 4 – Imbalanced data in LR–dataset.

	# bug reports	# fixed files per bug report			# total of files per bug report		
		max	median	min	max	median	min
Eclipse	6495	587	2	1	6243	3454	382
JDT	6274	118	2	1	10544	8184	2294
BIRT	4178	230	1	1	9697	6841	1700
SWT	4151	430	3	1	2795	2056	1037
Tomcat	1056	94	1	1	2042	1552	924
AspectJ	593	87	2	1	6879	4439	2076

positive examples (buggy files) also compose the dataset training set. In our experimental package, we maintain the same strategy. Since the imbalanced problem compromised the learning process of the experiments in Chapter 5, the inclusion of the other data balancing strategies is also a desirable complement for future works. We also apply different algorithms beyond SVMRank to handle the imbalance of data intrinsically, e.g., LambdaRank and Selective Gradient Boosting (LUCCHESE et al., 2018).

4.5 Data Splitting Strategies

In general, the dataset divides into training, validation, and testing sets during a machine learning process. There is a kind of confusion and mixing in the literature, especially between the validation and testing set concepts. Sometimes, it is unclear when to use one or another term since many papers swap their meaning. Here, we assume the validation set applies for tuning and model selection during the learning process. We consider the evaluation measures using training or validation data as training measures. In its turn, the testing split applies for the final performance measures of the tuned or selected model. Testing data do not apply in the training or validation process. These assumptions are consistent with the following definitions adapted from Ripley's book (RIPLEY; HJORT, 1995):

Training set the sample of data used to fit the model.

Validation set the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyper-parameters.

Testing set the sample of data used to provide an unbiased evaluation of a final model fully specified through the training and validation data set.

A usual strategy for training and validation of ML models is cross-validation, especially the k-fold cross-validation. However, the BL problem has a particular time restriction: given a bug report and the associated patch time-stamp, no data from the bug fixing commit and beyond should be considered for training and testing purposes to avoid biases and overfitting caused by data leakage, i.e. when the training data have the information trying to be predicted (KAUFMAN et al., 2012). Many studies already published can have questionable results because of this issue, especially in the software development area, as pointed out by Tu et al. (2018). An alternative is to apply back-testing, where the dataset is split based on the chronological order of the samples for

training, validation, and testing. For example, the Time Series Forecasting area applies strategies of back-testing to deal with data leakage, and one of the well-known is out-of-sample tests (TASHMAN, 2000). Some of these strategies are:

Walk Forward (LADYZYNSKI; ZBIKOWSKI; GRZEGORZEWSKI, 2013)
a.k.a.: Sliding Window, Roll Forward Cross Validation, Forward Chaining
This strategy sample the dataset in subsets (or windows) of fixed size: oWL (out-of-sample data), a training subset; tWL, a testing subset. Initially, oWL is positioned at the start of the dataset, while the tWL is positioned just after oWL. At each training/testing iteration, oWL and tWL slid by the size of tWL, this way covering all the remaining datasets. The total size of the testing window (TW) is given by the sum of all the iterated tWL blocks, while the first oWL training block represents the data feed length (DFL). When the testing window has a unitary size, this approach is equivalent to the Rolling Window in (TASHMAN, 2000).

Expanding Window
a.k.a.: Rolling Origin, Recursive Forecasting
This strategy is similar to the previous one, except that there is no sliding on the training data (oWL), but a continuous expansion with the inclusion of the testing data used in the last iteration.

Ye, Bunescu e Liu (2016) proposal applies the Walk Forward strategy with many training and testing subsets containing 500 bug reports each (window size), except for AspectJ that employs just a single training and testing subsets with sizes of 500 and 93 bug reports, respectively. We experiment with both strategies using different training and testing sizes, but the whole support still needs improvements to scale better, especially with large training/testing samples.

4.6 Source Code Representation

IR approaches usually process the textual input as word tokens, i.e., in a generic way without assumptions about the nature of the specific domain or context. However, programming languages have intrinsic semantic that can influence the interpretation of the processed input. For example, the order of the tokens handling, the hierarchical structure between tokens, and the scope can be meaningful and change how some bug reports should be associated with a target source code file.

In that sense, future works would research and apply strategies prepared to consider the intrinsic nature of source code, for example, leveraging representations like AST, recognizing the repetitive condition of textual input in the source code file, and considering the language mismatch between the query (bug report) and the corpus (source code files search space).

4.7 New Features for Bug Localization

This section discusses some features that potentially would improve BL. We tested the entropy feature, and some results are shown in Chapter 5. The obtained insights from (SOBREIRA et al., 2018) can be a start point to propose new features based on commits and patches. Word embedding is another technique to explore in the extraction of new features and can alleviate the semantic gap between words between the bug report and source code artifacts. Finally, the last part mentions other potential features that need additional research.

4.7.1 Entropy

We first attempted to improve the BL performance by implementing an entropy feature. The entropy expresses the inverse code "popularity" measure because recurrent code in a target code base receives a lower score than unfamiliar code. The implementation apply the SPL-Core[3] library (HELLENDOORN, 2017) available in GitHub. SPL-Core can extract a LM from a set of Java source code files. The learned LM allows measuring how natural is a token occurrence given a sequence of the predecessor or successor tokens in the code, expressed as a probability score. The aggregated entropy score represents the naturalness of a line computed by averaging the scores of tokens found in that line. The same rationale applies to computing the entropy score for methods and files by aggregating the associated lines. SPL-Core allows to parametrize the computed LM using the following configurations:

n-grams: the number of tokens considered after or before the target token (i.e., the sentence length).

Sentence direction: the entropy can be computed considering the *forward* direction of the sentence, the *backward* direction or both (*bidirectional*).

[3] SPL-Core:<https://github.com/SLP-team/SLP-Core>

Cache: the use of a *cache* model allows to emphasize a local context (i.e., a target file) beyond the whole LM. So for the experiments, it is considered the entropy score with and without cache.

Smoothing: the smoothing techniques allow for a trade-off between the use of long, more specific, but less frequent sentences against short, more general but flooded sentences in the coding corpus. The available smoothing alternatives are Jelinek-Mercer (JM), Witten-Bell (WB), Absolute Discounting (AD), and Kneser-Ney (KN). For the experiments in Chapter 5, we use only JM smoothing. However, we should conduct more tests to measure the impact of these different smoothing models.

SPL-Core does not provide a normalized score that considers the type of the tokens found in lines (e.g., assignment and if-statement). Thus, Eclipse JDT Parser [4] parsed the files, the types of each token were identified and then considered to define the normalized score of each line, using the methodology to compute Z-score of Ray et al. (2016). The following configurations reflect these two entropy alternatives:

En_r: Raw entropy computed from the LM created with SPL-Core. Computed as the average of entropies associated with 1) tokens in a coding line, 2) lines in a method, 3) lines in a file.

En_z: Entropy sensitive to type computed after parsing Java files and identifying the root node in Java AST related to each line. Then, the raw entropy associated with a line normalizations applies according to the AST root node type of this line.

4.7.2 Word2Vec, Glove and ConceptNet

Most of the features applied in the LR approach rely on the textual similarity between the bug report fields (summary and description) and the associated corpus to the target file (i.e., any source code content or API documentation). The features ϕ_1, ϕ_2, $\phi_{7..14}$ are the more impacted by this issue, since their similarity calculation is based on the classical VSM model for IR. A known problem of this approach is the gap of lexical similarity (a.k.a. lexical mismatch) between the comparison targets, especially in the context that source code and bug reports are produced (generally by different people and also in different languages). To illustrate, if a bug report refers to a term like "home"

[4] Eclipse JDT Parser: <https://goo.gl/YgHSiv>

to describe a bug and in the source code the same entity is referred to as "house", the classical VSM model would be of no help for a match in this context, and would contribute to lower the feature score.

Word embedding models fill this gap since it goes beyond lexical similarities comparisons and can approximate even semantic similarities. Word2Vec and Glove are among the most popular current applications. Another recent proposal is ConceptNet Numberbatch (SPEER; CHIN; HAVASI, 2017), derived from Word2Vec and Glove and promising to be a better option. The extraction of features replacing those based on the classical VSM by word embedding and related approaches versions would be a good target for future works.

4.7.3 Commits and Patches

Few approaches apply information based on commits and patches to produce features for BL (WEN; WU; CHEUNG, 2016; LE et al., 2016; ALMHANA et al., 2016; YE; BUNESCU; LIU, 2016; YOUM et al., 2015; WANG; LO, 2014). The work in (WEN; WU; CHEUNG, 2016) proposes the use of commits (i.e., bug-inducing changes) as a finer-grained alternative to the usual file granularity from static information BL approaches. Wen, Wu e Cheung (2016) advocate the use of commits for a better contextualization and to reduce the effort in the developer maintenance task. They also report the improvement in the performance of BL with their approach. BLIA (YOUM et al., 2015) use commit log messages to select commits associated with bugs and uses the recency of these commits as one of the components to compute the suspicious score of a file for a given bug report. The LR approach (YE; BUNESCU; LIU, 2016) applies some features based on the meta-information derived from the bug fixing commits, such as bug fixing recency (ϕ_5) and bug fixing frequency (ϕ_6).

A possible extension would be to introduce Wen's approach (WEN; WU; CHEUNG, 2016) in the LtR process and to explore other related features to the commits and patches contents. The insights from (SOBREIRA et al., 2018) would be integrated into the learning process using, for instance, some patch (or commits) characteristics as input features. We should explore the applied repair actions and patterns, size, and spreading of patches to produce new rankings. Other ideas include learning with patches and identifying more frequent bugs fixing contexts, such as reinforcing similar contexts with higher scores and penalizing not similar contexts with lower scores; based on patch patterns, weight information from patches according to these patterns.

4.7.4 Other features

The enumerated features in the last subsections are only a fraction of what remains to explore. For example, the BM25 and BM25F scores apply in the context of Feature Location, and better results are obtained with MRR metric when compared to classical VSM, Unigram Model and LDA (SHI; KEUNG; SONG, 2014). Since BL can be considered a specialization of Feature Location, the application of the BM25 as a feature in the context of BL is a natural unfolding. Other works that would inspire the adaptation of BM25 in the context of BL comes from Shi, Keung e Song (2014), Saha et al. (2013), Liu (2009), Keyhanipour e Moeini (2016). Another potential feature is the use of stack trace information present in bug reports in a structured way, already explored in BL by Wong et al. (2014a). The primary difference is that Wong et al. (2014a) does not explore stack traces with learning algorithms.

4.8 LtR Tools and Models for BL

The LtR area has been in continuous development in the last years, and many algorithms and tools exist. The LR approach (YE; BUNESCU; LIU, 2016) applies only an SVM based algorithm implemented in the SVM^{rank} tool[5]. In Chapter 5, we show some results with alternative LtR algorithms and compare the performance against the baseline, including parameters and hyper-parameter tuning. Some of the tested tools was:

1. SVM^{rank}: tool to produce baseline results according to (YE; BUNESCU; LIU, 2016);

2. QuickRank[6]: includes implementations for GBRT, LambdaMART, Oblivious GBRT, Oblivious LambdaMART, CoordinateAscent, LineSearch, RankBoost, DART and Selective Gradient Boosting;

3. RankLib[7]: includes implementations for MART, RankNet, RankBoost, AdaRank, Coordinate Ascent, LambdaMART, ListNet, Random Forests;

4. XGBoost[8]: for LtR task with the implementation based on LambdaRank.

[5] SVM^{rank}:<https://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html>
[6] QuickRank: <http://quickrank.isti.cnr.it/>
[7] RankLib:<https://sourceforge.net/p/lemur/wiki/RankLib/>
[8] XGBoost:<https://github.com/dmlc/xgboost>

4.9 Experimental Package for Bug Localization Assessment

As shown in the previous section, many factors influence results of a BL approach. Therefore, an experimental package for dealing with some of the enumerated factors is a significant design challenge. Since the work of Ye, Bunescu e Liu (2016) applies a reasonable number of features inspired and based on other related works, we applied it as our baseline work, implemented in an experimental package prototype. Initially, the package targets BL approaches using static information and has bug reports as the base for the fault localization query.

4.9.1 Experimental Package Overview

The experimental package is a prototype developed to support the experimentation, extraction of the features, producing performance measures, and generating input data for LtR algorithms. Figure 17 gives an overview of the experimental package's general workflow. The BL process depends on data from 1) the source code repository and 2) the bug reports registry system. The source code repository requires information related to the project history, including source code files and API documentation typically stored in a version control system (e.g., Git). The bug reports repository requires information describing the observed issue (title and description) and information to support the ground truth building for performance evaluation and ML-based training. Typically, bug report information comes from Bugzilla, Jira, or Git Issues systems, but the experimental package assumes a simple XML file exported from the mentioned systems. The initial processing steps involve the bug report targets selection, which would imply filtering and quality assessment. Then, the system needs to recover the project version associated with the buggy files existing before the bug patch application. Next, the source code elements need prepossessing to separate the buggy suspects from other code elements non-related to the bugs or external to the project (e.g., libraries or non-functional/testing files). The code is parsed and indexed for a database with the potential buggy suspects targets, and then the feature extraction starts to allow the ranking building. Features such as the 19 from Ye, Bunescu e Liu (2016) work are the base for score generation associated with each code element (i.e., source code files) suspected to contain a bug. The computed scores support the ranking generation produced by both strategies for BL: LtR-based algorithms (e.g., our LR-based approach and other third parties LtR-based libraries and tools) or Non-ML-based algorithms (e.g., BLUiR approach). Finally, we can sample bug

reports targeting the performance assessment with the generation of specific measures (e.g., MAP and MRR).

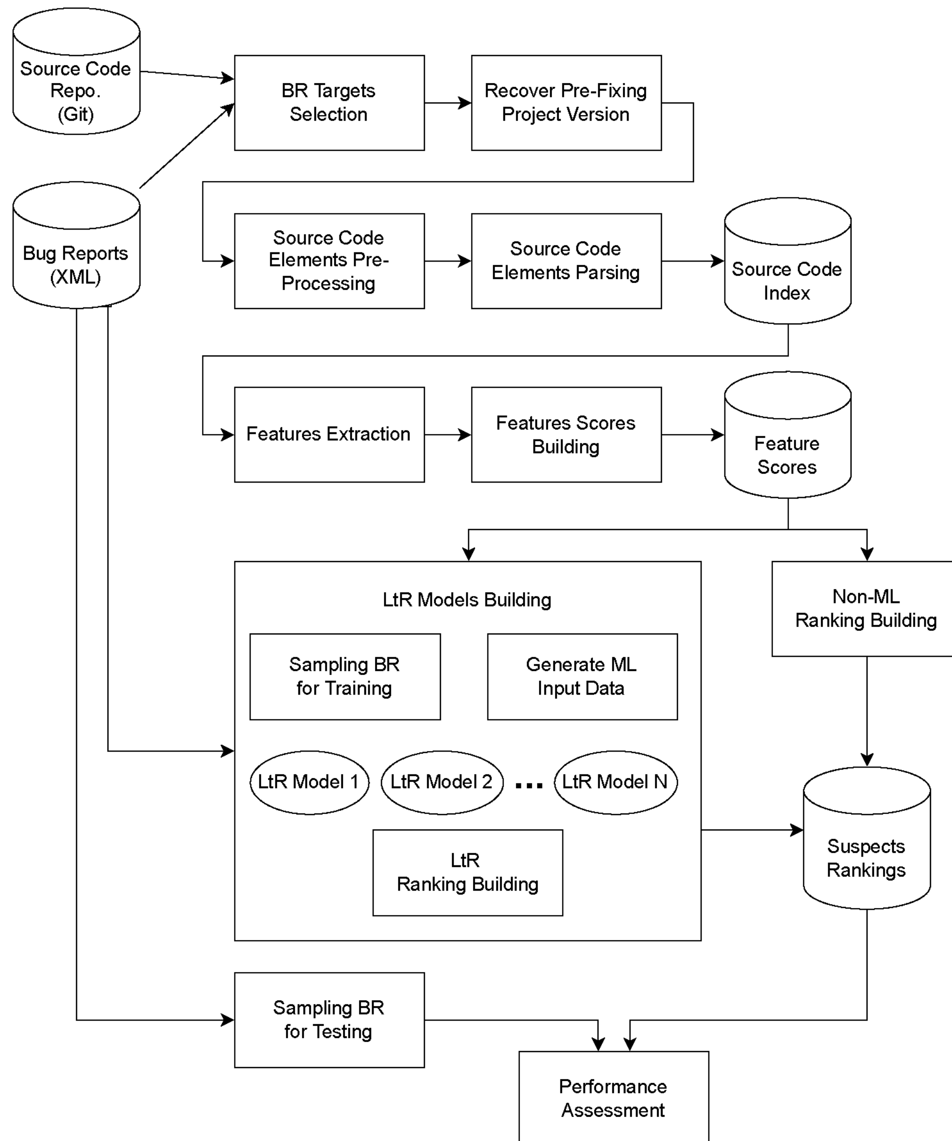


Figure 17 – BL process overview and associated modules of the experimental package.

There is no intention to describe the implementation details since it should change a lot with refactorings and re-design required to improve modularity, maintainability, and performance bottlenecks observed during the development of the experimental package. Furthermore, we do not consider it a reference architecture in the current state.

The main technologies and libraries applied were:

Programming Languages: Python, Java, Shell Script;

Python Libraries: Pony Object Relational Mapping/Mapper (ORM), Natural Language Toolkit (NLTK), Scikit-learn, Py4J, JavaScript Object Notation (JSON), Pandas, Jupyter Notebook, Anaconda;

Java Libraries: Eclipse JDT parser, Java Universal Network/Graph Framework (JUNG), SPL-Core;

Databases: SQLite and Postgres were initially applied and for comparisons purposes, but most parts of the final results were using MySQL/MariaDB in the last implementation;

Repository: GitHub.

Python is the primary language targeting the future application of Tensor Flow library (for Deep Learning (DL) models) and considering the extensive support for data science projects of the Python libraries (e.g., Scikit-learn, Jupyter Notebook, Pandas, Anaconda). Since the projects in the LR–dataset are in Java, Py4J is in charge of allowing access to Eclipse JDT parser, JUNG library for Graph algorithms, and SPL-core library for entropy calculations. In addition, Python and Shell's scripts were employed for experiments, interface with Git repository, and version control. The data model extracted from source code persists with Pony ORM in a MySQL database server. The following subsections briefly describe some interfacing parts related to the input and output.

4.9.2 Input

The input for the framework is composed of:

1. bug reports: bug id, description, summary, open timestamp;
2. bug fixing: timestamp, associated commit, associated fixed files;
3. source code: all source code file versions associated with the fixed version and the buggy version just before the fixing for a given bug report;
4. API documentation (embedded in source code): especially, documentation for classes and methods;

Bug reports and bug fixing information comes from XML files associated with each LR-dataset project. In addition, source code and API documentation come from the GitHub repository of each project.

Bug reports (preprocessed according to the criteria defined in Section 4.1) and the list of selected datasets are forwarded to feature extraction.

4.9.3 Feature Extraction

We compute ranking scores from a set of features for the selected bug reports and related file versions. The replication of the original 19 features in (YE; BUNESCU; LIU, 2016) was as close as possible to the methodology adopted in that paper. Additionally, the Entropy feature as discussed in Section 4.7.1 is also implemented. After each feature extraction, the associated ranking are built and stored in the database.

All feature scores were normalized values according to Equation 19.

$$score_{Norm} = \frac{score - \mu}{\sigma} \tag{19}$$

μ is the average value for all the file versions used in the feature extraction, and σ is the associated standard deviation.

4.9.4 LtR Input

Once the scripts generate the individual rankings, it is possible to feed them in LtR algorithms. Using all the file versions scores would compromise and make it unfeasible by memory, time, or processing constraints. Thus, the feeding occurs by sampling file versions associated with each bug report and using the normalized scores obtained for each feature. There are two types of samples: the positive examples, representing the fixed or buggy files; negative examples represent the non-fixed or buggy-free files to the associated bug report. All the positive examples are considered for the learning process, while some heuristics apply to select negative ones. The original approach in (YE; BUNESCU; LIU, 2016) selects the Top-200 negative examples based on the ranking of the feature ϕ_1 (Surface Lexical Similarity).

Ye, Bunescu e Liu (2016) approach limits the number of bug reports for training and testing in the baseline approach. First, the bug reports are sorted chronologically by their timestamp. Then, the most recent reports have priority in the selection. The optimal number of bug reports used for training is 500 bug reports, while 100 bug reports for testing, in most of the dataset projects, according to experiments in (YE; BUNESCU; LIU, 2016). Beyond the approach to choosing a fixed number of bug reports for training and testing, the strategies enumerated in Section 4.5 are also employed.

The SVM-Light plain text format is the usual choice to prepare the input for the LtR algorithms. Figure 18 shows the rules to define a line input in SVM-light. Each line represents a one-sampled file. The *target* field represents the condition of the file for a given bug report: value 0 if the file is a negative example (non-buggy); value one if the file is a positive example (buggy). The special feature *qid* identifies the bug report, the respective samples, and the *bug-id* info. Following *qid* are the features identified by positive integers indexes and valued with the normalized score of the respective feature. The info field receives the file identifier to facilitate later analysis, and the LtR algorithms do not apply it for learning purposes.

```
<line> .=. <target> <feature>:<value> ... <feature>:<value> # <info>
<target> .=. +1 | -1 | 0 | <float>
<feature> .=. <integer> | "qid"
<value> .=. <float>
<info> .=. <string>
```

Figure 18 – SVM-light format.

To generate the training SVM-light files[9], the strategies enumerated in Section 4.4 are employed, beyond the (YE; BUNESCU; LIU, 2016) baseline strategy discussed previously.

While the training files receive samples of the file versions scores, the testing files receive all the file versions scores for each bug report. Thus, the final ranking generated for performance evaluation using the chosen metrics for LtR is more representative of the actual situation while searching for the buggy files and ranking all available and relevant source code files in a project.

Beyond the data in SVM-light format, we should inform many parameters and hyperparameters before starting the learning process. Some parameters are general, e.g., the metric to be used, and other parameters are specific to an algorithm or a class of them, e.g., the number of leaves for tree-based algorithms and type of kernel function for SVM-based.

4.9.5 LtR Output

The output of the learning process depends on the tool applied (as discussed in Section 4.8). Generally, the LtR tools produce a model for testing, production, and predictions on training or testing data. Unfortunately, there is no standard format for the outputs.

[9] SVM-light format: <http://svmlight.joachims.org/>

The QuickRank tool has two basic outputs:

1. Model: a XML file with the learned model, composed of two main sections.

   - The first section is the info section that contains all the parameters to run the LtR algorithm. Tree-based models have general parameters, e.g., the maximum number of trees, leaves, and shrinkage. Some algorithms have specific parameters, such as DART (e.g., sample type, adaptive type, rate drop), Oblivious MART and Oblivious LambdaMART (e.g., depth), Coordinate Ascent and Line Search (e.g., number of samples, window size), and LambdaMART Selective (e.g., sampling iterations, rank sampling factor, adaptive strategy negative strategy);
   - The second section is the model section with the generated LtR model written, most parts as an ensemble of binary trees (e.g., DART, MART, and LambdaMART), where each tree has a weight, and the nodes represent the threshold for the model selected features. The exception in the model format is for Coordinate Ascent, which only enumerates a weight for the features, and Line Search, which contains ensembles of single node trees, each representing a feature with the related weight.

2. Predictions: a plain text with the predictions for training and testing and extra configuration information.

   - Some sections fixed in this file containing general information about the tool, parameter configurations (replicated from the XML model file), paths for the dataset involved (training and validation), runtime info (such as training time and reading time), dataset size (instance x features) and the number of queries, chosen training metric (NDCG@k, MAP, ...).
   - Most interesting part is the performance measures for training and validation. Most of the algorithms present the results in three columns, in the format $< iteration > < training_score > < validation_score >$, each line contains the value obtained in the associated iteration. In addition, some algorithms present extra information, such as DART (dropped trees, ensemble size, and dropout info) and Line Search (e.g., gain, window, and red factor).

The SVM^{rank} tool also produces one output for the learning process and one for the classification (prediction).

1. Model: a plain text file with the learned model, composed of:
    - kernel and other configuration parameters used in the SVM algorithm (one per line);
    - feature weights in the last line in the format <feature-index>:<weight>.
2. Predictions: a plain text file with the scores predicted (numeric float values) associated with each line in the input testing dataset file (in SVM-light format discussed in Section 4.9.4). We can use the scores values as the ordering key to recovering the final ranking for each bug report (the higher the score, the higher is the relevance of the file). However, we do not have automatic performance measuring metrics like QuickRank.

The RankLib tool produces an output similar to QuickRank:

1. Model: a plain text file composed of two main sections.
    - The first section contains all the parameters according to the chosen algorithm, one per line, started with "##" and followed by the parameter and value in the format $< parameter > = < value >$.
    - The second section contains the model written in plain text in a format dependent on the selected algorithm. Tree-based algorithms (e.g., MART, LambdaMART, and RandomForests) are written in an XML-like markup and represent the ensemble of trees as in QuickRank. Coordinate Ascent enumerates feature weights in the format $< feature >:< weight >$ in the file last line. RankNet lists all the Neural Net architecture information, including layers and weights.
2. Predictions: screen output with the predictions for training, validation and testing, extra information of configuration, and runtime.
    - As before information, there is a section about loaded files, algorithm parameters, dataset files (training, validation, and testing), and status about the processing.
    - The performance measures section is also similar to QuickRank. Most of the algorithms present the results in three columns in the format $< iteration > \ |< training_score > \ | \ < validation_score > \ |$, and each line contains the value obtained in the associated iteration. In addition, some algorithms

present extra information, such as RankNet (% mis-ordered), RankBoost (selected feature, threshold, and error), ListNet (c.e. loss), and AdaRank (selected feature). Other algorithms show another kind of output format, such as Coordinate Ascent, presenting blocks of information in the format $< feature > \mid < weight > \mid < score > \mid$.

- Final section shows the final computed performance metrics for training, testing, and validation and confirms the file to output the model.

4.10 Final Considerations

We developed an experimentation package to explore factors that can impact the performance of a BL approach. Through the experimental package implementation process, we perceived a need for an environment to experiment with, reproduce, assess and compare BL approaches aiming to propose improvements to the state-of-the-art. The ideal environment would be able to 1) facilitate the experimentation process, 2) introduce new features and composition strategies to generate and improve ranking scores for software components from a project suspected to be buggy, 3) allow comparisons between the approaches, experiment ideas and highlight strengths and weakness of the assessed approaches, 4) allow to sample bugs for experimentation based on bugs intrinsic characteristics, e.g., type of patch required and other associated dimensions, 5) deal with many challenges on the construction, e.g., reproduce past experiments, extract and process the massive amount of data, 6) optimize the use of resources (e.g., computational, memory, and storage), 7) integrate everything, from feature extraction to testing and comparisons of approaches. Many of the enumerated influential factors in this chapter would compose this ideal environment. While our experimental package made the generation of data and analysis possible for the experiments in this thesis, we just started to cover some of the factors described in this chapter. Additional work remains to achieve a complete and practical environment usable by the research community.

Chapter 5

Strategies for learning-to-rank bug localization improvement

This chapter presents the exploratory results obtained with the implemented models and the experimental package. The first section, Evaluation Method, details research questions proposed, bug dataset applied, data preparation and cleaning procedures, configurations and parameters for the experiments, metrics, and the run-time environment. The final section, Results, details the obtained results and analyzes each research question.

5.1 Evaluation Method

The initial experimentation target to answer the following research questions:

RQ1 What is the performance of the entropy feature compared to other features?

RQ2 The use of entropy feature can improve the results obtained by past learning approaches to BL?

RQ3 What is the impact of data balance strategies in the learning process?

RQ4 How does the tuning of LtR algorithms impacts the BL performance?

RQ5 How long does it take to conclude each step in the process (feature extraction, ranking generation, training, validation, and testing)?

5.1.1 Dataset

The target dataset used in the exploratory studies is the LR–dataset (YE; BUNESCU; LIU, 2014), briefly described in Section 2.2.2. However, we did not cover all projects and data in LR–dataset in these exploratory experiments to reduce computing time and because the current state of the experimental package did not scale to complete the processing in a reasonable time. Thus, we choose to sample only part of the data from Tomcat, SWT, and AspectJ from the six available projects in LR-dataset.

5.1.2 Data Preparation and Cleaning

The pre-processing of data follows the procedures of Ye, Bunescu e Liu (2016) work, as close as possible. Textual data extracted from bug reports and source code, especially those used in similarity comparisons (most of the features ϕ_1 to ϕ_{14}), include the steps:

Word tokenization input document split into words using white spaces.

Stop Word Filtering numbers, punctuation, and stop words are filtered using NLTK English stop words list.

Compound Words common in code entity names are split based on Camel Case (Capital letters) and added to document corpus.

Porter stemming is applied to all words also using NLTK package.

While analyzing the bug reports set in LR–dataset, we observed the presence of bug reports where no fixed file is informed. These types of bug reports, even considering their relation with an actual bug, are of no help as ground truth since we cannot directly recover the fixed code. Thus, we opt to exclude bug reports in this condition for this work. Similarly, we exclude bug reports containing only test files since they do not represent fixing patches for functional program behavior. As some studies have pointed out the disadvantages of using a dataset containing test files as fixing targets, we also opted to exclude from the search space the source code files containing only test cases or test suites. A manual analysis allowed the selection of folders for exclusion, and, fortunately, most projects use standard folders, easy to recognize as testing folders, and separate them from functional code. External files (usually libraries) were not considered candidates for fixing.

Many instances have missing values for some features, e.g., there is no reason to extract the method similarity from classes or interfaces without the implemented methods,

as could occur in computing of ϕ_8 and ϕ_{12} (Summary-method and Description-method names similarity). For these and other cases of missing values for features, we opted to replace the missing value with zero, considering that the range of normalized values goes from 0 to 1. We do not proceed with duplicated instances analysis (e.g., considering duplicated bug report). We would need additional experiments to assess the impact of replacing missing values with other alternatives (e.g., the average) and removing duplicated instances.

5.1.3 Experiment Configurations

This subsection presents all the configurations of parameters used in the experiments, including baseline adapted from (YE; BUNESCU; LIU, 2016) and additional configurations related to features, data balance strategies, and LtR algorithms. The last sections present evaluation metrics and the run-time environment.

5.1.3.1 Baseline

The implementation in this work is an adaptation from the paper (YE; BUNESCU; LIU, 2016), discussed in details in Section 3.1.1. The following configurations build the baseline:

LR-All: all 19 features presented by Ye, Bunescu e Liu (2016) and enumerated in Table 5;

LR-6Best: the six features with the best-combined performance for each project according to Ye, Bunescu e Liu (2016) (see Figure 20 in that paper for details). Table 6 summarize the features;

LR-6Exp: the six features with the best performance in individual rankings in this implementation (variate with the experiment).

5.1.3.2 Entropy Feature

Experiments with Entropy Feature applied following configurations (described in Section 4.7.1):

n-grams: values in range 3, 6, 10.

Sentence direction: set to forward, backward and bidirectional

Table 5 – Original features in (YE; BUNESCU; LIU, 2016).

Feature	Short Description	Query Dependent?
ϕ_1	Surface lexical similarity	yes
ϕ_2	API-enriched lexical similarity	yes
ϕ_3	Collaborative filtering score	yes
ϕ_4	Class name similarity	yes
ϕ_5	Bug-fixing recency	yes
ϕ_6	Bug-fixing frequency	yes
ϕ_7	Summary-class name similarity	yes
ϕ_8	Summary-method name similarity	yes
ϕ_9	Summary-variable name similarity	yes
ϕ_{10}	Summary-comments similarity	yes
ϕ_{11}	Description-class name similarity	yes
ϕ_{12}	Description-method name similarity	yes
ϕ_{13}	Description-variable name similarity	yes
ϕ_{14}	Description-comments similarity	yes
ϕ_{15}	In-links = # file dependencies of s	no
ϕ_{16}	Out-links = # files that depend on s	no
ϕ_{17}	PageRank score	no
ϕ_{18}	Authority score	no
ϕ_{19}	Hub score	no

Cache: set to "with Cache" (default).

Smothing: set to "Jelinek-Mercer - JM" (default).

Entropy type: set to "type sensitive - En_z".

We pin the last three configurations in the experiments, so the reference key to identifying the Entropy configuration points only to the n-grams and the sentence direction

Table 6 – Best features per project according with (YE; BUNESCU; LIU, 2016)

Project	Features
Eclipse Platform UI	ϕ_1, ϕ_7, ϕ_4, ϕ_3, ϕ_{15}, ϕ_{11}
JDT	ϕ_1, ϕ_3, ϕ_7, ϕ_4, ϕ_2, ϕ_{12}
BIRT	ϕ_3, ϕ_4, ϕ_{11}, ϕ_{10}, ϕ_2, ϕ_8
SWT	ϕ_4, ϕ_6, ϕ_1, ϕ_3, ϕ_{11}, ϕ_5
Tomcat	ϕ_1, ϕ_3, ϕ_6, ϕ_{10}, ϕ_2, ϕ_{14}
AspectJ	ϕ_1, ϕ_{12}, ϕ_4, ϕ_6, ϕ_3, ϕ_{10}

Table 7 – Entropy features computed in exploratory experiments.

Feature	Short Description	Query Dependent?
ϕ_{20-1}	File Entropy, Forward Sentence, 3-ngrams	no
ϕ_{20-2}	File Entropy, Backward Sentence, 3-ngrams	no
ϕ_{20-3}	File Entropy, Bidirectional Sentence, 3-ngrams	no
ϕ_{21-1}	Method Entropy, Forward Sentence, 3-ngrams	no
ϕ_{21-2}	Method Entropy, Backward Sentence, 3-ngrams	no
ϕ_{21-3}	Method Entropy, Bidirectional Sentence, 3-ngrams	no
ϕ_{22-1}	Max Entropy between ϕ_{20-1} and ϕ_{21-1}	no
ϕ_{22-2}	Max Entropy between ϕ_{20-2} and ϕ_{21-2}	no
ϕ_{22-3}	Max Entropy between ϕ_{20-3} and ϕ_{21-3}	no

values, i.e., En_{10f}, for entropy with ten n-grams and forward sentence direction, and En_{3b} for entropy with three n-grams and bidirectional sentence direction. Table 7 shows the combination of configurations of the Entropy feature used in this work.

5.1.3.3 Data Balance Strategies

WE apply under-sampling strategies over the negative examples in the following configurations:

Single Feature Top-K: this is the strategy used by the LR approach in baseline experiments, with $k = 200$ and ϕ_1 is the feature for ordering and getting the top files. Example: SF_{k200} indicates we picked the 200 top negative files from the selected feature ranking.

Multi-Features Top-K: a variation in the previous strategy is using more features rankings to pick the negative examples. Example: MF_{k33f6} indicates the top 33 negative examples from the selected features rankings are picked, and the 6 most impacting features overall projects is took, i.e., $\{\phi_1, \phi_3, \phi_4, \phi_6, \phi_7, \phi_{12}\}$, according with Table 6.

Multi-Features Var-K: in this case, we use many feature rankings, each with a specific number of negative examples picked. A heuristic was applied to rank, select, and weight the features according to their appearance in the first six positions for each project, shown in Table 6. One appearance in the first position scores 10, the second position scores 8, and so on, until the sixth position scores 1 point for each appearance. Example: MF_{k200w} and a mapping with the number of files selected

Table 8 – Selection and weighting of features from Table 6.

Feature	Appearence Count 1st	2nd	3rd	4th	5th	6th	Weight	Samples	Ranking
ϕ_1	4	0	1	0	0	0	**46**	59	1°
ϕ_2	0	0	0	0	3	0	6		
ϕ_3	1	2	0	2	1	0	**36**	46	2°
ϕ_4	1	1	2	1	0	0	**34**	43	3°
ϕ_5	0	0	0	0	0	1	1		
ϕ_6	0	1	1	1	0	0	**18**	23	4°
ϕ_7	0	1	1	0	0	0	**14**	18	5°
ϕ_8	0	0	0	0	0	1	1		
ϕ_9	0	0	0	0	0	0	0		
ϕ_{10}	0	0	0	2	0	1	9		
ϕ_{11}	0	0	1	0	1	1	9		
ϕ_{12}	0	1	0	0	0	1	**9**	11	6°
ϕ_{13}	0	0	0	0	0	0	0		
ϕ_{14}	0	0	0	0	0	1	1		
ϕ_{15}	0	0	0	0	1	0	2		
ϕ_{16}	0	0	0	0	0	0	0		
ϕ_{17}	0	0	0	0	0	0	0		
ϕ_{18}	0	0	0	0	0	0	0		
ϕ_{19}	0	0	0	0	0	0	0		
						Total	157	200	

for each feature is provided, e.g., {ϕ_1: 59, ϕ_3: 46, ϕ_4: 43, ϕ_6: 23, ϕ_7: 18, ϕ_{12}: 11}, limited to 200 files and detailed in Table 8.

For these configurations, the undersampling applies only to negative examples (non-buggy files), all the positive examples are included (buggy files). To be fair in the comparisons, the total number of negatives files sampled in each configuration for each bug report will be as close as possible each other.

5.1.3.4 LtR Algorithms applied

The experiments involved a total of 14 LtR different algorithms, all them from third-party libraries and first listed in Section 4.8. Some have more than one implementation (one from QuickRank and one from RankLib). Next, the algorithms are enumerated:

SVM^{rank}: Rank SVM, algorithm also applied in baseline reference work (YE; BUNESCU; LIU, 2016);

QuickRank: DART, LambdaMART, MART, Oblivious LambdaMART, Oblivious MART, Random Forest, Stochastict;

RankLib: AdaRank, Coordinate Ascent, LambdaMART, Linear Regression, ListNet, MART, RankNet, RankBoost, Random Forest;

5.1.4 Metrics Extracted

The performance measures were based mainly on the metrics described in Section 2.3: MAP, MRR, Top-N, NDCG@10. For example, the range of values for Top-N is in the set {1, 5, 10}.

5.1.5 Runtime Environment

The experiments ran in a Lenovo ThinkServer TD340 with following specifications:

CPU: 12-core Intel Xeon E5-2430 v2 @ 2.50GHz.

GPU: GM206GL [Quadro M2000] NVIDIA.

RAM: 32 GB RAM (2 x 8 GiB DIMM DDR3 1600 MHz 0.6 ns, 1x16 GiB DIMM DDR3 1600 MHz 0.6 ns).

Hard Disk: 600 GB Seagate Savvio 10K.6 SAS 6 GBS (ST600MM0006).

OS: Experiments started in Open Suse 42.3, and after a system crash, the host Operational System was replaced by Ubuntu 18.04 LTS and Open Suse 42.3 run in VirtualBox to maintain database compatibility.

5.2 Results

This section shows the results obtained in the experiments and answers the research questions proposed before. The analysis and discussions go along with data presentation.

5.2.1 RQ1: What is the performance of the entropy features compared to other features?

The experiments use data from AspectJ and SWT projects to measure the performance of each feature. First, the initial 200 bug reports from AspectJ were selected

(chronologically ordered according to the bug fixing timestamp). Then, we selected 300 bug reports from SWT: 200 bug reports from positions 200 to 400 and 100 bug reports from positions 700 to 800. The original idea was to cover more bug reports, but entropy computation consumes many resources (memory and processor). SWT entropy extraction was spent from 4.2 to 17.0 minutes by bug report on average, which gives from 21 to 85 hours to complete Entropy extraction for these exploratory studies. Since to compute all the 19 features, the time spent by bug report was between 3.1 to 5.4 minutes on average, we did not spend too much time with entropy computation in the exploratory studies. We should proceed with the optimization and refactoring of the experimental package code to make the coverage of all the dataset viable.

Figures 19 and 20 show the results obtained considering MAP, MRR, Top-N and NDCG@10. The general results obtained for the different metrics are consistent, and the best performance features are almost the same, no matter the metric. For AspectJ, the best feature by far is ϕ_1, while for SWT, the best feature is ϕ_4. It is interesting to note that this simple divergence can have an impact on the learning process since the feature ϕ_1 is not the best for SWT, and in the baseline experiments in (YE; BUNESCU; LIU, 2016), it applies in the sampling of the top negative examples for learning.

The results obtained for the entropy individually show poor performance in these exploratory experiments. AspectJ has the worst performance compared to SWT, and the baseline features show far superior performance (it is important to note that the charts have different vertical scale range). For SWT it is possible to see that ϕ_{21-1} and ϕ_{22-1} has better results than features ϕ_9, ϕ_{13}, and ϕ_{16}. These results can be a sign that entropy would have a role in the performance improvement for some bugs.

In the baseline, previous work (YE; BUNESCU; LIU, 2016), query-independent features (ϕ_{15} to ϕ_{19}) were between the worst individual features in terms of MAP results. However, entropy is also a query-independent feature, and it is not a big surprise that its performance would not be so different from the baseline query-independent features. Thus, according to these results, we confirm this behavior since query-independent features do not significantly impact the ranking as the query-dependent features in general.

As the results involve a small number of bug reports, we should consider these results with care compared to the previous study by Ye, Bunescu e Liu (2016) with around 500 bug reports for training and testing. For further studies, the expectancy is on the complementary role of entropy when combined with other features. We need more experiments to confirm this hypothesis and cover more projects and bug reports.

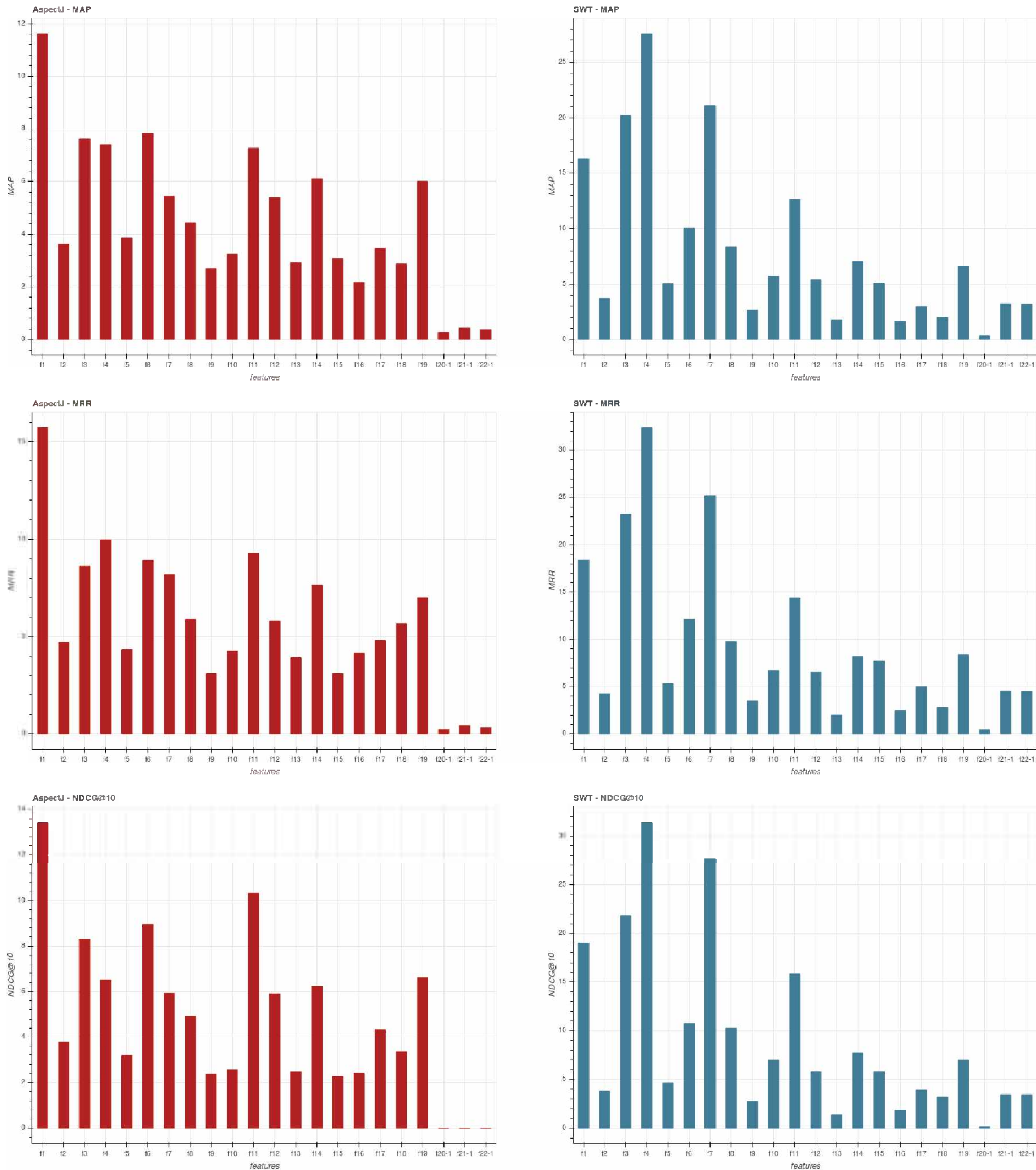


Figure 19 – MAP, MRR, NDCG@10 of AspectJ and SWT features.

5.2.2 RQ2: The use of entropy feature can improve the results obtained by past learning approaches to BL?

A similar setting to RQ1 was employed to evaluate the impact of entropy in the learning process, and only SWT was evaluated. Figure 21 shows the results obtained applying SVMRank tool to learn from 200 bug reports of SWT and test on 100 bug

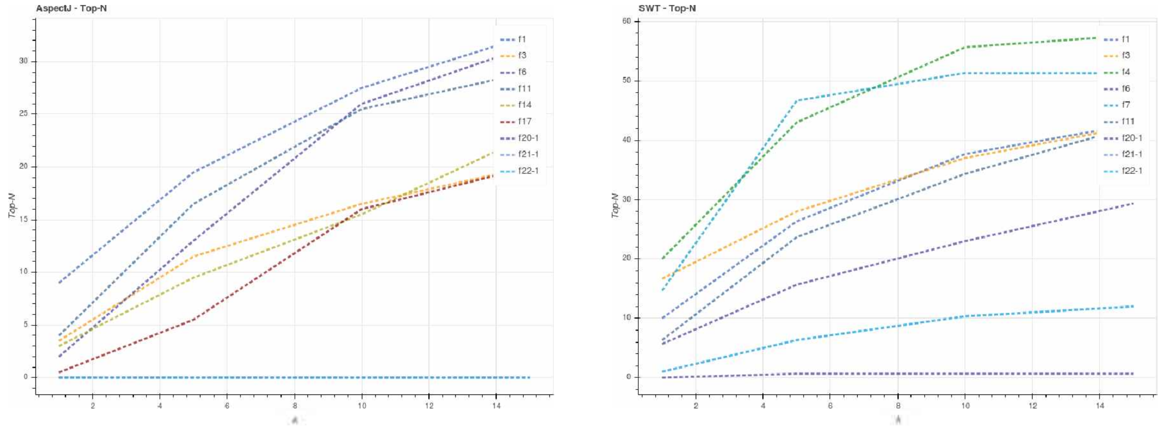


Figure 20 – Top-N of AspectJ and SWT features

reports. The baseline is represented by LR–All setting, including features ϕ_1 to ϕ_{19}. Another baseline is represented in LR–6Best, including the 6 most impacting features overall according to Table 6 and discussed in section 5.1.3.3, i.e., {ϕ_1: 59, ϕ_3: 46, ϕ_4: 43, ϕ_6: 23, ϕ_7: 18, ϕ_{12}: 11}. The additional settings include the entropy features individually to each baseline: ϕ_{20-1}, ϕ_{21-1}, ϕ_{22-1}. The capacity parameter of SVMRank was set to values in the set {0.01, 0.1, 1, 10, 100, 200}. The evaluation metrics were MAP, MRR, Top-N and NDCG@K, with N and K values in the set {1, 5, 10}.

The general observed behavior was consistent between all metrics. The best performance was with capacity set to 0.1 for most metrics settings with some oscillations towards a capacity value of 0.01. The original LR experiment (YE; BUNESCU; LIU, 2016) has found the optimal performance with capacity in the interval [10, 100]. Interesting to note that settings with the six best features have shown better performance than settings with all features, while in the original experiment, this does not occur for the SWT project. Our best performance results are also slightly better than the results found in the original LR work. For example, the best MAP and MRR values reported in the original LR experiment were 40.0 and 46.0, while we have found a MAP value of 42.75 and an MRR value of 51.36, both with the setting LR-6Best+f20.1 and capacity = 0.1. Nevertheless, we need additional experiments, including more bug reports in the training set (the original LR approach had a training size of 500 bug reports, while only 200 bug reports were applied) and involving the other projects to confirm these findings.

Although the best absolute results were with the use of entropy feature $\phi_{20.1}$, the improvement over the baseline in these experiments is marginal: for MRR metric, an improvement of 2.21% over LR–6Best, and 4,90% over LR–All; for MAP metric, an improvement of 0.25% over LR–6Best, and 3.69% over LR–All; for Top-1 metric, an

improvement of 2.56% over LR–6Best, and 8.11% over LR–All; the improvement found for NDCG@1 is the same than Top-1. Considering LR–All only, the best corresponding entropy setting LR–All+21.1 performed worst than the LR–All baseline setting: -0.82% for MRR, and -0.49% for MAP. We need more experiments to confirm the role of entropy in the learning process, but based on these results, the expectancy is to obtain low improvements if the same settings are applied.

5.2.3 RQ3: What is the impact of data balance strategies in the learning process?

To evaluate the influence of the balancing strategies enumerated in Section 5.1.3.3, many experiments were conducted using different LtR algorithms available in RankLib and QuickLearn tools. Figures 22 and 23 present the results for experiments over Tomcat using 100, 250 and 500 bug reports for training and 100 bug reports for testing. Baseline stragegy SF_{k200f1} is shown in green, MF_{k33f6} is shown in yellow and MF_{k200w} is shown in blue. The metric aplied in the comparisons was NDCG@10. Only features ϕ_1 to ϕ_{19} were considered.

By training with 100 bug reports, the differences are not meant for most of the algorithms, while training with 250 and 500 bug reports, it is notable the positive contribution of the data balancing strategies in the performance of the learning algorithms. Although the performance gain variates a lot between the algorithms, there is some consistency between the tools. For example, LambdaMART has a maximum performance gain of 52.3% and 70.86% over baseline SF_{k200f1} strategy, respectively, in QuickRank and RankLib implementations. Similar comparisons also confirm these performance gains, e.g., with MART, Random Forest. However, it is fair to state that this high gain in the performance would be more related to the overfitting with SF_{k200f1} strategy, causing poor performance in training with 250 and 500 bug reports, so the merit of the data balancing strategies is mainly related to the avoidance of overfitting in the algorithms.

The best performance obtained in each tool were:

QuickLearn: LambdaMART and Stochastic with NDCG@10 = 59.64%, using MF_{k200w}, overcomming SF_{k200f1} in 52.30% and MF_{k33f6} in 3.97%, considering the same algorithm.

RankLib: LambdaMART with NDCG@10 = 57.92%, using MF_{k33f6}, overcoming SF_{k200f1} in 70.86% and MF_{k200w} in 2.87%, considering the same algorithm.

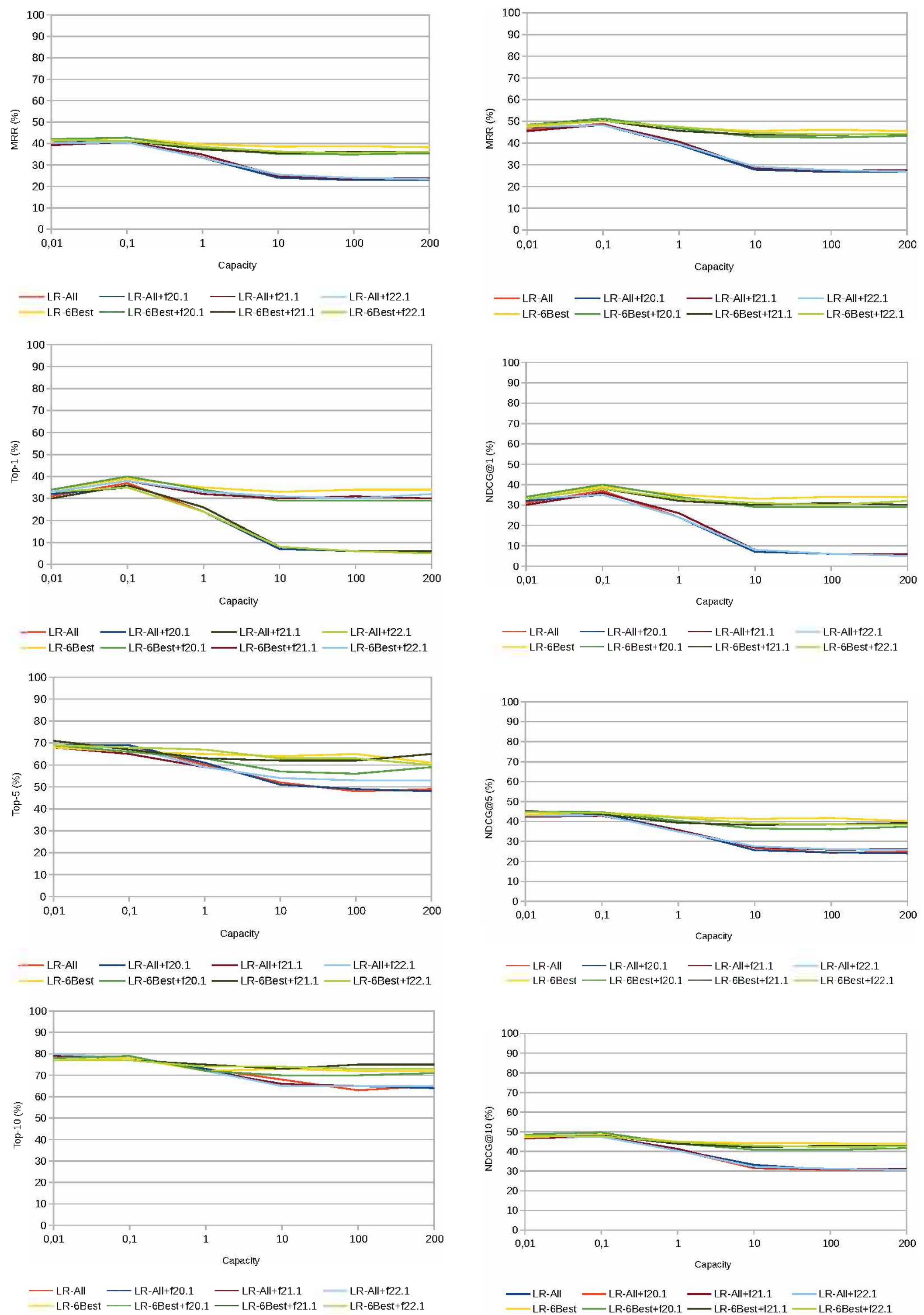


Figure 21 – MAP, MRR, Top-{1,5,10} results on 300 bug reports of SWT (200 for training and 100 for testing)

The balancing strategies did not positively impact RankNet, RankBoost, and AdaRank. For these algorithms, SF_{k200f1} overcame MF_{k200w} and MF_{k33f6}. Even considering the best-obtained result of these algorithms (Rankboost with NDCG@10 = 55.72%), LambdaMART with MF_{k33f6} is still better 3.95%.

5.2.4 RQ4: How does the tuning of LtR algorithms impact the BL performance?

The influence of the tuning in LtR for BL was first shown in RQ2 while evaluating the SVMRank tool with different values for the capacity parameter. The performance range changes a lot depending on the capacity value applied. To illustrate, the Table 9 shows the performance range variation for LR–All setting. The high decay observed between the best and the worst settings reinforces the need to tune the algorithms before analyzing and insights. Since it is usual to find works relying only upon the default parameters, this is another issue because they can obfuscate the optimum results and even lead to a premature discarding of a tool or algorithm.

Metric	Best	Worst	Decay (%)
MRR	48.96	27.31	42.22
MAP	41.23	23.81	42.81
Top-1	37.00	6.00	83.78
Top-5	69.00	48.00	30.43
Top-10	79.00	63.00	20.25
NDCG@1	37.00	6.00	83.78
NDCG@5	43.88	24.24	44.76
NDCG@10	47.85	30.66	35.92

Table 9 – Performance variation of SVMRank on SWT by tuning the algorithm with capacity parameter in LR-All setting.

To extend this analysis, we proceed with experiments using the LambdaMART algorithm in RankLib tool over the 300 bug reports of SWT and comparing baseline setting (LR-All) with baseline + entropy (LR-All+$\phi_{20.1}$). The metrics extracted were MAP and NDCG@10. The parameters to tune were Number of Trees {32, 64, 128, 256, 512}, Number of Leaves {1, 2, 5, 10} and Shrinkage {0.05, 0.5, 0.8}. We defined the last parameter's value sets after some experiments and also based on the optimum values found in the previous work of Ye, Bunescu e Liu (2016). We opted to restrict the values

Figure 22 – NDCG@10 for MART (Lambda and Obliviuos) and Random Forest algorithms from QuickRank and RankLib tools.

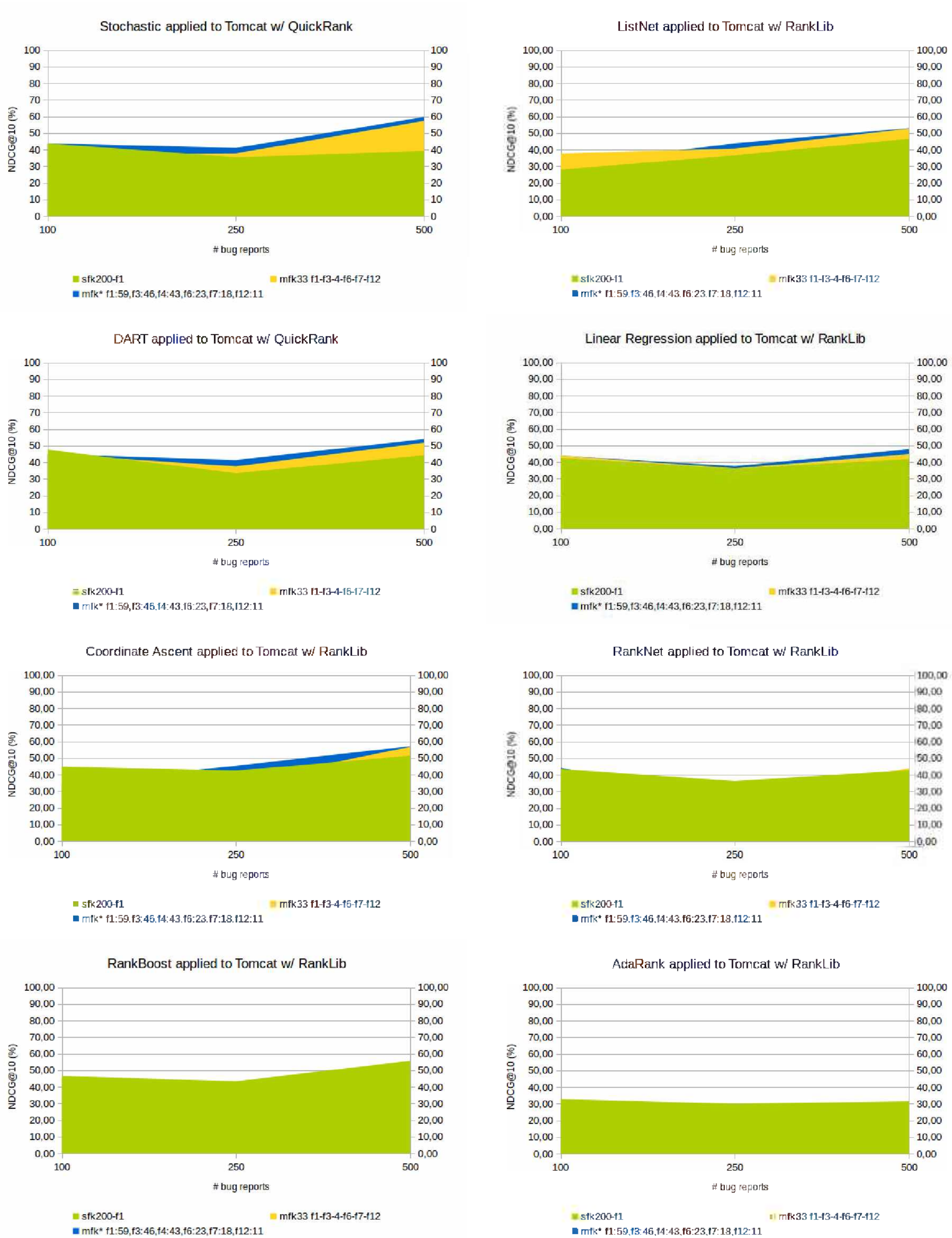


Figure 23 – NDCG@10 for Stochastic, ListNet, DART, Linear Regression, Coordinate Ascent, RankNet, RankBoost, AdaRank from QuickRank and RankLib tools).

to avoid extending the experiments. We proceed with a grid search-based approach to produce results and find the best combinations.

Table 10 summarizes some performance values and statistics from the experiment with LambdaMART. The section *Best Performance* in that table shows the best values obtained for the settings with baseline (LR-All) and entropy (LR-ALL+$\phi_{20.1}$), for both metrics MAP and NDCG@10. The following two sections present the *Average Performance* and *Standard Deviation* considering: 1. only the five settings with the best performance (5-Best); 2. the ten settings with the best performance (10-Best); 3. all the settings. Finally, the last sections show the direct comparison between performance obtained for baseline and entropy settings, considering the best setting and the averages shown above in the table. For MAP, baseline overcomes entropy by 2.7% in the best setting, while for NDCG@10, there is a draw since the performances are equal. However, the differences are minimal and below 1% for the 5-Best settings. In the 10-Best settings, the performance is almost the same for MAP, while the entropy setting overcomes baseline in 2.21%. In the All Settings comparison, entropy presents better performance overall than baseline (14.95% better with MAP, and 42.47% better with NDCG@10). The standard deviation perspective also confirms these results. The low values for the 5-Best and 10-Best indicate that top settings have similar performance. However, All Settings has a far higher standard deviation, confirming a big difference between best and worst settings. In almost all the cases, the standard deviation in entropy settings is below the baseline. Since the average performance results of entropy settings are almost the same or better than baseline, the tendency is that settings with entropy deliver better results overall than baseline.

Table 11 shows the best parameters found in the experiments and also the statistical Mode for the five best and the ten best performance settings. There is no unanimity with MAP measure for the Shrinkage parameter since all the parameter values appear in the top settings. Considering the best setting and the 10-Best-Mode with LR–All settings, 0.05 would be the choice. Considering LR–All+$\phi_{20.1}$ again, it is not easy to point the winner. With NDCG@10 measures, there is no doubt about the choice for the Shrinkage of 0.05, either for LR–All or LR–All+$\phi_{20.1}$. For the Number of Trees parameter and with MAP measure, most of the settings have 32 trees, except in the best setting of LR–All with 64 trees. For NDCG@10 measures, most settings have 64 trees, including the best settings, but 32 and 128 values also appear in the 5-Best-Mode and 10-Best-Mode. Finally, the Number of leaves has the value 1 present in most settings, either for MAP or for NDCG@10. Some exceptions occur for the best setting of baseline LR–ALL with MAP, while some mode settings where the value 2 also appears. The following charts

Best Performance						
	MAP			NDCG@10		
LR-All	42.29			47.66		
LR-All+$\phi_{20.1}$	41.18			47.66		
Average Performance						
	MAP			NDCG@10		
	5-Best	10-Best	All Set.	5-Best	10-Best	All Set.
LR-All	41.33	40.77	19.55	47.33	45.42	15.47
LR-All+$\phi_{20.1}$	41.04	40.76	22.99	47.56	46.44	26.90
Standard Deviation						
	MAP			NDCG@10		
	5-Best	10-Best	All Set.	5-Best	10-Best	All Set.
LR-All	0.54	0.87	17.68	0.51	2.92	15.80
LR-All+$\phi_{20.1}$	0.15	0.35	15.22	0.11	1.36	18.85
How much entropy setting is better than baseline?						
	Best Set.	5-Best (Avg)	10-Best (Avg)	All Set. (Avg)		
MAP	-2.7 %	-0.72 %	-0.04 %	14.95 %		
NDCG@10	0.00 %	0.5 %	2.21 %	42.47 %		

Table 10 – Performance statistics from the tuning of LambdaMART on SWT using baseline and $\phi_{20.1}$ entropy settings.

show values 1 and 2 for leaves have almost the same performance overall, making the value of 1 the best choice for the Number of Leaves.

To finalize the tuning analysis, the charts in Figures 24 and 25 show how the performance variate with the parameters. We restrict the number of charts to facilitate the analysis and insights since many charts were generated considering all the metrics and parameters used in the experiments and would bloat the text unnecessarily. The bars in blue represent performance data for baseline testing data, while the light blue bars represent the baseline testing data. Similarly, the bars in red represent the performance of entropy testing data, while the light red bars represent the entropy testing data. The parameters shown in the legend define the ordering of the data associated with each main group of bars, where nt is the number of trees, and sh is the Shrinkage. To illustrate, in the Figure 24-a, there are three groups of bars, one for each shrinkage value in the horizontal axis (0.05, 0.5, and 0.8). In the first group (sh=0.05), there are 20 bars. The first four bars are associated with the settings with 32 trees: 1st bar is the training performance of the baseline setting (light blue), 2nd bar is the testing performance of the baseline setting (blue), 3rd bar is the training performance of the entropy setting (light red), 4th bar is the testing performance of the entropy setting (red). The next four bars are associated with the settings with 64 trees and follow the same previously

	Parameter	Best	5-Best-Mode	10-Best-Mode
MAP				
LR-All	Shrinkage	0.05	{0.5, 0.8}	0.05
	Number of Trees	64	32	32
	Number of Leaves	5	{1, 2}	1
LR-All+$\phi_{20.1}$	Shrinkage	0.8	{0.5, 0.8}	0.05
	Number of Trees	32	32	32
	Number of Leaves	1	1	{1, 2}
NDCG@10				
LR-All	Shrinkage	0.05	0.05	0.05
	Number of Trees	64	{64, 128}	{32, 64}
	Number of Leaves	1	1	{1, 2}
LR-All+$\phi_{20.1}$	Shrinkage	0.05	0.05	0.05
	Number of Trees	64	64	{32, 64}
	Number of Leaves	1	{1, 2}	1

Table 11 – Parameters found in the best performance settings while tuning LambdaMART on SWT bug reports.

described sequence. This logic repeats for the settings with 128, 256, and 512 trees, completing the first group of bars. This sequence repeats for Shrinkage of 0.50 and 0.80. The Figure 24-b is derived from the Figure 24-a, by removing all the bars of training performance, remaining only test data for baseline (blue) and entropy (red) settings. The vertical axis was also limited to 50. This chart facilitates the direct comparison of the baseline with the entropy testing performance settings.

In Figure 24 it is possible to perceive the higher values for both baseline (blue) and entropy (red) settings, when the Shrinkage value is 0.05, and also for 0.80, although the former has a little bit higher values overall. It is also visible that the performance falls with increasing the number of trees. The performance falls are even more severe for the Shrinkage of 0.50, especially from 128 trees and above. The bars in light blue (baseline training settings) and light red (entropy training settings) represent the training performance and highlight when the over-fitting occurs. For example, the differences between training and testing data for the baseline are the largest (the number of trees greater or equal to 128). We can confirm some over-fitting in settings with more trees, mainly with a Shrinkage of 0.50 value. Figure 24-c and 24-d are similar to the previous, but considering the NDCG@10 measure. The differences between baseline and entropy settings are more accentuated, favoring testing settings with entropy. In this case,

especially in Figure 24-c the over-fitting on baseline settings is clearly more accentuated with shrinkage of 0.50 and 0.80. Nevertheless, this occurs only for baseline settings since it seems the over-fitting is not severe for entropy settings. Best values are clearly in the first bar group, where Shrinkage is 0.05.

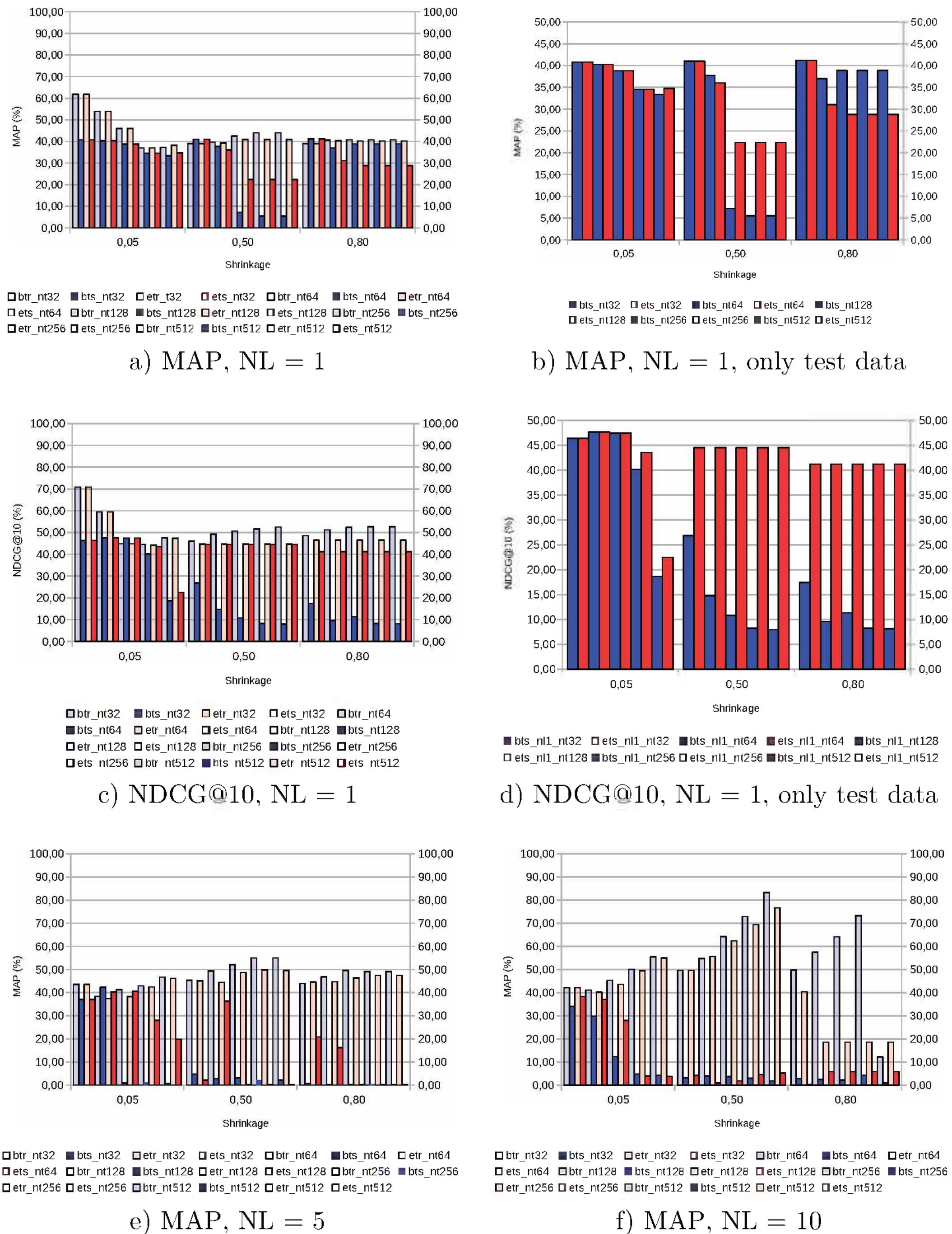
Figure 24-e and Figure 24-f show the performance for the $nl = 5$ and $nl = 10$. The performance for $nl = 2$, has no meaningful difference from $nl = 1$, and was ignored. The over-fitting with the increase in the number of leaves is clear, even for the settings with previous good performance such as sh = 0.05, and mainly win baseline setting with nt above 64. The higher over-fitting in the settings occurs with $nl = 10$. Figure 25 show same but now with the bar groups organized according to the Number of Trees parameter. The ordering inside each group follows the shrinkage parameters (0.05, 0.5, 0.8). These charts reinforce the insight that the overfitting increases with the number of leaves and with the number of trees.

Even considering the Entropy feature does not improve the performance of the top settings in general meaningfully compared to baseline settings, after the exposed results, it is more or less clear that this feature seems to play some role in reducing over-fitting for the settings overall. The algorithm tuning is also evident, and it is crucial to obtain the best from the applied algorithm. To reinforce this need, after running LambdaMART using the default parameters (#trees = 1000, #leaves = 10, shrinkage = 0.1), we just get over-fitted and poor results: for LR-All, training data gives 71.09%, and testing gives %5.76 for MAP, while with NDCG@10, training gives 88.02% and testing gives 2.54%; for LR-All+$\phi_{20.1}$, training gives 72.97%, and testings give 3.9% for MAP, while with NDCG@10, training gives 87.28% and testing gives 2.02%.

5.2.5 RQ5: How long does it take to conclude each step in the process (feature extraction, ranking generation, training, validation, and testing)?

Figure 26 show the average computing time distribution to generate the rankings and to extract the performance metrics for the individual features. These rankings are the primary input data to produce the input SVM-Light files for the training and testing through the LtR approaches. The process steps groups in:

Pre-processing: related to loading bug report information from the text files, source code file parsing, index generation, and persistence on the database (e.g., class,



a) MAP, NL = 1

b) MAP, NL = 1, only test data

c) NDCG@10, NL = 1

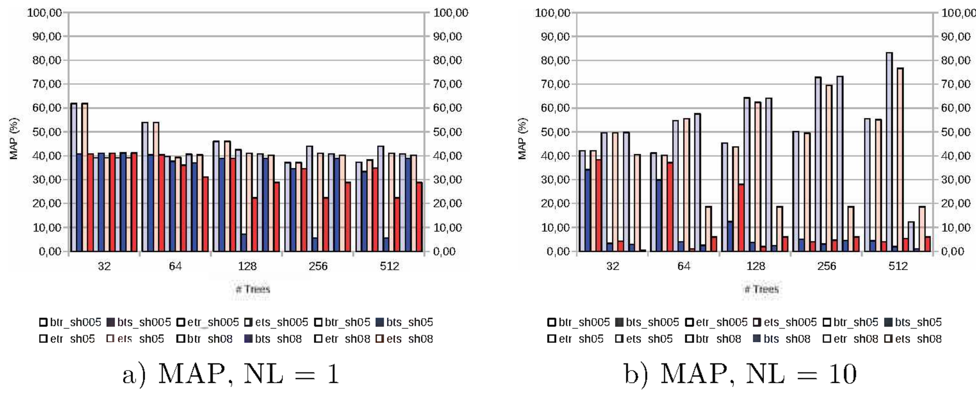
d) NDCG@10, NL = 1, only test data

e) MAP, NL = 5

f) MAP, NL = 10

Figure 24 – Tuning LambdaMART (RankLib) on 300 bug reports of SWT: MAP and NDCG@10 performance changing Shrinkage {0.05, 0.5, 0.8} and Number of Leaves (NL) {1, 5, 10}.

method, and many other entities found in source code files). Some of these steps are pre-requisites shared by some features.



a) MAP, NL = 1 b) MAP, NL = 10

Figure 25 – Tuning LambdaMART on SWT (RankLib): MAP performance changing Number of Trees (NT) from 32 doubling until 512 and Number of Leaves (NL) = {1, 10}.

F1..F19: this includes the time spent on the feature extraction and on the computing of the individual rankings for each feature found in the baseline work of (YE; BUNESCU; LIU, 2016).

F20.1..F22.1: this is the time spent in entropy feature extraction, including language model generation and the ranking generation for the features $\phi_{20.1}$, $\phi_{21.1}$, and $\phi_{22.1}$.

Normalization: includes the time spent to access all the individual rankings and produce the corresponding normalized ranking.

Metrics: includes the time spent to produce the performance results for all the individual rankings, based on the metrics MAP, MRR, Top-N

From the Figure 26b and 26c, it is clear the main bottleneck is in the entropy extraction that consumes 56% to 62% of the time in each bug report processing. The computing time is worrying, especially because it relates to only three entropy features. In Figure 26c, without the entropy feature computing, the total time to complete the process is much lower, 4.13 minutes, while the previous with entropy requires 9.99 and 11.11 minutes. Considering the 300 bug reports used in the experiments with SWT, the average estimated total time to produce the necessary input data for the LtR approaches is around 55.5 hours. A total of 768.63 hours (or around 32 days) is estimated to cover all the bug reports only for the SWT project (4151 bug reports), including entropy extraction (only three of them). Thus, it is essential to optimize the current implementation so the experimentation and covering of all the 22,747 bugs in the LR-dataset become viable. Even discarding the entropy feature and taking the reduced time

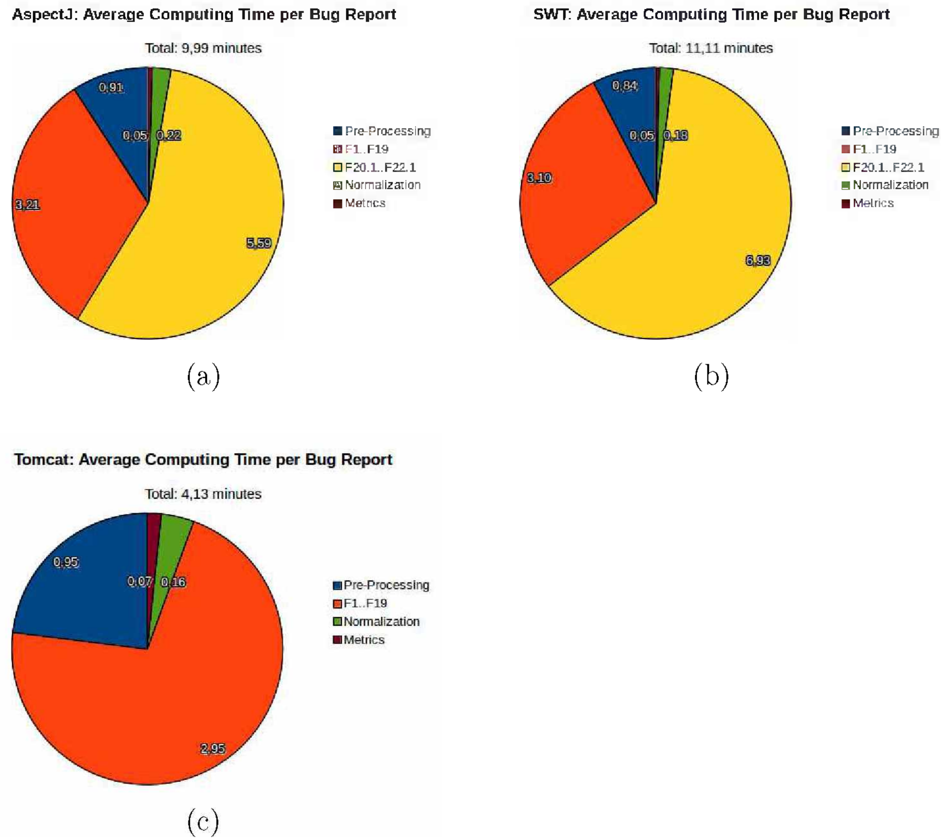


Figure 26 – Average computing time distribution per bug report for AspectJ, SWT, and Tomcat.

of 4.13 minutes per bug report in Tomcat would require 65 days to cover all the bug reports in this dataset (and we still need to consider time for SVM-light files generation, training, validation, and testing).

For these exploratory experiments, we do not measure the time spent on the generation of SVM-Light files and the training and testing with LtR algorithms. We do not compute the time spent in the checkout of each source code repository since it only depends on other factors not directly related to the framework implementation (e.g., network speed and project size). Since the SVMRank, RankLib, and QuickRank were employed in the experiments, running many instances, each with a different setting, and in parallel to complete the experiments faster, we opted not to include the measures information about training and testing resources consumption in this phase. From our informal perception from the experiments, the time spent in training and testing is by far lower than the time to produce the input data and lower than the time spent in the generation of SVM-Light files necessary to the learning process. Thus, the critical point

to reduce the time spent on the process overall is refactoring and optimizing the feature extraction process, followed by the index generation. This way, the BL solution would be more scalable.

5.3 Final Considerations

In these exploratory studies, we tested our experimental package against many possible settings and influential factors for LtR-based BL strategies applied to some samples of the LR-dataset. As in Ye, Bunescu e Liu (2014) original work, we confirm ϕ_1 as the most influential individual feature and also the more substantial influence of the query-dependent features. On the other hand, our tested non-query-dependent Entropy feature does not produce such impactful results. It also imposes high demands on computational resources, requiring additional optimizations for a viable and scalable application, even to proceed with new tests on more samples from the LR-dataset. These results also extend with the application of the Entropy feature to compose the ML training sets, and again, we did not get such promising results. The experimentation with two alternative data balancing strategies, (MF_{K33F6} and MF_{k200w}), combined with various ML-based algorithms from QuickLearn and RankLib tools, show more noticeable results over the baseline (using MF_{k200f1} data balancing with SVMRank LtR algorithm). The best performance increase was 55.72% in Rankboost with NDCG@10. Additionally, these data balancing strategies show some role in overfitting avoidance. While experimenting with tuning parameters for each LtR algorithm, we observe the importance of good tuning so we can obtain the best performance results. From the worst to the best performance settings, we observed significant differences, reaching a standard deviation around 15 and 18 points in MAP and NDCG@10 measures, respectively. There is no evidence that the Entropy feature contributes substantially to improving performance between the best-tunned settings. Finally, we have shown the long time demanded to conclude the experiments and process each bug report (from 4 to 11 minutes on average). The demand is aggravated with the Entropy feature introduction and asks for the implementation optimization to make the experimentation package scalable and able to process larger samples and cover all data in LR-dataset.

Chapter 6

Analysis of repair actions and patterns

In Chapter 4, many influential factors for BL were raised. An experimental package designed to test many of these ideas was also briefly introduced, and Chapter 5 shows preliminary experiments. Since one of our more interesting and relevant work was about the study of the bug characteristics in a dataset (SOBREIRA et al., 2018; MADEIRAL et al., 2018), the natural sequence is to apply the experience gained in this area to the BL problem and to include support in the experimental package for evaluations and comparisons. Nonetheless, before proceeding with the experimental tasks, we discuss bug patches characteristics in this chapter, extending our initial work with Defects4J Dissection to another bug dataset. Here we confirm that some of our findings in Defects4J are still valid for a larger bug dataset, the LR–dataset (YE; BUNESCU; LIU, 2014), usually applied to evaluate BL approaches, as occurs in our baseline implementation for the experimental package.

6.1 The role of patches on Bug Localization

A software bug can cause a software fail or misbehavior. To solve this situation, debugging is necessary, and the bug removal process requires the application of a bug patch to the software codebase. The bug patch comprises all the required changes, so the expected system behavior is reestablished. These changes can involve the addition, removal, or modifications of single or many lines of source code.

Since version control systems (like Git) are already part of the software best practices, it is possible to access the buggy easily and fixed versions of a codebase. A usual way to extract a patch is through the difference between two versions of a system, and a common approach is to access the version before the patch application (the buggy version) and

the patched version of the system (the fixed version). Of course, many types of bugs can exist, included bugs non-related to source code, but for this work, we are interested in bugs related to and fixed by source code changes. In this context, we can think of bug patches to validate and identify the bug location for the already fixed bugs.

6.2 Understanding the nature of the bugs through their patches

While we can easily perceive the existence of a bug in a system through problems, fails, or software misbehavior caused by the bug, usually, the bug's delimitation and characterization are not clearly defined. Many types of bugs exist, but what exactly are these bugs? Since the bug is unknown until we find the root cause of the problem and a way to fix it, a possible alternative to characterize bugs is to look for and study the many ways to fix them. So the study of the nature of the bug patches would indirectly inform us about the nature of their associated bugs. Many patches can apply for the fixing of a given bug. However, the multiplicity of solutions for the same bug is not our study target. Instead, we intend to study the regularity and the reuse of particular solutions to fix the bugs, observing patterns or other recurrent structures and situations. We observe that many of these recurrences in patches are common even between different projects and with similar frequency distribution.

6.3 Analysis dimensions of a bug patch for Bug Localization

In our previous work (SOBREIRA et al., 2018) we show many types of bugs present in Defects4J, now a frequently applied bug dataset. We analyzed many dimensions of Defects4J, involving the patch size, the patch spreading, and composition, expressed in terms of repair actions and repair patterns. We also defined a taxonomy to refer to these repair actions and patterns. While the work on Defects4J was valid to identify the presence of many patterns and the reuse of similar solutions, even between different projects, the dataset size (only 395 bugs) limits its applicability, especially while evaluating approaches that depend on a more significant number of samples, such as those employing machine learning techniques. To overcome this problem, we decided

to evaluate the LR–dataset applied by Ye, Bunescu e Liu (2014) using the dimensions defined in the Defects4J dissection analysis.

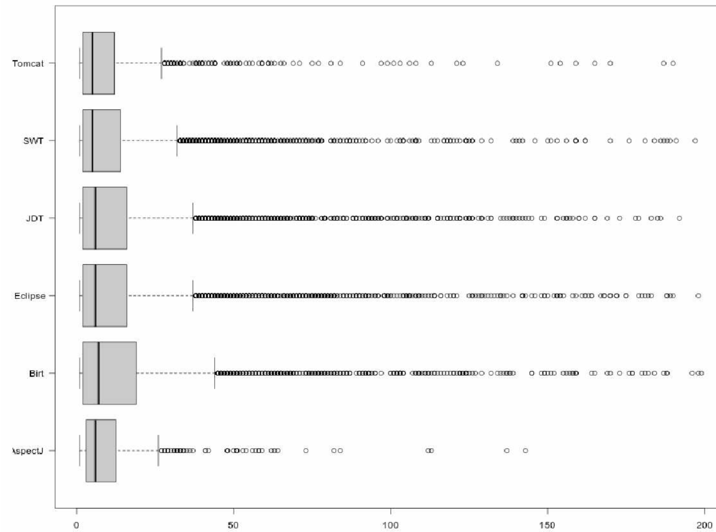
6.3.1 Size dimensions

From the original 22,747 bug reports in projects of LR–dataset, ADD tool (MADEIRAL et al., 2018) can extract information from 21,177 bug patches. Some of the excluded patches would present problems during the processing, e.g., not generating an AST. Still, there are many outliers between these 21,177 bug patches and, without a restriction to filter out these outliers, it is difficult to show the overall size distribution of the patches. Limiting the number of lines to 200 (Figure 27), we exclude 113 outliers, but we still have many of them (represented by circles beyond the whiskers limit in the boxplot). With 60 lines limit, we exclude 1039 outliers (4.91% reduction), and we have a patch size very close to the maximum patch size found in Defects4J (54 lines). In this new setting, we can see that most patches range between 1 and around 35 lines for all the projects. In addition, the median size ranges from 5 to 7 lines. Except for the outliers, these results are very close to what we have found in Defects4J since the size ranges from 1 to 54 code lines (max), and the median patch size was 4 lines.

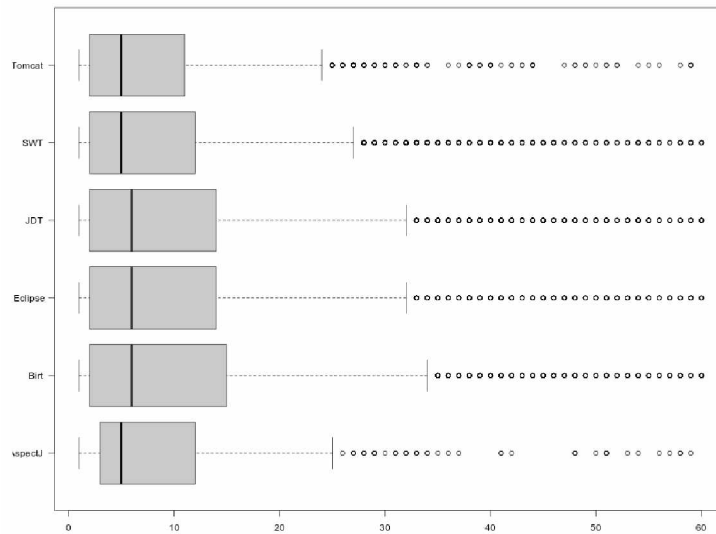
Figure 28 shows the added lines, removed lines, and modified lines of the patches from all the projects in LR–dataset, maintaining the same limit of 60 code lines for the patch size. Since the distribution of added lines is closer to the total patch size distribution, this confirms the same tendency found in Defects4J, where patches composition contains more lines added than lines modified and removed.

Patches can combine any code lines type: added, modified, and removed. Figure 29 shows a) the number of patches for each combination overall; b) the same info but only for patches with a maximum of 60 lines. Comparing the diagrams, we can observe that most of the excluded outliers were in the set with patches containing all the types of line changes, and the reduction was 781 patches (75.17% of the 1039 removed outliers). It is reasonable since we can expect that huge patches can include more types of line changes than small ones.

Next diagrams show the previous information split between each project (Figure 30). The proportion of patches in each set seems to be very similar to the overall distribution. When compared to Defects4J, these proportions are also very close.



a) maximum patch size of 200 lines (20,164 patches).

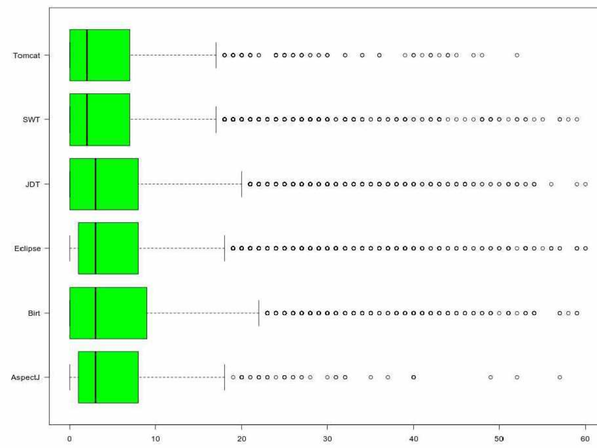


b) maximum patch size of 60 lines (20,138 patches).

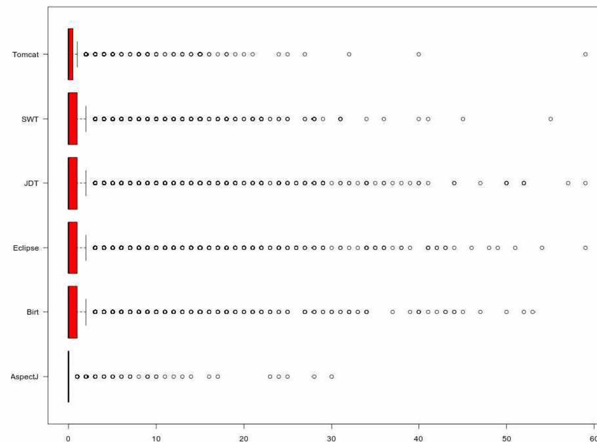
Figure 27 – Distribution of the number of lines of the patches limited to a maximum size.

6.3.2 Spreading

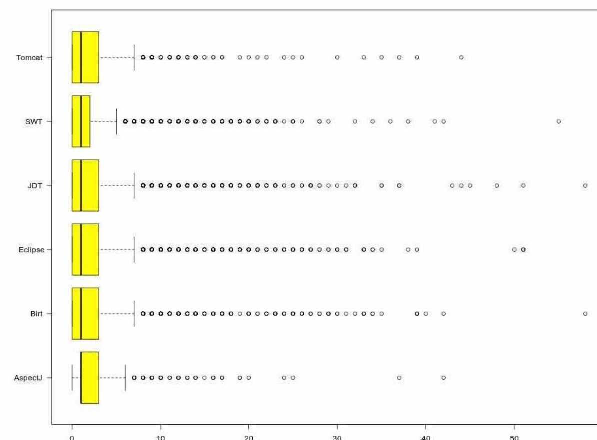
We can define the patch spreading from many perspectives. First, patch spreading allows understanding how much the bug fixing is concentrated or spread through the codebase. Furthermore, the patch spreading can significantly impact the bug localization since a strategy that performs well for a small and single block of code patch would have an inferior performance trying to localize a bug with the patch spread on many files (or



a) code lines added.



b) code lines removed.



c) code lines modified.

Figure 28 – Distribution of the number of code lines by type in 20,138 patches of LR–dataset.

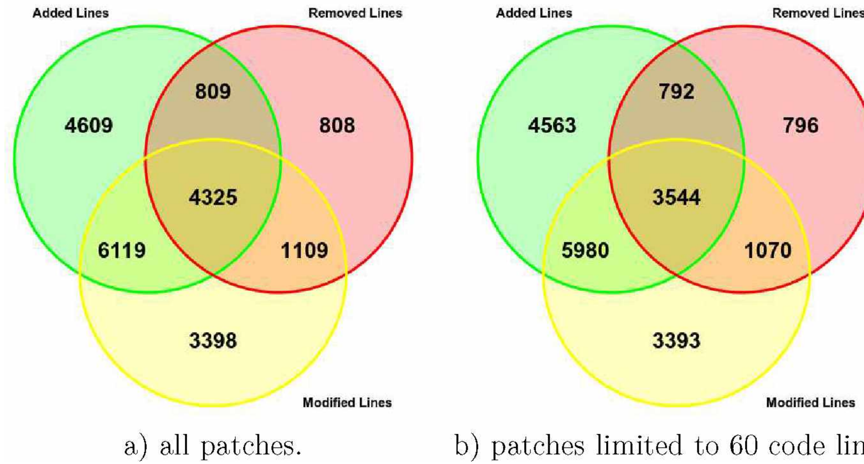


Figure 29 – Overall patches distribution according to the type of code lines affected.

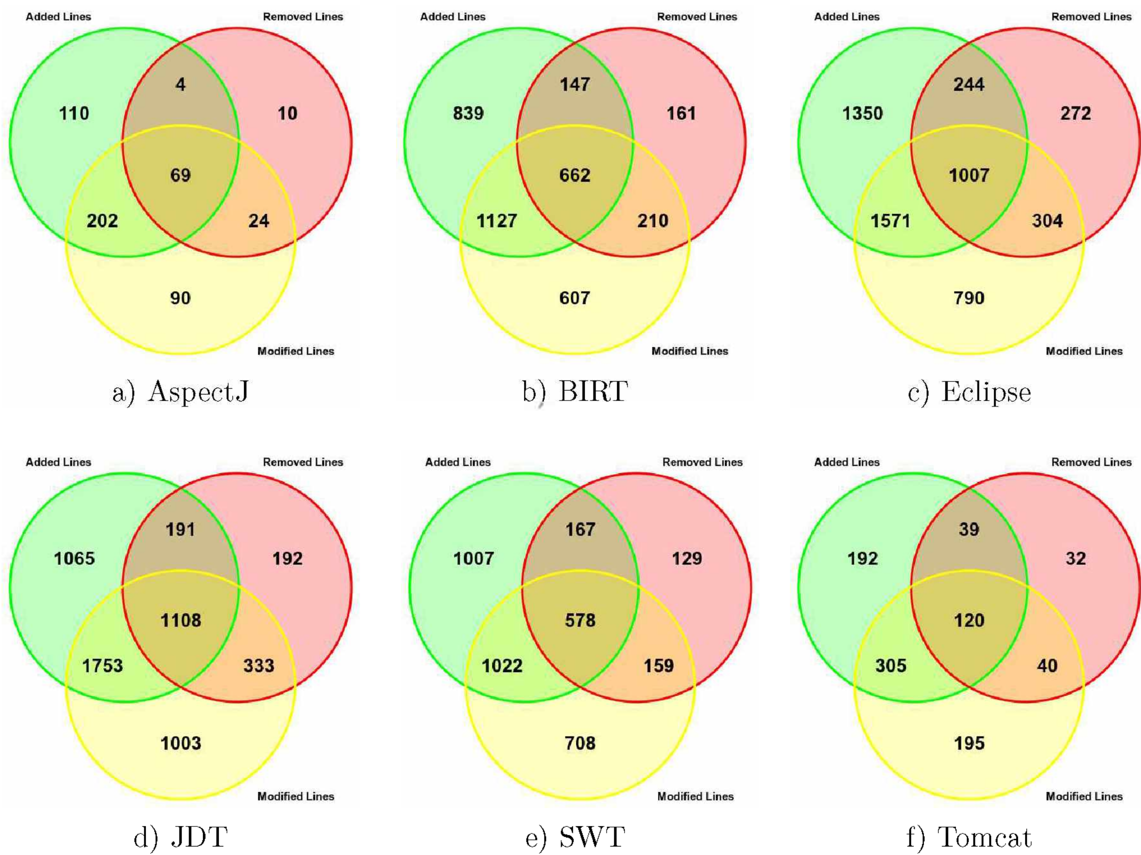


Figure 30 – Patches according to the type of code lines affected in each project.

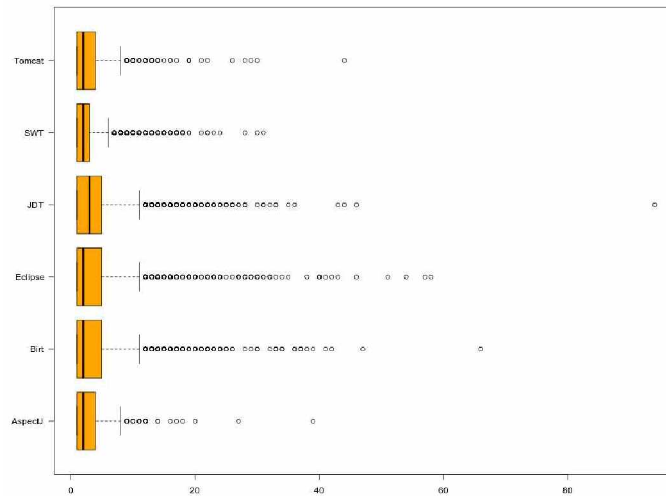
even a single file patch but sparse in many lines). We show in this section the spreading profile for the (YE; BUNESCU; LIU, 2014) dataset, based on the spreading measures applied in (SOBREIRA et al., 2018): number of chunks, spreading of the chunks, number

of modified classes, number of modified methods, and number of modified files.

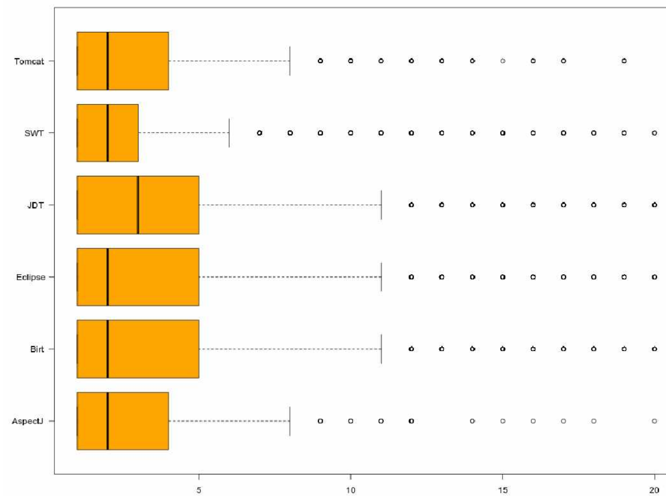
As occurs for the patch size, outliers also impact and contribute to extending the range for the number of chunks beyond 200. After the application of the same patch size limited to 60 lines (so we can maintain the same bug set analyzed before), we obtain the distribution shown in the first chart of Figure 31, and for these patches, the number of chunks reduces somehow (now the maximum is 94). From this distribution, we see that most of the patches have less than ten chunks, and the instances beyond that would be considered outliers. The second chart in Figure 31 is an alternative to show the patches distribution removing patches where the number of chunks is beyond the limit for outliers. The limit applies to the number of chunks of the patches, excluding patches with more than 20 chunks (applying this filtering criterion, we still maintain 20,688 patches). The median number of chunks is close between the projects (around 2), except for the JDT (around 3). For all the cases, 75% or more patches in each project have a maximum of five chunks. Compared to Defects4J, LR–dataset has a slightly higher number of chunks per patch (90% have eight or fewer chunks), but it is still very close since 90% of the patches in Defects4J have at most five chunks.

The spreading of the patches measures the accumulated number of code lines between their chunks and is in the first chart of Figure 32. The spreading of the patches gives a clue about the patch dispersion in the codebase. Again, there are many outliers, even considering the patch size reduction, limited to 60 lines. To get a better perception about the distribution, in the second chart of Figure 32 the spreading (and not the patch size) is limited to 350 lines. In this new chart, the distribution is more evident, and, in most projects, the first half of the patches have no spreading at all, or the spreading is below 25 lines. While patch size has a more uniform distribution between the projects, the spreading variability is higher. For example, while in SWT, the patches seem to be more concentrated (half has no spreading, and 75% have a spreading below 25 lines, the threshold for outliers is around 60 lines), in JDT, 75% have a spreading that can achieve 100 lines, and their threshold for outliers is the highest (more than 200 lines).

Like Defects4J, most parts of the LR–dataset bug patches concentrate in a few chunks. There is both type of patches: those concentrated in single lines (or even single blocks) of code, significantly below the first half in the distribution; and those patches dispersed in more blocks of code that can be very close to each other (just a few lines distance) or with distances of hundreds of lines, especially in the upper half of the distribution.



a) maximum of 60 code lines (20,138 patches).

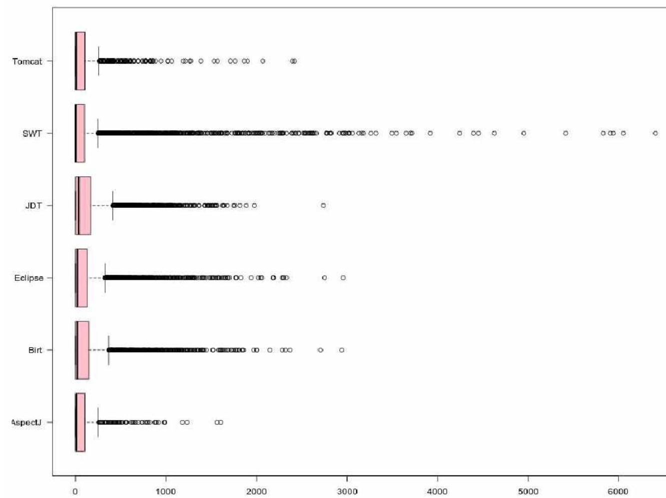


b) maximum 20 chunks (20,688 patches).

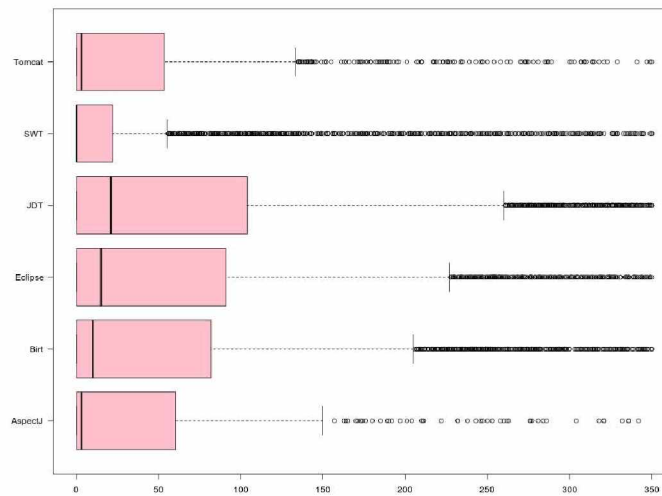
Figure 31 – Distribution of the number of chunks of the patches.

6.3.3 Size and Spreading Dimensions' Statistics

Table 12 summarizes the patch size and spread information. These statistics consider the 21,177 patches, without restrictions to patch size. Table 13 shows the same information but only for the patches with a maximum size of 60 lines. The tables also summarize modified files, classes, and methods associated to the patches. One more time, the distribution is close to Defects4J, and most of the patches are associated with a few files, classes, and methods. Additionally, we can see that even with filtering some of the outliers using the 60 lines size restriction, this condition is not enough to remove them. One example is the patch that affects 328 files (the maximum value observed in Table 13).



a) patches with maximum size of 60 lines.



b) maximum spreading of 350 code lines.

Figure 32 – Distribution of chunks spreading of the patches.

Even considering that it is possible to have patches associated with that amount of files since the 60 lines size restriction does not account for lines related to comments and blank lines (e.g., white spaces), it is clear that additional filtering would be necessary to avoid noise and these type of outliers, especially in experimental settings. Therefore, the application of maximum number for chunks, spreading, files, classes, and methods would be required to remove other outliers and maintain a representative dataset. For example, 99% of the patches affect a maximum of 21 files (20.54), and considering the 60 lines restriction, the number of affected files drops down to 19 files (far below the 328 files previously cited). Considering the exposed rationale, a threshold of 20 files would

Table 12 – Descriptive statistics for 21,177 bug patches.

	Min	25%	50%	75%	90%	95%	Max
# Added lines	0	0	3	9	22	36	1,088
# Removed lines	0	0	0	1	8	16	1,298
# Modified lines	0	0	1	3	8	14	4,647
Patch size	1	2	6	16	37	60	4,652
# Chunks	1	1	2	5	9	14	215
Spreading	0	0	21	157.0	457.4	747.2	6,407.0
# Files	1	1	1	1	2	7	383
# Classes	1	1	1	1	1	2	14
# Methods	0	1	1	2	4	5	76

Table 13 – Descriptive statistics for 20,138 bug patches, without outlier patches (more than 60 lines).

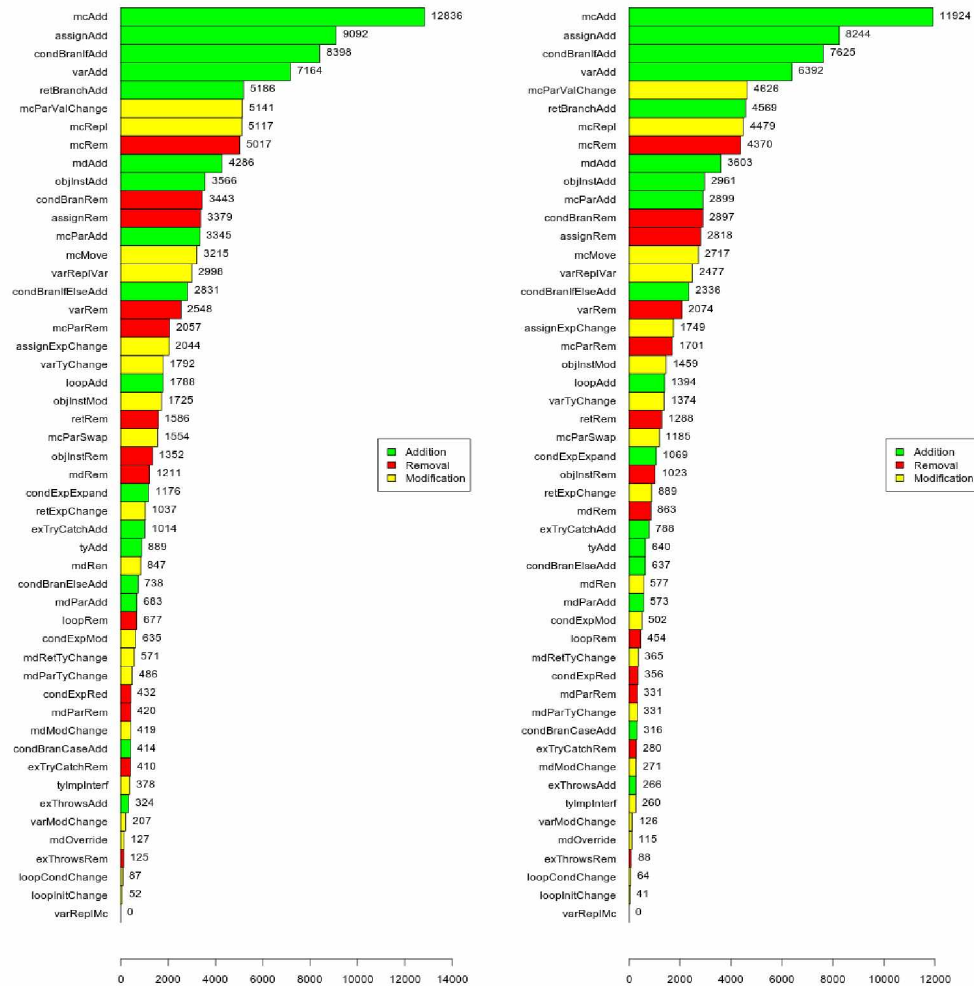
	Min	25%	50%	75%	90%	95%	Max
# Added lines	0	0	3	8	17	25	60
# Removed lines	0	0	0	1	5	10	59
# Modified lines	0	0	1	3	7	10	58
Patch size	1	2	6	14	27	38	60
# Chunks	1	1	2	4	8	10	94
Spreading	0	0	16	137	413	674	6,407
# Files	1	1	1	1	2	7	328
# Classes	1	1	1	1	1	2	14
# Methods	0	1	1	2	3	4	49

be a good complement to the outlier filtering schema.

6.3.4 Repair actions

A *repair action* is a basic syntactic building block composing the patch for a bug. We have defined a taxonomy to refer to the repair actions found in Defects4J (SOBREIRA et al., 2018). Here we apply the same taxonomy, and we conduct the detection of these repair actions with our tool, ADD, partially presented by Madeiral et al. (2018). To facilitate the understanding of the acronyms, Table 14 shows its correspondence (more details in (SOBREIRA et al., 2018). The first chart of Figure 33 shows the incidence of all repair actions from LR–dataset (YE; BUNESCU; LIU, 2014), while the second chart shows the same info without some of the outliers (patches with a size beyond 60

lines). As shown in these charts, removing the outliers is not so impacting, and most repair actions positions are maintained. The dominance of repair actions involving code addition is clear, especially in the top positions. The higher presence of repair actions more associated with code addition helps explain why there is more code added than modified and removed in the patches (shown in the previous sections). As occurs in Defects4J, the top actions are related to *Method Call Addition*, *Assignment Addition*, and *Conditional Branch Addition*.



a) overall (21,177 patches). b) patches with maximum size of 60 code lines (20,138 patches).

Figure 33 – Repair actions found in the LR–dataset.

To facilitate the comparison with Defects4J, Figure 34 shows the repair actions grouped according to Table 15, and in the same format of the charts in (SOBREIRA et al., 2018). In the middle column we have: A = Addition action; R = Removal ac-

Table 14 – Repair actions acronyms and full names.

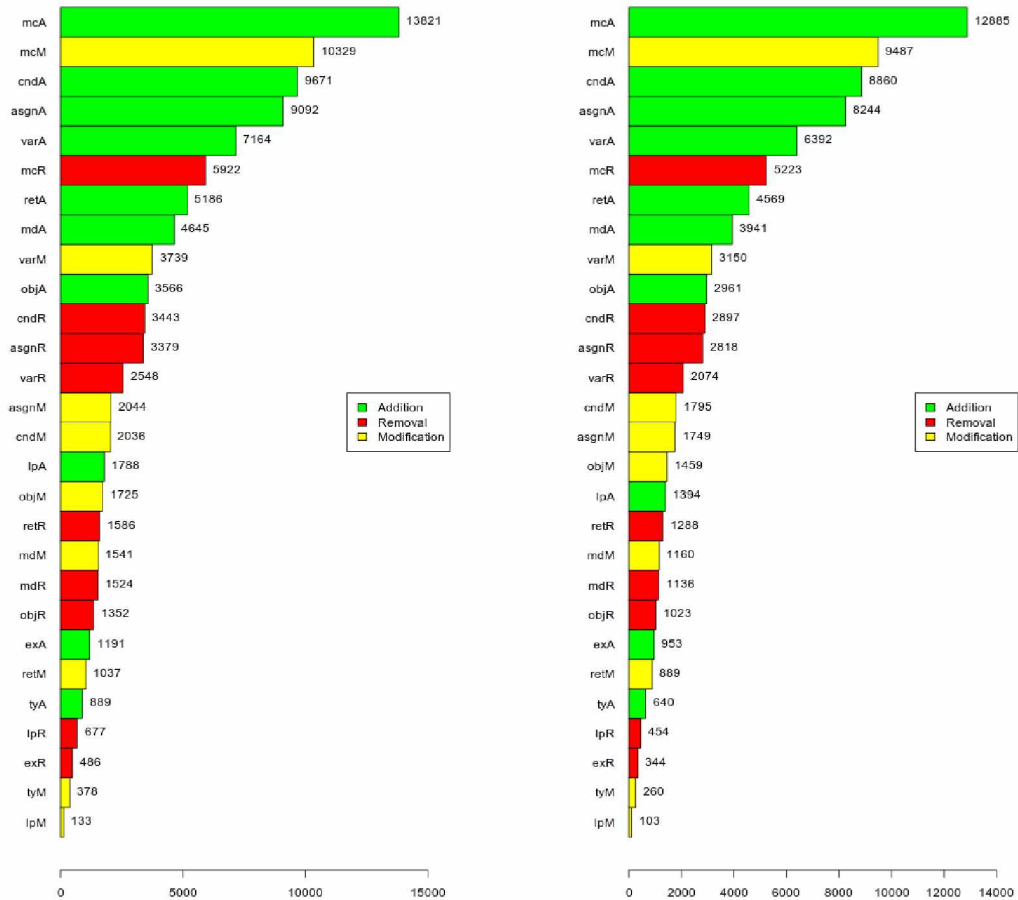
Acronym	Repair Action
assignAdd	Assignment addition
assignExpChange	" expression modification
assignRem	" removal
condBranCaseAdd	Conditional (case in switch) branch addition
condBranElseAdd	" (else) branch addition
condBranIfAdd	" (if) branch addition
condBranIfElseAdd	" (if-else) branches addition
condBranRem	" (if or else) branch removal
condExpExpand	" expression expansion
condExpMod	" " modification
condExpRed	" " reduction
exThrowsAdd	throw addition
exThrowsRem	" removal
exTryCatchAdd	try-catch addition
exTryCatchRem	" removal
loopAdd	Loop addition
loopCondChange	" conditional expression modification
loopInitChange	" initialization field modification
loopRem	" removal
mcAdd	Method call addition
mcMove	" " moving
mcParAdd	" " parameter addition
mcParRem	" " " removal
mcParSwap	" " " value swapping
mcParValChange	" " " value modification
mcRem	" " removal
mcRepl	" " replacement
mdAdd	Method definition addition
mdModChange	" " modifier change
mdOverride	" " overriding (addition or removal)
mdParAdd	" " parameter addition
mdParRem	" " " removal
mdParTyChange	" " " type modification
mdRem	" " removal
mdRen	" " renaming
mdRetTyChange	" " return type modification
objInstAdd	Object instantiation addition
objInstMod	" " modification
objInstRem	" " removal
retBranchAdd	Return statement addition
retExpChange	" expression modification
retRem	" statement removal
tyAdd	Type addition
tyImpInterf	Type implemented interface modification
varAdd	Variable addition
varModChange	" modifier change
varRem	" removal
varReplMc	" replacement by method call
varReplVar	" replacement by another variable
varTyChange	" type change

tion; M = Modification action. The distribution shape of the repair actions from the LR–dataset is very similar to the Defects4J distribution. The top repair action, *Method Call Addition*, is the same in both datasets. Eight actions from the top-10 in Defects4J distributions appear between top-10 positions in the LR–dataset. *Conditional Modification* and *Return Modification* actions in Defects4J top-10 are out of the LR–dataset top-10. *Method Definition Addition* and *Variable Modification* in LR–dataset top-10 are out of the top-10 in Defects4J. While both datasets are intrinsically composed of bugs from different Java projects, it is notable how close are the distributions of the repair actions of the patches. With the repair actions explicit, we understand the nature of the bugs and their associated patches. This kind of information would help to guide and allow more informed decisions while testing or evaluating certain approaches on these bug datasets. For example, we could expect an approach for bug localization (or even automatic program repair) would not perform well if it can not handle bugs requiring the addition of code, especially method calls, since this kind of action would be present in 12,885 from the 20,138 patches (64%). Another expected behavior is a poor performance (but not insignificant) of approaches guided by removal of code to fix a bug (the strategy of Kali approach (QI et al., 2015)) since patches containing removal actions are less prevalent.

Table 15 – Repair actions acronyms and grouping names.

Acronym	Action	Group
asgn	A/R/M	Assignment
cnd	A/R/M	Conditional
ex	A/R	Exception
lp	A/R/M	Loop
mc	A/R/M	Method Call
md	A/R/M	Method Definition
obj	A/R/M	Object Instantiation
ret	A/R/M	Return
ty	A/M	Type
var	A/R/M	Variable

The distribution of repair actions present in each project is in Figure 35 (all limited to patches with no more than 60 lines). Generally, the distributions between projects are close, even considering the significant differences in the number of patches in each project (from the 509 patches of AspectJ to the 5645 of JDT).



a) overall (21,177 patches).

b) patches with the maximum of 60 code lines (20,138 patches).

Figure 34 – Grouped repair actions found in LR–dataset.

6.3.5 Repair patterns

While the *Repair Actions* are basic building blocks found in patches composition, the *Repair Patterns* are more abstract structures found recurrently in the patches of Defects4J (SOBREIRA et al., 2018). In Defects4J Dissection, we have found 9 more general patterns (Figure 36), and some of them with variations, totaling 25 specific repair patterns (Figure 37). Table 16 summarizes the patterns acronyms and full names.

We apply the same taxonomy and descriptions of the patterns defined in our Defects4J Dissection study to analyze LR–dataset. Figure 38 shows a) the overall distribution of these patterns in LR–dataset and b) the distribution with a patch size limit of 60 lines.

Figure 39 shows a more compact view of these patterns, grouped according to Ta-

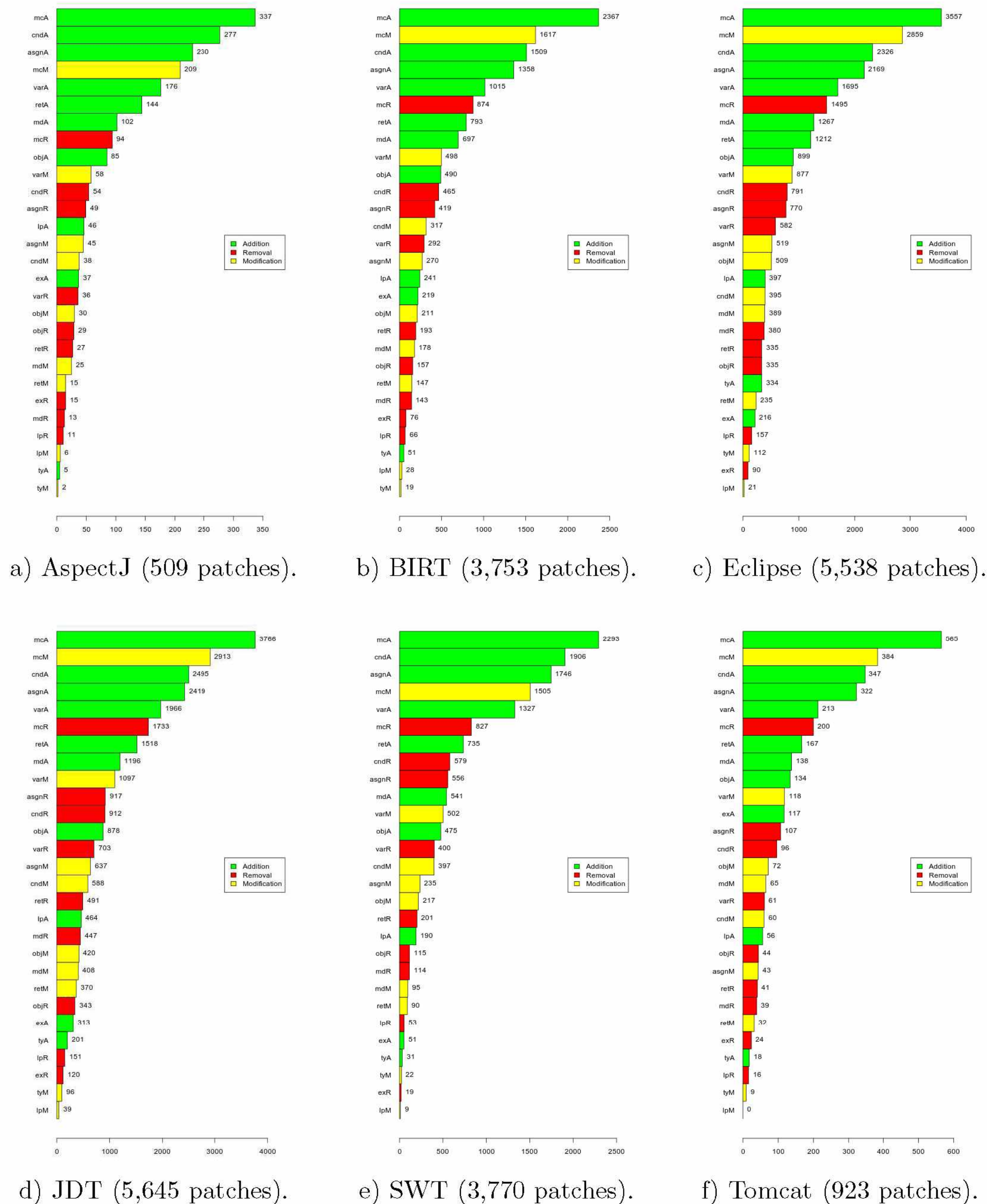


Figure 35 – Grouped repair actions incidence on each project.

ble 16 and allowing straight comparison to similar charts in (SOBREIRA et al., 2018). As in Defects4J, *Conditional Block* pattern continue as the top pattern found in the patches and also have a percentage of occurrences very close (around 42%). In the bottom, *Code Moving* and *Constant Change* continues as the less frequent patterns in

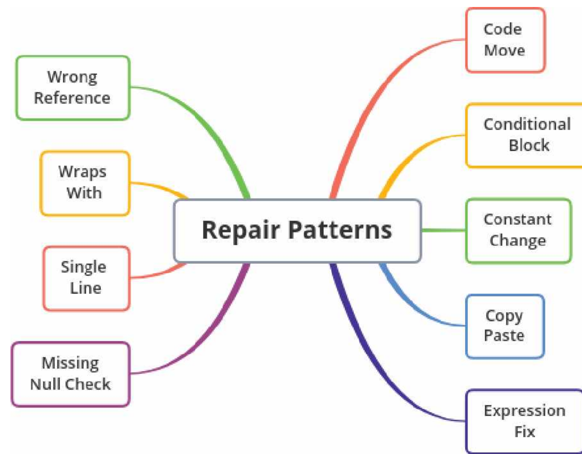


Figure 36 – Repair patterns found in Defects4j Dissection.

Repair Patterns with variations

	Conditional Block	Expression Fix	Missing Null Check	Wraps With	Wrong Reference
Pattern Variations	Add: Exception, Return, Others	Arithmetic: Modify	Null Check Add	Unwrap: If Else, Method, Try-Catch	Wrong Method
	Remove	Logic: Expand, Modify, Reduce	Not Null Check Add	Wrap: Else, If, If-Else, Loop, Method, Try-Catch	Wrong Variable

Figure 37 – Repair patterns with variations found in Defects4j Dissection.

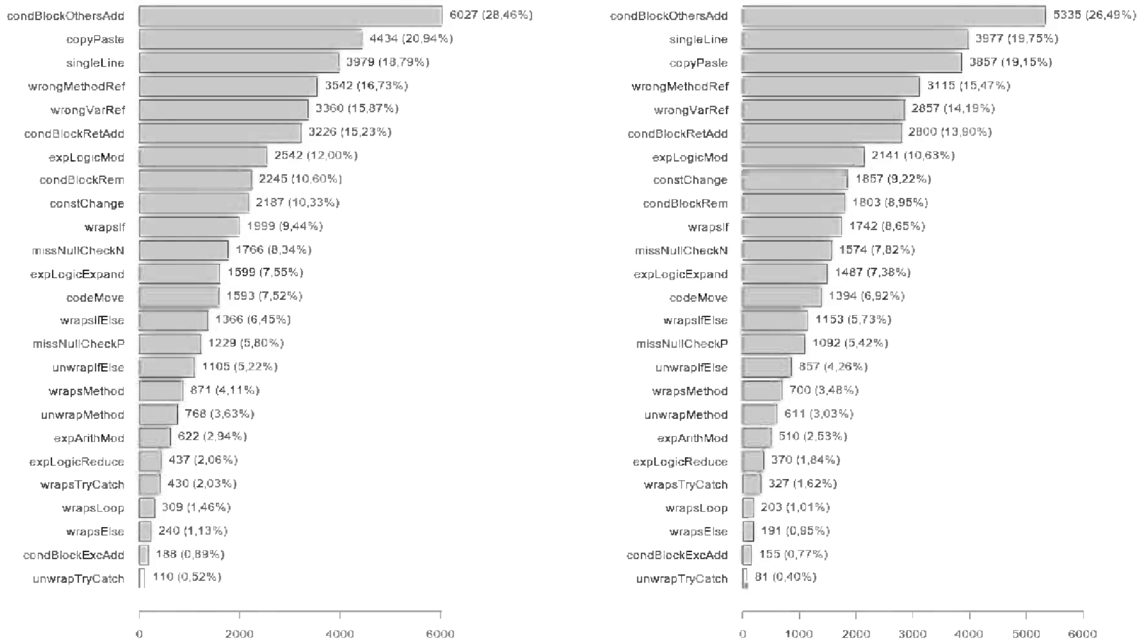
patches, but with higher percentages (around 7%, and 10% in LR–dataset, while Defects4J have 1.77%, and 4.81%, respectively). In the middle, we have some position changes and changes in the percentage prevalence. *Wraps With* did not change its third position and had an incidence of 22.83% in LR–dataset (-4.51% compared to Defects4J). *Single Line* also has a similar reduction (-5.06%) and appears in 19.75% of the patches in LR–dataset. *Expression Fix* have the higher percentage decreases (-13.15%) and appears in 19.75% of LR–dataset patches. *Null Check* appears in LR–dataset practically with the same percentage of Defects4J (around 12%). Two patterns increase its incidence: *Copy Paste* appears in 19.15% of the patches (+7%), and *Wrong Reference* appears in 26.74% of the patches (+9.02%) of LR–dataset. These findings reinforce the value and broader applicability of the patterns found in Defects4J since, in LR–dataset, we have a

Table 16 – Repair patterns, acronyms and groups.

Group	Acronym	Pattern
codeMove	codeMove	Code Moving
condBlock	condBlockExcAdd	Conditional block addition with exception throwing
	condBlockOthersAdd	" " addition
	condBlockRem	" " removal
	condBlockRetAdd	" " addition with return statement
constChange	constChange	Constant Change
copyPaste	copyPaste	Copy/Paste
expFix	expArithMod	Arithmetic expression modification
	expLogicExpand	Logic expression expansion
	expLogicMod	" " modification
	expLogicReduce	" " reduction
nullCheck	missNullCheckN	Missing not-null check addition
	missNullCheckP	" null check addition
singleLine	singleLine	Single Line
wrapsWith	unwrapIfElse	Unwraps-from if-else statement
	unwrapMethod	" " method call
	unwrapTryCatch	" " try-catch block
	wrapsElse	Wraps-with else statement
	wrapsIf	" " if statement
	wrapsIfElse	" " if-else statement
	wrapsLoop	" " loop
	wrapsMethod	" " method call
	wrapsTryCatch	" " try-catch block
wrongRef	wrongMethodRef	Wrong Method Reference
	wrongVarRef	" Variable Reference

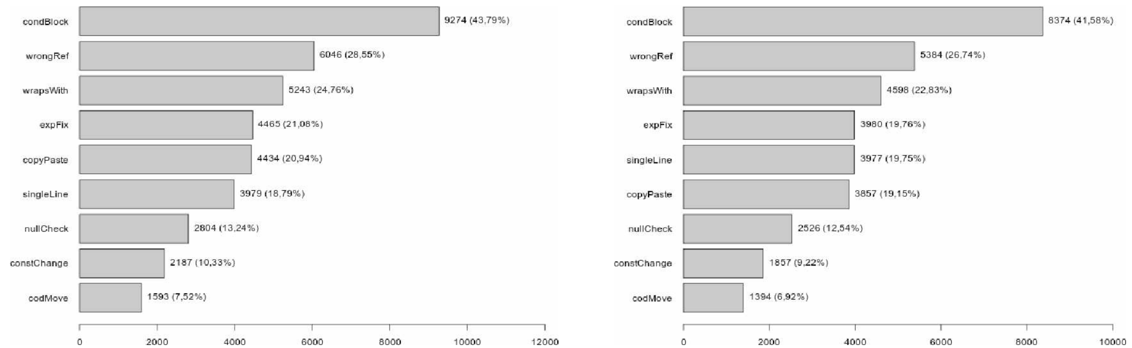
much higher number of bug patches and six different projects from those in Defects4J. Considering these differences, we have a very close distribution of the repair patterns.

The Figure 40 show the repair patterns distribution for each project in LR–dataset, limited to patch size of until 60 lines. *Conditional Block* continues as the top pattern for all the projects, while *Code Move*, and *Constant Change* remains in the opposite extreme, except in SWT, where *Null Check* swap in the lowest positions. Anyway, *Null Check* is another pattern in lowest positions for almost all the projects, with incidences from 9.10% to 17.68%. BIRT, Eclipse, JDT, and Tomcat shows distributions that resembles the overall distributions shown first, and with a laddered shape. AspectJ and SWT are the projects with a more visible difference when compared to the others. Both projects show the higher incidences of *Conditional Block* (AspectJ with 49.51%, and SWT with 46.63%). AspectJ has the lowest incidence of *Code Move* (2.76%), and



a) overall (21,177 patches). b) limited to 60 lines (20,138 patches).

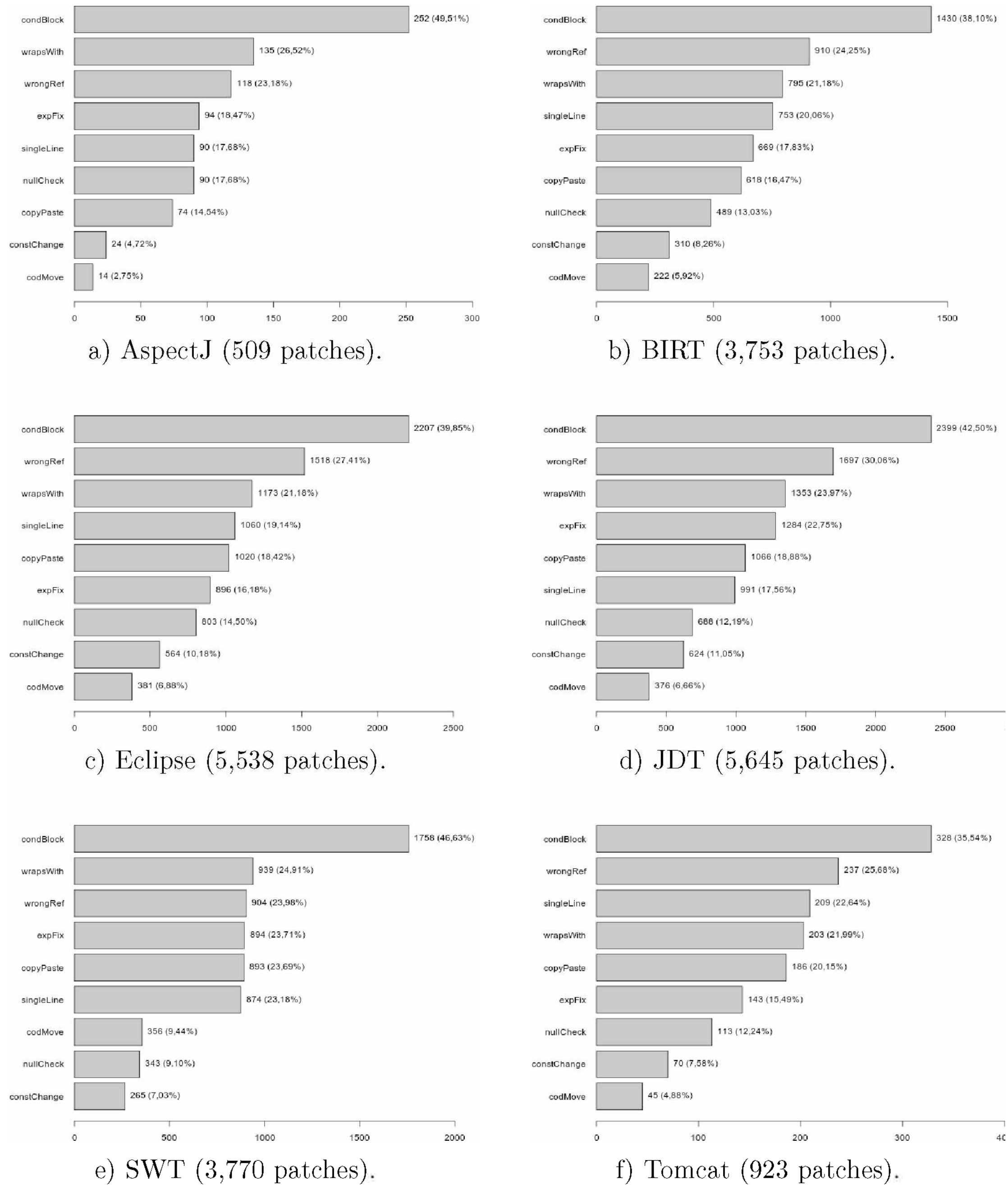
Figure 38 – Repair patterns found in LR–dataset.



a) overall (21,177 patches). b) limited to 60 lines (20,138 patches).

Figure 39 – Grouped repair patterns found in LR–dataset.

Constant Change (4.72%). *Copy Paste* (14.54%), *Null Check* (17.68%), *Single Line* (17.68%), and *Expression Fix* (18.47%) have an incidence slightly lower than *Wrong Reference* (23.18%), and *Wraps With* (26.72%). In SWT, *Constant Change* (7.03%), *Null Check* (9.10%), and *Code Move* (9.44%) are all bellow the 10% incidence, while *Single Line* (23.18%), *Copy Paste* (23.69%), *Expression Fix* (23.71%), *Wrong Reference* (23.98%), and *Wraps With* (24.91%) are all around 25% of incidence.



a) AspectJ (509 patches).

b) BIRT (3,753 patches).

c) Eclipse (5,538 patches).

d) JDT (5,645 patches).

e) SWT (3,770 patches).

f) Tomcat (923 patches).

Figure 40 – Grouped Repair Patterns incidence on LR–dataset projects.

6.4 Patterns composition

The basic idea behind the repair patterns found in Defects4J Dissection is finding implicit and more abstract structures that appear recurrently in many patches to fix the bugs in a benchmark dataset. We have found many of these structures and summarized them in 9 groups of repair patterns (Figure 36), totaling 25 specific repair patterns

when variations are considered (Figure 37), as discussed before and detailed in Table 16. We tried to name it meaningfully to facilitate its recognition and understanding. Some patterns, as *Single Line*, reflect the idea behind the most simple patches we can apply to fix a bug, while other patterns like *Copy Paste* may involve an arbitrary number of repair actions (possibly unrelated between different patches). The last sections show that around 19% of the patches in LR–dataset presents the *Single Line* pattern. While this observation gives an idea about the pattern representativeness in a bug dataset, even for such simpler patches, more questions may arise. Since having a single line is a very generic characteristic, the next natural question would be "what are the repair actions associated with this kind of pattern?", "is there any other type of characteristic associated with single line patches?". This section details some additional characteristics found for these and other patterns in LR–dataset, looking to clarify these and other questions.

6.4.1 Repair Actions

Each patch can contain one or more Repair Actions, and many of these actions can co-occur recurrently in many patches. The reasoning behind identifying many Repair Patterns comes from the perception of these repair actions recurrences. For example, the pattern *Wraps with Method* requires the addition of a method call around an existent piece of code, leading to the potential presence of the repair action *Method Call Addition*. A similar pattern, *Wraps with If*, will lead to the potential presence of other repair actions like *Conditional branch If addition* and other related actions that would depend on the logic of the added code structure. Since there is no guarantee of purity between the patterns or the actions found in a patch, analysis or assessment that does not consider these variabilities would be uncertain or imprecise conclusions. Next, we show some relations between repair patterns and the most common actions that appear when these patterns occur in the patches for the LR–dataset.

Many combinations are possible since we have identified 50 repair actions and 25 repair patterns. Taking just one pattern exclusively detected in a patch, the number of possible repair actions combinations is 2,369,936. It is just an estimation considering that the patches would have from 0 to 5 from the 50 recognized repair actions and based on the statistics for Defect4J, 75% of the patches have at most five repair actions detected. Of course, not all the combinations will occur in practice, and we next show the most common combinations found for each repair pattern in patches of LR–dataset.

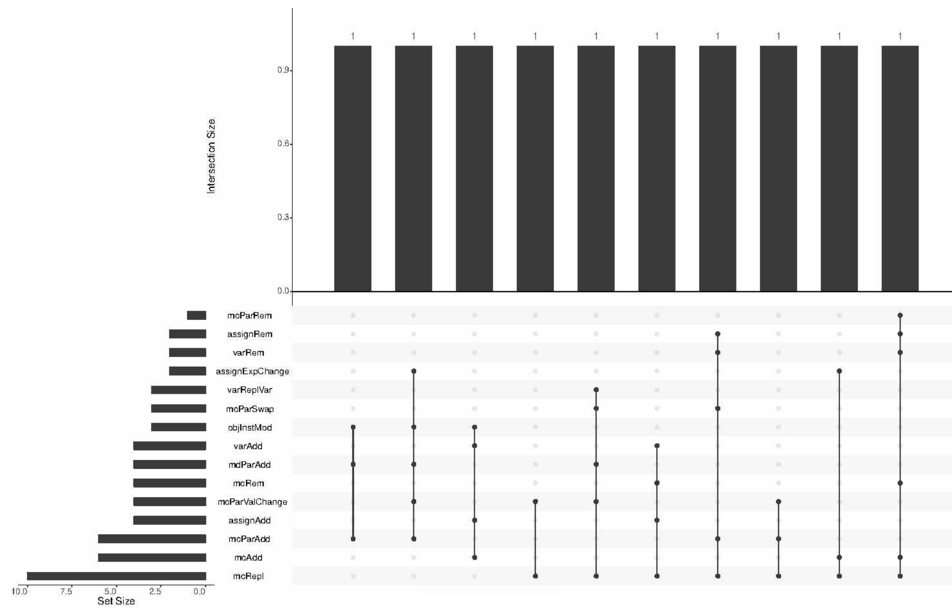
First, to extract the most common combinations, we analyze the intersection be-

tween the repair patterns and the co-occurring repair actions. The Figure 41 shows the intersection for the pattern *Wrong Method Reference* and the top-10 most common combinations in a) AspectJ and b) BIRT. The same is show in Figures 42 for projects Eclipse and JDT, and in Figure 43 for SWT and Tomcat. These figures are typical Upset charts (LEX et al., 2014). The horizontal bar chart shows the total number of patches containing each repair action enumerated in its vertical axis on the bottom left area. The vertical bar chart shows the number of intersections between the repair actions marked in the matrix just below each vertical bar on the top central area. A black circle appears when the repair action is in the intersection. When more than one repair action is in the intersection (co-occurs in a patch), its correspondent circles are connected by a line. In the AspectJ chart, we can see: 1. the most common repair action is *Method Call Replace* (`mcRepl`), with 10 occurrences (last bar on the bottom left area); 2. each repair action combination from the top-10 occurs in just one patch (vertical bars on the top area); 3. the first repair actions combination is composed by *Object Instantiation Modification* (`objInstMod`), *Method Definition Parameter Addition* (`mdParAdd`), and *Method Call Parameter Addition* (`mcParAdd`), shown by the three connected circles just below the first vertical bar; *Method Call Replace* co-occurs in at least seven different repair actions combinations (black circles in the last line of the matrix on the central area). For the BIRT chart, we have some considerations about *Method Call Replace* actions: 1. it is still the most common action, occurring in more than 60 patches; 2. this action occurs at least in 12 patches without intersection with the other shown actions; 3. the last line in the matrix shows that this action co-occurs in at least seven other combinations; 4. the second most common combination is composed of *Method Call Replace* and *Method Call Parameter Addition*, occurring in 8 patches.

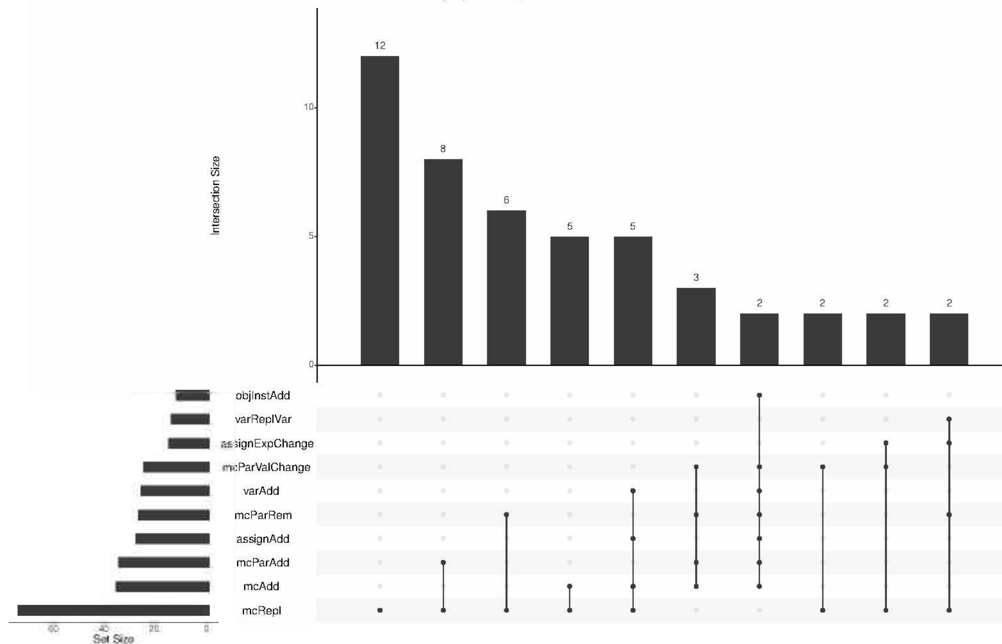
Since Repair Patterns are recurrent code structures found in the bug patches, some Repair Actions are expected in some of these patterns, while others would be associated with extra code and specificity of each patch. Again, considering the presence of the *Wrong Method Reference* pattern in a patch, we also would expect the presence of different groups of actions in some of the following situations:

- Wrong method called: a call to an alternative method to fix the bug, possibly with a different name. Some of the expected detected actions: Method Call Remove, Method Call Addition, Method Call Replace.

- Wrong parameter passed: to bug fix replaces a parameter, and this would imply in the call for an alternative method version (especially if the old and the new parameter types are different). Additionally to the previous case, the expected



(a) AspectJ

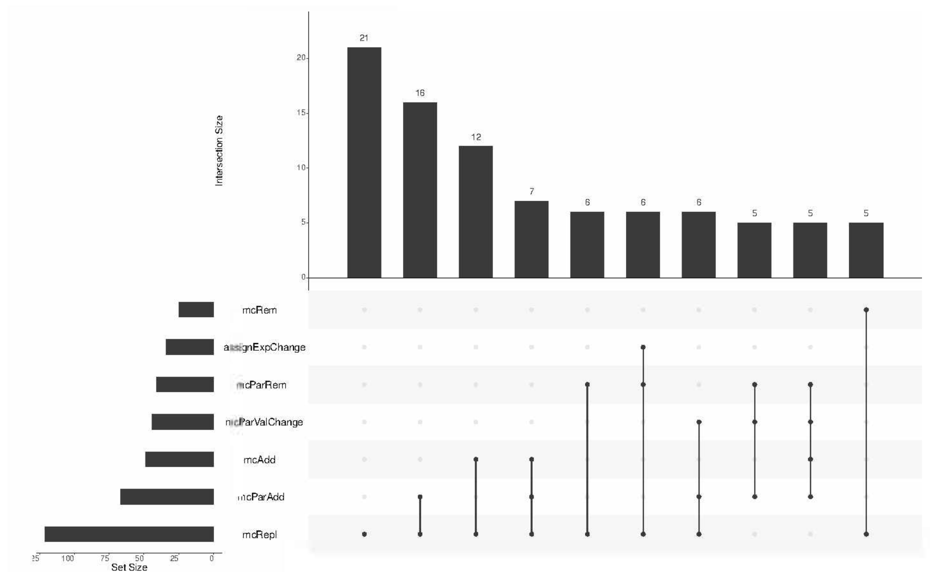


(b) BIRT

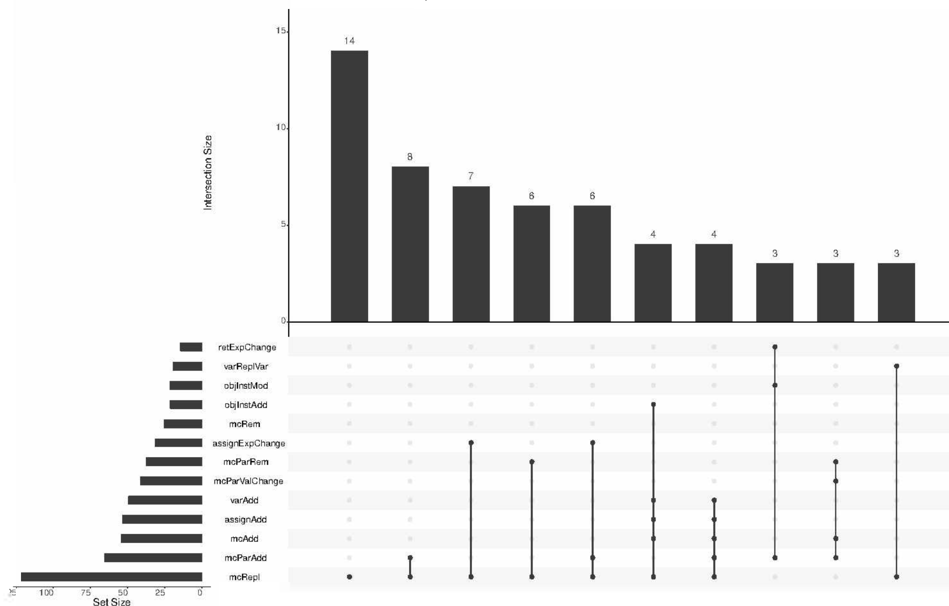
Figure 41 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) AspectJ and in b) BIRT.

detected actions would be: Method Call Parameter Value Change, Method Call Parameter Swap;

- Wrong number of parameters passed: to fix the bug, again, the parameters' addition (or removal) would imply in the call for an alternative method version.



(a) Eclipse

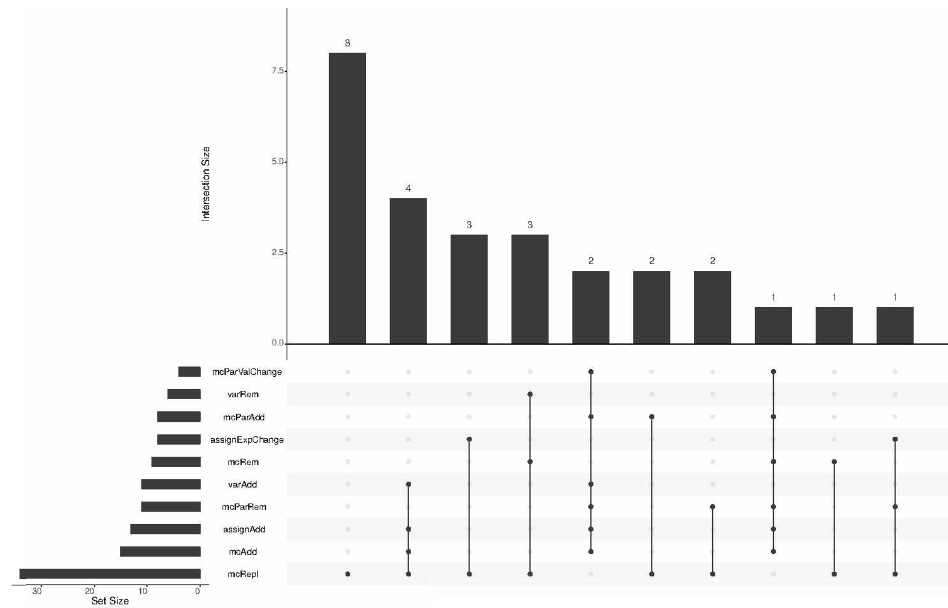


(b) JDT

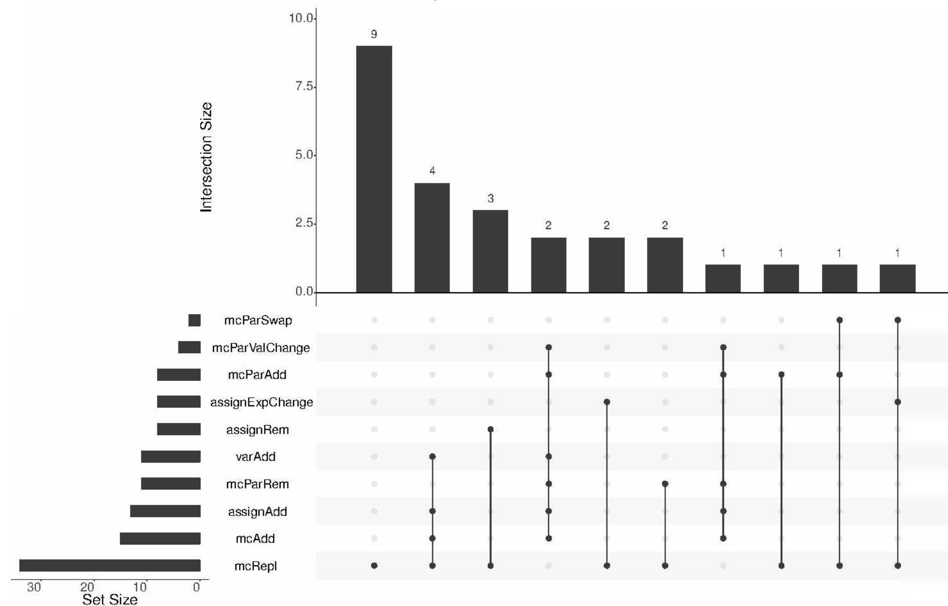
Figure 42 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) Eclipse and b) JDT.

Expected detected actions: Method Call Parameter Addition, Method Call Parameter Removal;

- Change in the returned value or the expression evaluation: the fixes in the above cases imply an expression change caused by methods with returned values assigned to a variable or methods composing a larger expression. Expected values: Assign-



(a) SWT



(b) Tomcat

Figure 43 – Most common Repair Actions co-occurrences for Wrong Method Reference repair pattern in a) SWT and b) Tomcat.

ment Expression Change, Assignment Addition, Assignment Removal.

- Wrong object instantiating: when one more of the above situations involves a change in the call for the object constructor to fix the bug. Expected detected actions: Object Instantiation Modification.

With "pure" repair patterns, the expected repair actions would ideally be the unique repair actions found in a patch. However, in practice, a patch is composed of different repair actions, repair patterns, and other required actions (some of these would not be detected by ADD or even categorized by the Defects4J Dissection). Therefore, when a given repair pattern is detected, it is not reasonable to consider that the patches will contain only the expected repair actions, as enumerated before with *Wrong Method Reference* pattern. Instead, the expected repair actions would be a starting point (or filter) to select these related patches, while the observed repair actions would still differ (a little or a lot, depending on the patch).

After the intersection analysis and the previous considerations about the expected and observed repair actions, we selected the most common groups of repair actions associated with each repair pattern for the projects in LR–Dataset. Next, from Figure 44 to 49, we show the selected groups of repair actions representing the found variations for each pattern. The simply detection of a repair pattern is also a considered variation, and it is marked as *<No Action>*, meaning that there are patches where ADD do not recognize any repair action for the repair pattern (i.e., no co-occurring repair action with the detected repair pattern).

6.4.2 Patterns Co-occurrences

Some repair patterns are detected isolated in its patches as occurs with the bug 7861 from Eclipse, whose patch matches with three occurrences of the *Wrong Variable Reference* (Figure 50 show the first occurrence patched in DecoratorManager.java file). Patches with more than one pattern are also common, as shown in Figure 51 with the patch for the bug 187445 from BIRT. The bug patch has two occurrences of *Logic Expression Expansion* and one occurrence of *Copy Paste* repair patterns. While we can expect a variability between patches composition in a bug dataset, it is essential to know when these patterns would co-occur. An analysis involving these patches would consider possible confounding factors associated with each pattern on the evaluation results.

From the 593 bugs found in LR–dataset for the AspectJ project, 521 were loaded and classified by ADD tool (72 is out). Applying an additional outlier filter (1 to 60 lines total, 1 to 20 chunks, 0 to 350 lines between chunks, 1 to 20 files changed), a total of 134 bugs were removed (outliers + not processed by ADD). The remaining 459 bugs were used in the next analysis and charts. Figure 52-a shows an UpSet Chart (LEX et al., 2014) with the top-20 more frequent sets of bugs included sets with co-occurrences of repair patterns and sets without co-occurrences (exclusively detected).

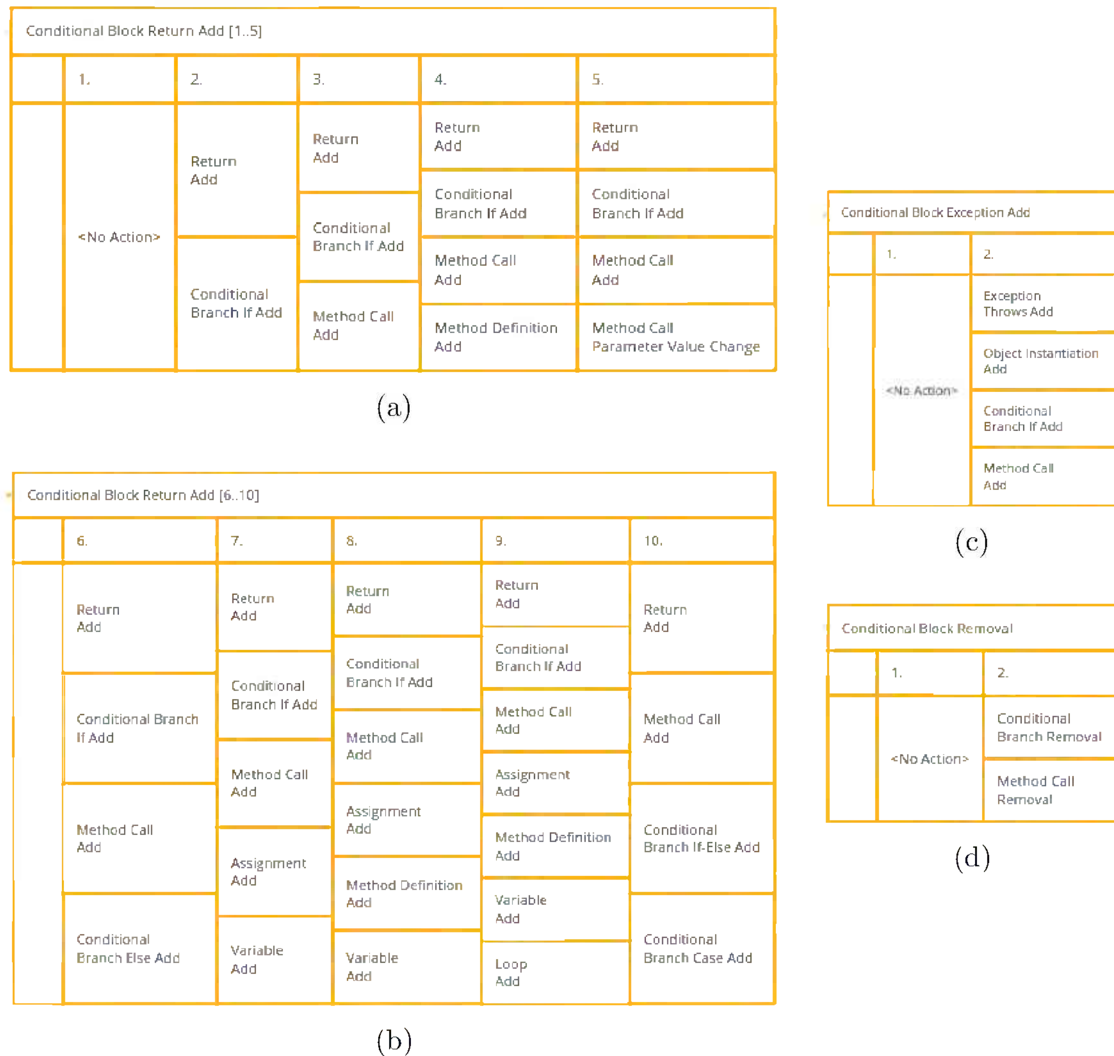
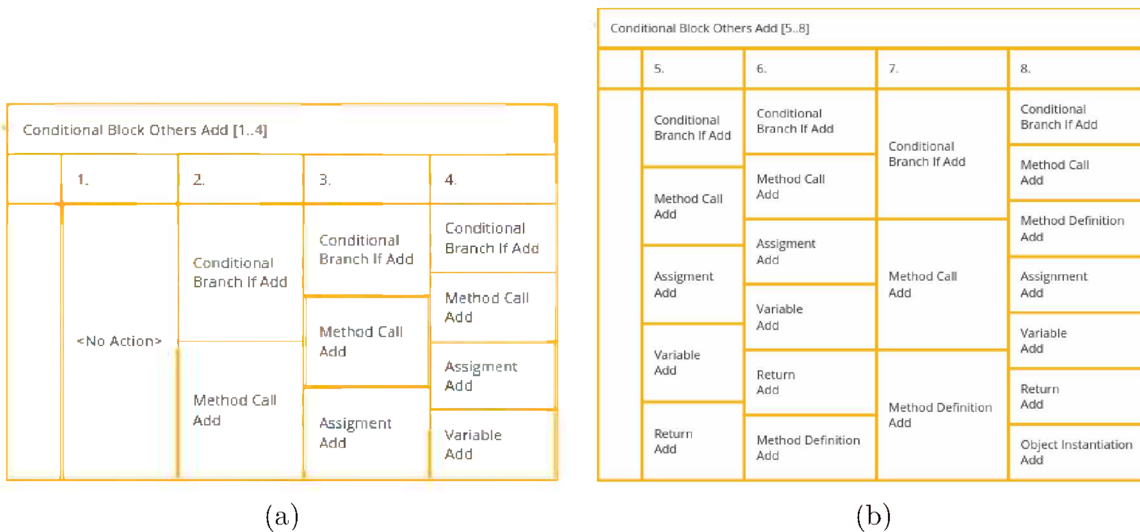


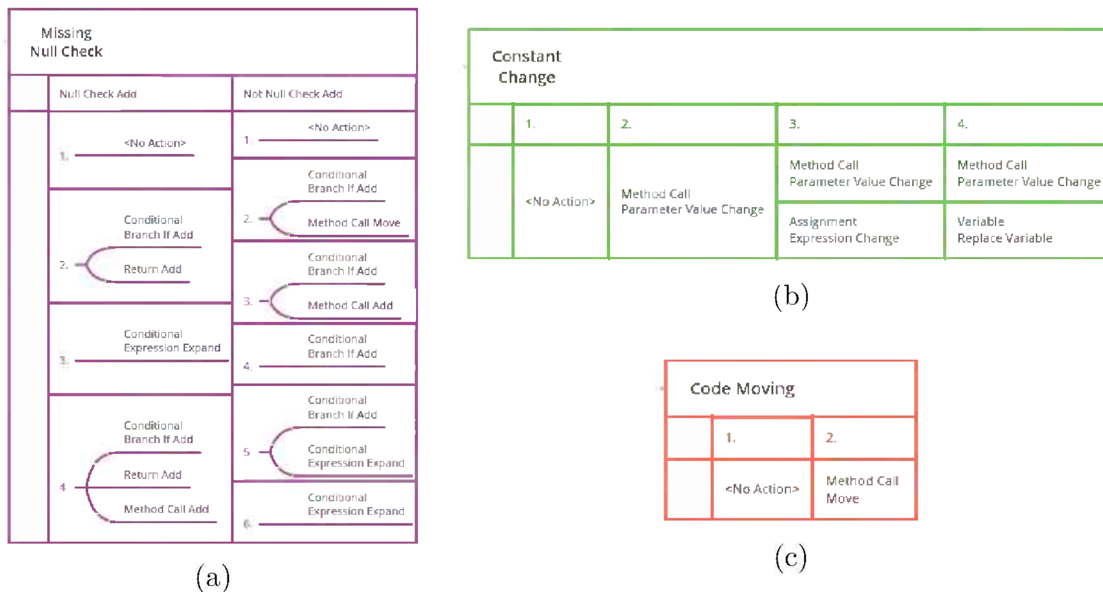
Figure 44 – Repair Patterns variations: a) Conditional Block Return Add (1..5); b) Conditional Block Return Add (6..10); c) Conditional Block Exception Add; d) Conditional Block Removal.

Conditional Block Others Addition (137 bugs, with 61 exclusive), *Single Line* (89 bugs, with 29 exclusive), *Conditional Block Return Addition* (80 bugs, 27 exclusive) are the top-3 sets of bugs. *Wrong Method Reference* (66 bugs, with 20 exclusive) is the fourth, but deserves a mention, because of the co-occurrences discussed next. The first more frequent co-occurrence occurs between *Wrong Method Reference* + *Single Line* in 15 bug patches, followed by *Expression Logic Expand* + *Single Line*, and also *Conditional Block Others Addition* + *Conditional Block Return Addition*, both with 12 co-occurrences. As show in the chart, except by *Expression Logic Expand* + *Missing Null Check (Negative)* + *Single Line* (7 co-occurrences in bug patches) there is no co-occurrence with more



(a) (b)

Figure 45 – Repair Patterns variations: a) Conditional Block Others Add (1..4); b) Conditional Block Others Add (5..8).



(a) (b) (c)

Figure 46 – Repair Patterns variations: a) Missing Null-Check; b) Constant Change; c) Code Moving.

than two patterns detected between the top-20 sets of bugs in AspectJ.

From the 4,178 bugs found in LR–dataset for the BIRT project, 4,167 were loaded and classified by ADD tool (11 is out). The outlier filter removes 984 bugs, while 3,230 bugs remain for analysis. Figure 52-b shows the UpSet Chart. *Single Line* (744 bugs, with 383 exclusive), *Conditional Block Others Addition* (688 bugs, with 274 exclusive), and *Wrong Method Reference* (467 bugs, with 153 exclusive) are int the top-3 set of bugs.

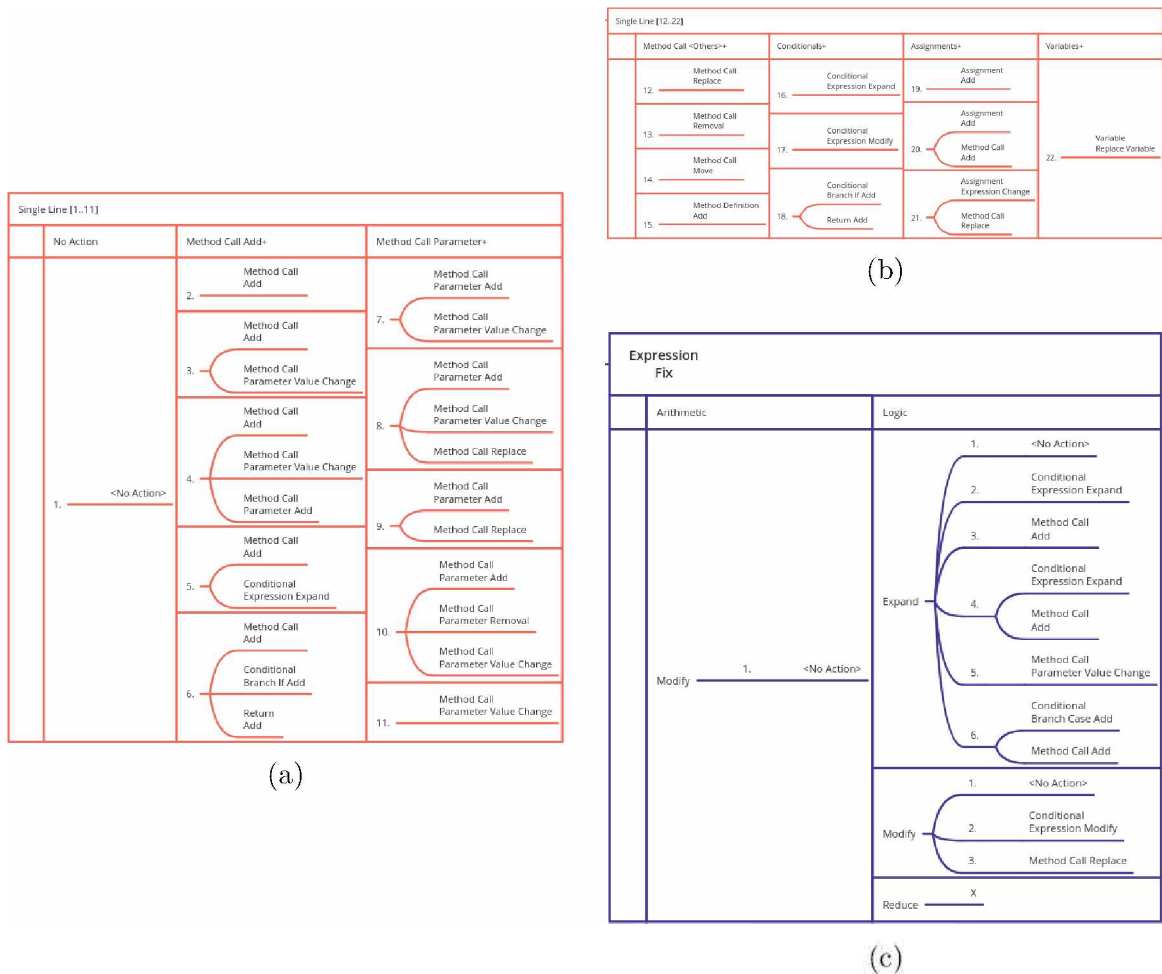
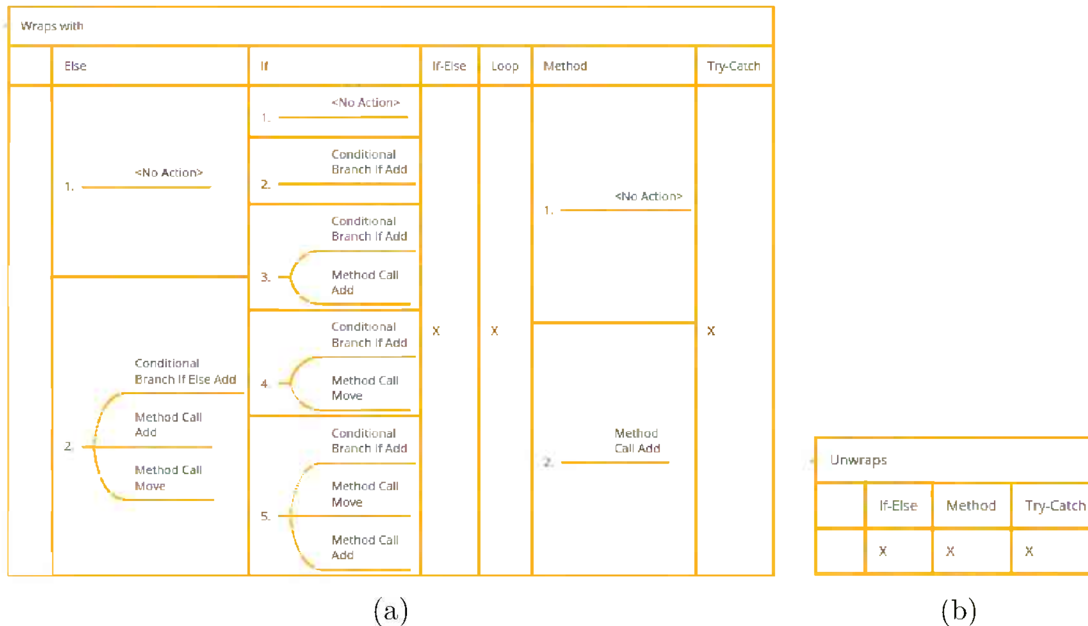


Figure 47 – Repair Patterns variations: Single Line a) 1 to 11; b) 12 to 22; c) Expression Fix.

The first more frequent co-occurrence occurs between *Wrong Method Reference + Single Line* in 92 bug patches, followed by *Wraps with If + Missing Null Check (Negative)*, both with 59 co-occurrences. As shown in the chart, there is no co-occurrence with more than two patterns detected in the top-20 set of bugs in BIRT.

From the 6,495 bugs found in LR–dataset for the Eclipse Platform UI project, 5,839 were loaded and classified by ADD (656 is out). The outlier filter removes 1,508 bugs, while 4987 bugs remain. Figure 53-a shows the UpSet Chart. *Conditional Block Others Addition* (1,154 bugs, with 582 exclusive), *Single Line* (1,050 bugs, with 405 exclusive), *Wrong Method Reference* (718 bugs, with 225 exclusive) are in the top-3 set of bugs. The first more frequent co-occurrence occurs between *Wrong Method Reference + Single Line* in 113 bug patches, followed by *Wraps with If + Missing Null Check (Negative)* occurring in 93 bug patches. As shown in the chart, there is no co-occurrence with more



(a) (b)

Figure 48 – Repair Patterns variations: a) Wraps-with; b) Unwraps-with.

than two patterns detected in the top-20 set of bugs in Eclipse.

From the 6,274 bugs found in LR–dataset for JDT project, 6,172 were loaded and classified by ADD (102 is out). Applying the outlier filter, a total of 1,342 bugs were removed, and 4,932 bugs remains. Figure 53-b shows the UpSet Chart. *Conditional Block Others Addition* (1,125 bugs, with 340 exclusive), *Single Line* (966 bugs, with 441 exclusive), *Wrong Method Reference* (841 bugs, with 256 exclusive) are in the top-3 set of bugs. For the co-occurrences, first *Conditional Block Others Addition* + *Conditional Block Return Addition* appears in 118 bug patches, followed by *Single Line* + *Wrong Method Reference* in 110 bugs, and then *Single Line* + *Expression Logic Expand* in 105 bug patches. As before, there is no co-occurrence with more than two patterns detected in the top-20 set.

From the 4,151 bugs found in LR–dataset for SWT project, 4,114 were loaded and classified by ADD (37 is out). Applying the outlier filter, a total of 981 bugs were removed, and 3,213 bugs remains. Figure 54-a shows the UpSet Chart. *Conditional Block Others Addition* (981 bugs, with 393 exclusive), *Single Line* (871 bugs, with 410 exclusive), *Wrong Variable Reference* (329 bugs, with 84 exclusive) are in the top-3 set of bugs. For the co-occurrences, first *Conditional Block Others Addition* + *Conditional Block Return Addition* appears in 83 bug patches, followed by *Single Line* + *Expression Logic Expand* in 82 bug patches and *Single Line* + *Wrong Method Reference* in 71 bugs. One more time, there is no co-occurrence with more than two patterns detected in the

Wrong Method [1..5]				
1.	2.	3.	4.	5.
<No Action>	Method Call Replace	Method Call Replace Assignment Expression Change	Method Call Replace Assignment Expression Change Method Call Parameter Add	Method Call Replace Method Call Parameter Add

(a)

Wrong Method [6..10]				
6.	7.	8.	9.	10.
Method Call Replace Method Call Parameter Add Method Call Parameter Value Change	Method Call Replace Method Call Parameter Value Change	Method Call Replace Variable Removal	Method Call Parameter Add Object Instantiation Modification	Method Call Parameter Add Method Call Parameter Value Change Method Call Parameter Removal

(b)

Wrong Variable				
1.	2.	3.	4.	5.
<No Action>	Method Call Parameter Value Change	Method Call Parameter Value Change Variable Replace Variable	Method Call Parameter Value Change Method Call Parameter Add Method Call Parameter Removal	Variable Replace Variable

(c)

Figure 49 – Repair Patterns variations: Wrong Method Reference a) 1 to 5; b) 6 to 10; c) Wrong Variable Reference.

top-20.

From the 1,056 bugs found in LR–dataset for Tomcat project, 956 were loaded and classified by ADD (100 is out). Applying the outlier filter, a total of 234 bugs were removed, and 822 bugs remains. Figure 54-b shows the UpSet Chart. *Single Line* (208 bugs, with 101 exclusive), *Conditional Block Others Addition* (187 bugs, with 84 exclusive), *Wrong Method Reference* (123 bugs, with 44 exclusive) are in the top-3 set of bugs. For the co-occurrences, first *Wrong Method Reference* + *Single Line* appears in 29 bug patches, followed by *Expression Logic Expand* + *Single Line* and *Missing*

Bug-ID: Eclipse-7861
Title: Bug 7861 Multiple enabled decorators doesn't work
Description: If you have two decorators enabled at the same time only one of them wins and gets to decorate. If you load the org.eclipse.team.* projects from dev.eclipse.org and enable both the CVS and Team Examples decorators then create a CVS project and an Example (file system provider) both decorations are never shown although both decorators are enabled.

```
/bundles/org.eclipse.ui/Eclipse UI/org/eclipse/ui/internal/DecoratorManager.java

@@ -91,7 +91,7 @@ public class DecoratorManager
   DecoratorDefinition[] decorators = getDecoratorsFor(element);
   String result = text;
   for (int i = 0; i < decorators.length; i++) {
-      result = decorators[i].getDecorator().decorateText(text, element);
+      result = decorators[i].getDecorator().decorateText(result, element);
   }
}
```

Figure 50 – Bug Report for the bug 7861 from Eclipse, with a snippet of the patch matching the *Wrong Variable Reference* repair pattern.

Null Check (Negative) + *Conditional Block Others Addition*, both occurring in 22 bug patches. Again, as show in the chart, there is no co-occurrence with more than two patterns detected in the top-20 set of bugs in Tomcat.

6.5 Actual applications for the Defects4J Dissection study

Considering the exposed in previous sections, we would imagine and raise potential applications for the dissection study, initially applied to Defects4J, and then extended in this chapter with LR–dataset. However, since the dissection study with Defects4J (SOBREIRA et al., 2018), we observed many citations confirming a consistent interest from the research community to the dissection study and giving a more realistic idea about the actual applications. Until November 2021, we account for 82 citations of the Defects4J Dissection study from the Arxiv pre-print in 2017 to the final paper version in SANER'18 conference. The number of citations have been increasing each year. The Dissection study's works concentrate especially on Automatic Program Repair (APR) and Bug Localization (BL) as shown in Figure 55. Still, other areas are also accessing it as those studying API Misuse (API), Bug and Patch Analysis (BPA), Bug Datasets

Bug-ID: Birt-187445
Title: Bug 187445 The joint result of script computed column is incorrect[07]
Description: 1, Each dataset including a computed column: name: index; type:integer; expression:0 2, Specify the script of datasets to: beforeOpen: index=0; onFetch: row["index"] = index; index++; 3, Join the two datasets Build number: 2.2.0.v20070516-0630 Steps to reproduce: 1, open the attached report design 2, open the joint dataset and preview Actual result: The result of column Data "Set1::index" is all 0 but no value 1.

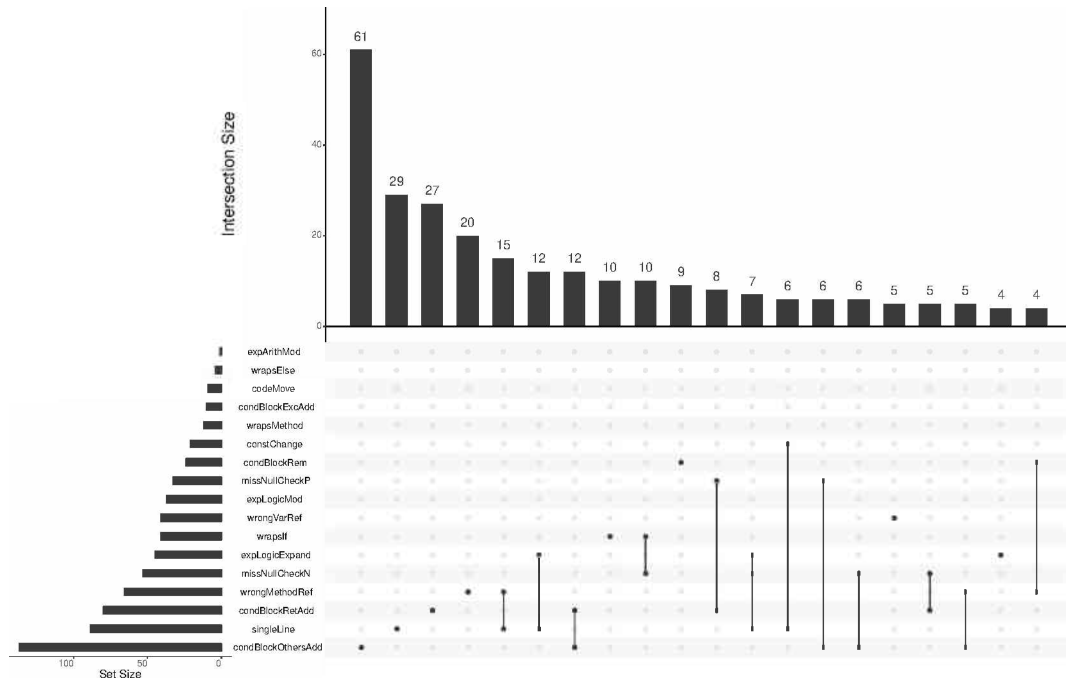
```
/data/org.eclipse.birt.data/src/.../engine/impl/ComputedColumnHelper.java

@@ -194,7 +194,8 @@ public class ComputedColumnHelper implements
    IResultObjectEvent
     continue;
  }

- if ( ExpressionCompilerUtil.hasAggregationInExpr(column.getExpression()) )
+ if ( ExpressionCompilerUtil.hasAggregationInExpr(column.getExpression()) ||
+      column.getAggregateFunction() != null )
  {
     continue;
  }
@@ -254,7 +255,8 @@ public class ComputedColumnHelper implements
    IResultObjectEvent
  }
  if ( column != null )
  {
-   if ( ExpressionCompilerUtil.hasAggregationInExpr(column.getExpression()) )
+   if ( ExpressionCompilerUtil.hasAggregationInExpr(column.getExpression()) ||
        column.getAggregateFunction() != null )
    {
      return true;
    }
```

Figure 51 – Bug Report for the bug 187445 from BIRT, with a snippet of the patch matching the *Logic Expression Expansion* and *Copy Paste* repair patterns.

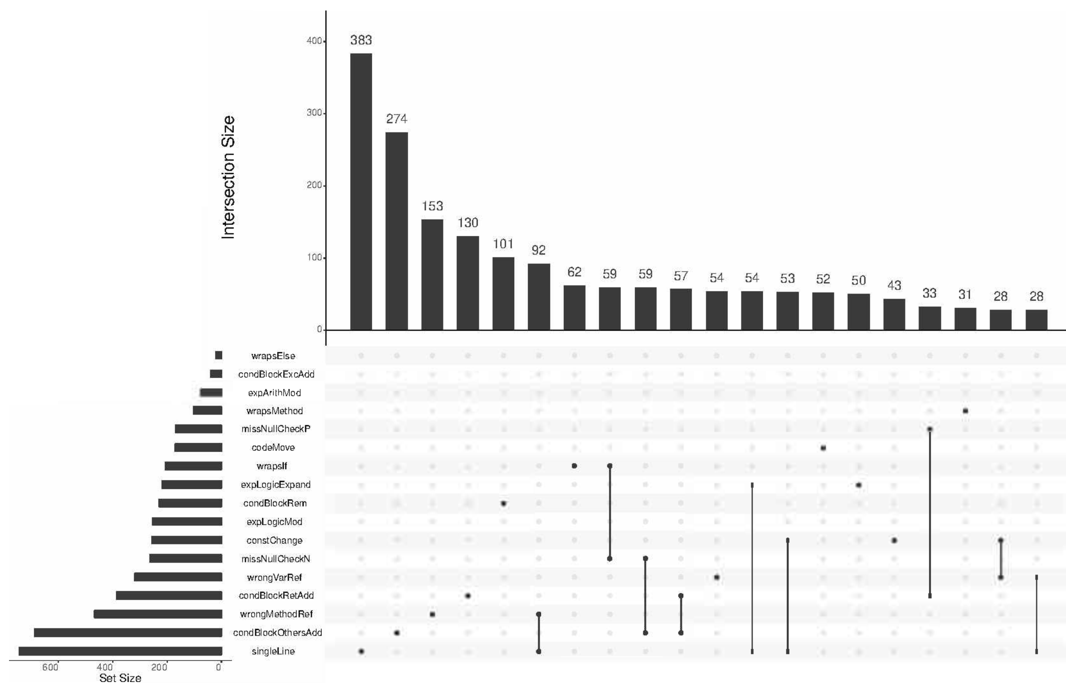
(BD), Debugging (DBG), Program Synthesis (PS), Source Code Analysis (SCA), and Software Testing (TS). Most of the publications are in Conference Papers, followed by Journals, but we can find citations in Pre-prints (most in Arxiv), Ph.D. Thesis, MSc. Dissertations, a book chapter, and a public presentation.

6.6 Related Work

Our dissection study (SOBREIRA et al., 2018) on the 395 bugs of Defects4J bug dataset (JUST; JALALI; ERNST, 2014) produced similar results to the presented in this chapter, especially those related to size and spreading dimensions. We found that



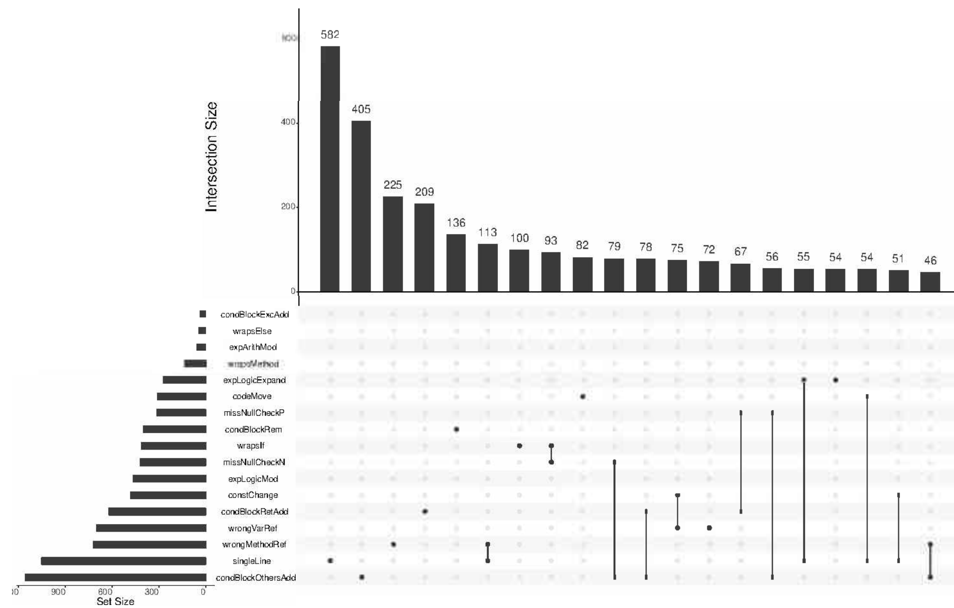
(a)



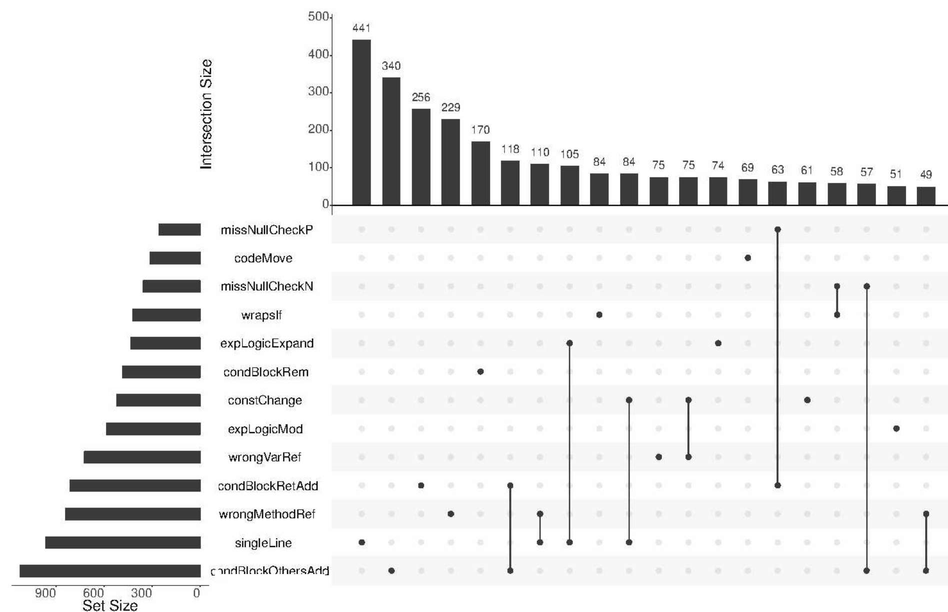
(b)

Figure 52 – Patterns co-occurrence without outliers in: a) AspectJ and b) BIRT.

95% of Defects4J bugs have patches with at most 22 changed lines, in blocks of 236 lines (including buggy and non-buggy lines in the gap between first and last buggy line,



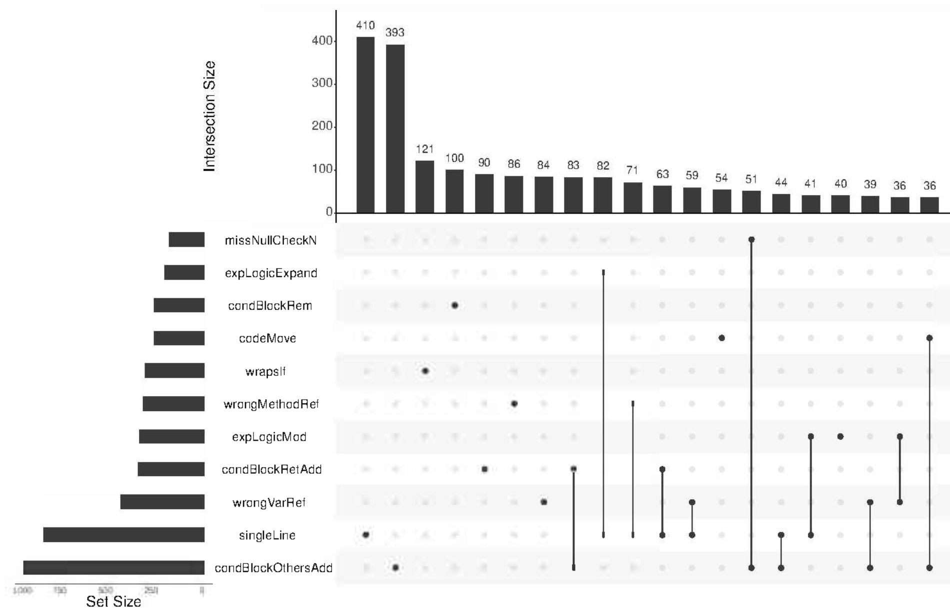
(a)



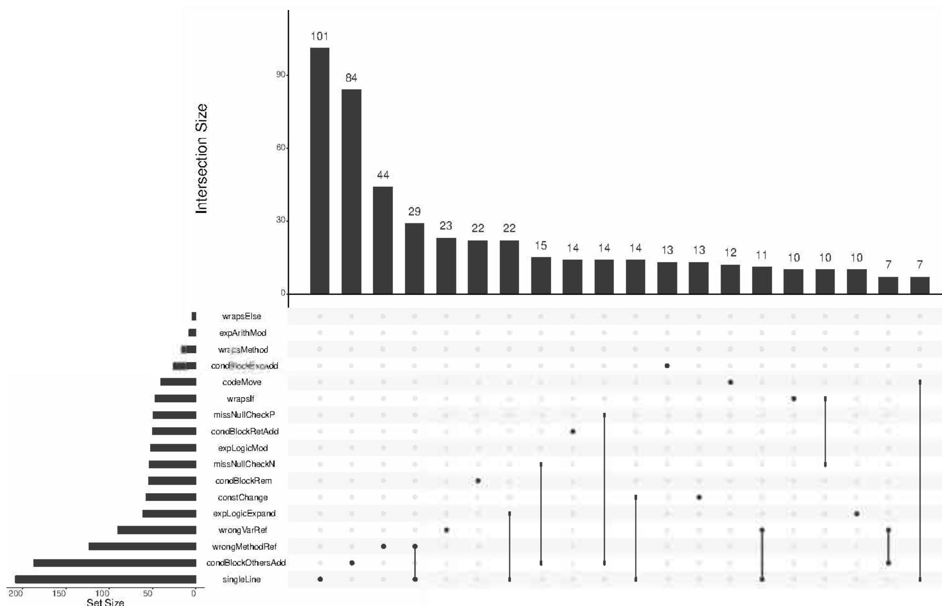
(b)

Figure 53 – Patterns co-occurrence without outliers in a) Eclipse Platform UI and b) JDT.

excluded empty or comment lines), and spans at most three methods, and two files (1 file in 92.41% of patches). The dissection on Defects4J (SOBREIRA et al., 2018) has a good intersection with Liu et al. (2018), since we also consider similar repair actions in an initial manual analysis, culminated in the recognition of many repair patterns. In



(a)



(b)

Figure 54 – Patterns co-occurrence without outliers in a) SWT and b) Tomcat.

our subsequent work (MADEIRAL et al., 2018) we also applied GumTree and Spoon (PAWLAK et al., 2015) to recognize 9 kind of repair patterns automatically (or a total of 25 repair patterns, if variations are considered individually).

Several bug datasets exist to support empirical studies on techniques and tools related to software bugs. Usually, these datasets do not include detailed information on the *bugs*

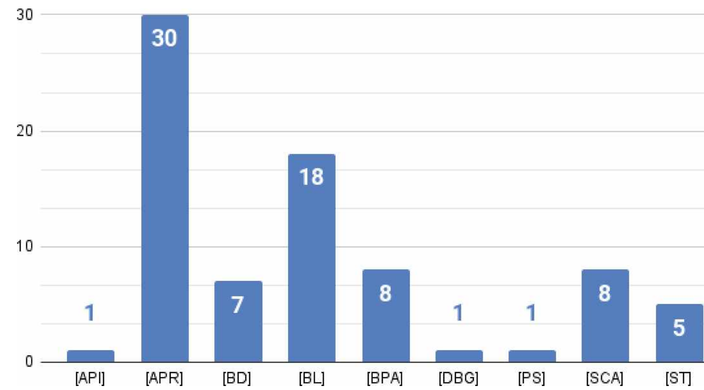


Figure 55 – Applications of Defects4J Dissection study by research area until November of 2021.

and their *patches* if any (e.g., Siemens suite (HUTCHINS et al., 1994) and SIR (DO; ELBAUM; ROTHERMEL, 2005)), or they include simple information on the *bugs* (e.g., BugBench (LU et al., 2005)), like bug type. Next, we present notable and recent bug datasets where information about the *patches* are delivered, which is close to our work on Defects4J.

iBugs (DALLMEIER; ZIMMERMANN, 2007) (390 Java bugs) contains bugs annotated with size and syntactic properties on their patches. iBugs' size properties include similar patch size and spreading metrics. iBugs' syntactic properties consist of fingerprints describing which syntactic tokens the patch changed, such as keywords, method calls, and expressions, augmented with information on variable usage, operators, and literals. These fingerprints are similar to the repair actions, but we organize the taxonomy in this work differently. For instance, the groups of token "keyword" and "expression" in iBugs represent different changes on `if`; we used the repair action group "Conditional" specific to changes on conditionals. Distinctly, our analysis includes repair patterns that they have not investigated.

ManyBugs (GOUES et al., 2015) (185 C bugs), besides information on the bugs, delivers manually evaluated information about patches. Each patch annotates whenever some changes happen, for instance, in functions, loops, conditional and function calls, and arguments to a function or function signature. The process is similar but more fine-grained with repair actions. They also calculated the number of changed lines (size) and changed files (spreading), but differently, ManyBugs does not provide the number of chunks and repair patterns.

Codeflaws (TAN et al., 2017) (3902 C bugs) delivers bugs annotated with syntactic differences between buggy and patch code at AST level. Like in iBugs, Codeflaws'

syntactic differences are similar to repair actions, but we use a more comprehensive taxonomy; for example, in Codeflaws, conditionals and loops are considered together in one group, "control flow". Moreover, Codeflaws delivers no information on patch size, spreading, or repair patterns.

Motwani et al. (MOTWANI et al., 2018) have annotated each bug in Defects4J with abstract parameters regarding five characteristics: importance, complexity, independence, test effectiveness, and characteristics of the human-written patch. One example of an abstract parameter is the number of lines edited in a patch, which applies to compute the defect complexity. Similar to this work, they annotated Defects4J bugs with patch size and the number of modified files. On the characteristics of the patches, they annotated the bugs with nine code modification types, such as whether the patch contains the addition of method calls, which are similar to our repair actions. However, the used taxonomy of repair actions in this work is more comprehensive and fine-grained since the actions were arranged in groups considering more detailed changes. For instance, instead of only showing a generic change in patch arguments of a method call, we detail with information about an argument's addition (or removal), a change on the argument value, or the argument swap in a method call. Moreover, Motwani et al. considered other information, such as the number of relevant test cases, which makes both works complementary for the Defects4J part. Our work extends it with a much larger scale dataset.

Pan et al. (PAN; FELLOW, 2009) and Soto et al. (SOTO et al., 2016) identified patterns in human patches. Pan et al. (PAN; FELLOW, 2009) manually analyzed seven open-source projects and found 27 bug fix patterns covering from 46 to 64% bug fixes. Furthermore, they observed that the most common bug fix patterns are related to the method call and if condition (both are around 20% bug fixes), which is consistent with our findings, since *Method Call Addition* and *Conditional Branch Addition* are the most prevalent repair actions in patches. Nonetheless, the proportions have some differences indicating that different choices in bug selection may induce different performance rates on approaches assessed with such datasets.

Tomassi et al. (2019) proposes a dataset of reproducible bugs for Java and Python. Their main goal is to provide a dataset with failing and passing pairs to help drive research on bug localization and automatic repair approaches. They characterize bugs with failing-pass-oriented characteristics and do not focus on the characteristics of bugs similar to the taxonomy proposed in this work.

6.7 Final Considerations

In this chapter, we analyzed LR–dataset according to the patch dimensions defined in the Defects4J Dissection study (SOBREIRA et al., 2018). While Defects4J would be considered a relatively small size bug dataset with only 395 bug patches extracted from six Java projects, LR–dataset pushes this scale to a different level. LR–dataset contains more than 20 thousand bug patches, also extracted from six Java projects but different from the projects found in Defects4J.

After extracting the repair patterns (and the other analysis dimensions) from Defects4J, the first question is how these patterns would appear in other projects and how they would not be a singularity from the Defects4J dataset. We have shown that, in fact, and based on the Defects4J Dissection dimensions, there are far more similarities between the bug patches in these datasets than differences, despite the very high scale distance between the number of patches in each dataset. Furthermore, all the patterns found in Defects4J were also in LR–dataset and some of them appear in even higher proportions.

The repair actions and patterns provide another layer to understand the bug nature inside a dataset. Beyond the type of lines edited to apply a patch, we can know the meaning and the representative level of each type in a bug dataset. "Do this dataset covers programmers' errors related to the absence of some verification coding?" The presence of a *Null Check* pattern would help to address it. "Usually, do bugs in a dataset require simpler or more complex patches?" *Single Line* incidences would help to quantify part of the simpler fixings, while patch size and spreading distributions would show the level of code demanded and help to answer about the complexity involved in the other extreme. "How many bugs are related to errors in testing conditions?" A study on the *Expression Fix* patterns would help with this type of issue. A dissection study on the patches composing the target dataset as we have presented for LR–dataset and our first study on Defects4J would address the last questions and many others. The dissection study would help guide research path decisions and the analysis for the evaluation of the proposed approaches. Next chapter, we conduct an experiment to show how much these pattern dimensions would influence the evaluation of the approaches to the BL problem.

Chapter 7

Influence of repair patterns on BL approaches

Previous chapters have shown some of the characteristics of bug datasets that the research approaches do not usually consider in their evaluations, as occurs in tasks like Bug Localization and Automatic Program Repair. We have analyzed patch characteristics, initially found in the Defects4J dataset and then found in LR–dataset. To show how these characteristics would influence the evaluations based on the bug datasets, we present the results of the conducted experiments considering some of the described patch characteristics. Previous approaches to automatic BL are our baseline since it does not differentiate the bugs' nature. Then, the patch characteristics related to each bug report, especially the repair patterns, are applied to guide the dataset sampling of bug reports. The repair patterns' presence and absence on the samples can produce features and BL scores with statistically significant differences compared to approaches that do not consider the bug patches characteristics. Next, we detail our study.

7.1 Research Questions

In Chapter 6 we described factors involved in the characterization of a large bug dataset, following our initial work with the dissection of Defects4J. Most research approaches do not proceed with a broad characterization of their bug datasets, considering the presented dimensions. Likewise, the review of past research with dataset characterization and under new perspectives or frameworks as we show in Chapter 4 is also a significant challenge. Obliviously we do not intend to cover all the issues and possible unfoldings since it is a work for some years ahead of research. Here we focus on how the

repair patterns would affect the Bug Localization task, trying to characterize the extension of this impact (or given some clues). First, then, we define the research questions that we will try to focus on and answer objectively in this chapter:

RQ6 When we compare a sample of bugs where the respective patches match a given repair pattern against another sample of bugs where this pattern is not present, is there any difference in the measured metrics targeting the ranking of bug suspects? Are these differences statistically significant?

RQ7 What type of impact is associated with the evaluated metric's score rankings by the presence of a repair pattern in the patches of a bug sample? Moreover, when the repair pattern is absent?

RQ8 What is the degree of the impact correlated to the repair pattern's presence or absence on the metrics measured?

7.2 Evaluation Method

Since we applied the dissection analysis to LR–dataset and considering its application on studies for Bug Localization tasks like (YE; BUNESCU; LIU, 2014), this dataset is our natural choice. The section presents some of the preparation steps to conduct the experiments. Subsection 7.2.1 presents the filtering criteria for the bug reports considered for the experiments, including the outlier considerations. Subsection 7.2.2 presents the selected settings for the set of samplings considered for the experiments. Subsection 7.2.3 summarizes the metrics, hypothesis tests and formulations. Finally, Subsection 7.2.4 briefly shows the experimental runtime environment.

7.2.1 Dataset preparation and cleaning

Not all bug reports and files in the LR–dataset (YE; BUNESCU; LIU, 2014) were chosen for the experiments. Some of the reasons are 1) patches with testing code: some bug reports are associated with the fixing of testing code (exclusively or not); 2) files out of the project and analysis scope: some files do not seem to be related to the main functional features of its projects, some files are not in the project development scope, and some files are not Java source code files; 3) no baseline results: some bug reports do not present the ranking results of the fixed files obtained with the LR approach (YE; BUNESCU; LIU, 2016); 4) bug patches not processed by ADD: some bugs do not have

any repair pattern to detect, neither other supported bug patches properties to analyze and as a consequence, ADD do not have any result to produce for these patches; 5) outliers: some bug patches have properties that exceeds a lot the properties found on most patches in the dataset and are outliers instances.

The fixing of testing code is not the primary target of a bug localization approach since the adoption of unit testing practices is not yet a universal reality in software development, and the production of functional software is possible without testing code. While desired and recommended, the testing practices are strategies to improve the software quality. Usually, the testing code is not a directly influential factor on a faulty functional behavior caused by a bug in a software. Additionally, some studies have already pointed out disadvantages in maintaining testing code in the datasets to evaluate bug localization approaches and how it negatively impacts the results (KIM; LEE, 2018). Thus, we discard all bug reports when only testing code is the fixing target. We consider bug reports fixed by testing code only if they involve fixing at least one functional source code. However, the testing code is wholly ignored in the source code search space to generate the rankings of potential fixing targets even in these cases. Like testing code, other types of files found in the dataset do not seem related to the main project functionally or the project development scope. Therefore, we excluded these out-of-scope files from the source code search space, e.g., folders containing documentation, configuration files, setup files, external libraries, or non-Java source code files.

We defined three categories to differentiate these possible types of bug reports based on the objectives for the code: functional code only, non-functional code only, and mixed code (functional + non-functional). To achieve this first level of filtering, we manually looked at project repository folders and identified folder names associated with non-functional source code. Some examples of these folders are: unit tests ("test", "test suite"), documentation (e.g., "docs"), and external libraries (e.g. "libs"). We excluded the folders (or Java packages) from the source code search space containing the name patterns enumerated next. These are our assumptions about the location of non-functional codes in each project.

1. AspectJ: "*.tests*", "*.tests.*", "testsrc", "testdata", "testing", "tests", "test", "docs", "lib".

2. BIRT: "*.tests*", "*.tests.*", "testsuites", "tests", "testhelper", "test", "common", "docs", "features", "nl".

3. Eclipse: "*.tests*", "*.tests.*", "tests".

4. JDT: "*.tests*", "*.tests.*", "tests", "junit".

5. SWT: "*.tests*", "*.tests.*", "tests".

6. Tomcat: "bin", "conf", "res", "test", "webapps".

We also have found bug reports without information about the ranking with the LR approach from Ye et al. (YE; BUNESCU; LIU, 2016), i.e., the result section in the provided XML file was empty. This situation does not directly compare with the original LR approach, so we also ignore these bug reports. Below are the Bug-ID's of some bug reports without results for the original LR approach:

1. AspectJ: 259528, 249710, 84260.

2. BIRT: 211884, 375600, 362714.

3. Eclipse: 413943, 411967, 209190.

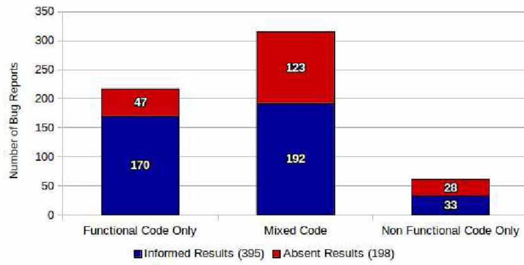
4. JDT: 277299, 262389, 158292.

5. SWT: 409353, 312371, 308445.

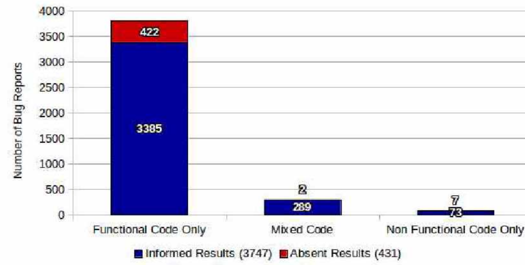
6. Tomcat: 55245, 55217, 55046.

Figure 56 shows the distributions of the bug reports according to the previous definitions. The most significant difference occurs for AspectJ, where there are more bug reports involving mixed code (192+123=315) than functional code (170+47=217). The number of absent results for the LR approach is proportionally higher (47+123+28=198), remaining 362 bug reports to be considered from the original 593 bug reports in the dataset file (170 functional, and 192 non-functional, both with results informed, and corresponding to 61.05% of the total available in AspectJ). For the other projects, the number of bug reports with bug fixes only in functional code is higher than non-functional code fixes, and most parts have informed results for LR. The number of reports selected for each project considering the total available in the dataset are: 362 (61.05%) for AspectJ, 3,674 (87.94%) for BIRT, 5,609 (86.36%) for Eclipse, 5,365 (85.51%) for JDT, 3,754 (90.44%) for SWT, and 934 (88.45%) for Tomcat. Overall, the total number of bug reports to be considered (informed results, functional + mixed code) is 19,698 (86.60% from the original 22,747 bug reports).

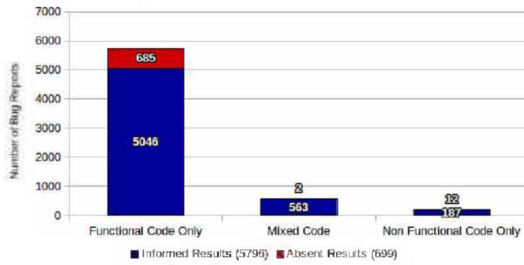
ADD cannot process some patches. ADD tries to detect patterns in patches extracting the Abstract Syntax Tree resulting from the difference between the buggy and the



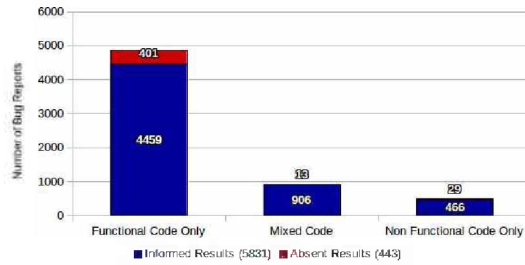
a) AspectJ (593 bug reports).



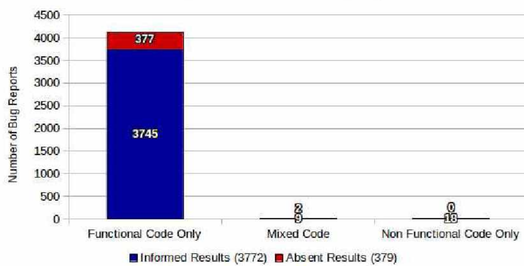
b) BIRT (4,178 bug reports).



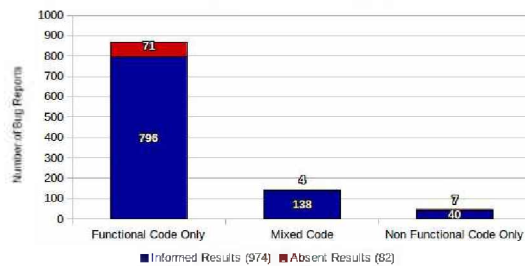
c) Eclipse (6,495 bug reports).



d) JDT (6,274 bug reports).



e) SWT (4,151 bug reports).



f) Tomcat (1,056 bug reports).

Figure 56 – Bug reports categories: 1) Functional vs Non-Functional; 2) With or Without LR-Results.

fixed source code files. However, for some bugs, the AST extraction is impossible, even considering the patching over valid functional source code. This situation happens in bug 117526 from Eclipse Platform UI. Figure 57 shows the bug report for this bug.

The patch for the bug 117526 in Eclipse Platform UI is composed by changes in the next .JAVA files:

- In folder: bundles/org.eclipse.core.commands/src/
  - 4: org/eclipse/core/commands/Command.java
  - 5: org/eclipse/core/commands/INamedHandleStateIds.java
- In folder: bundles/org.eclipse.ui.workbench/Eclipse UI/

Project: Eclipse Platform UI
Bug-ID: 117526
Title: Bug 117526 [Contributions] [Commands] Javadoc warnings in N20051122-0010
Description:

```
/builds/N/src/plugins/org.eclipse.platform.doc.isv/
../org.eclipse.core.commands/
src/org/eclipse/core/commands/INamedHandleStateIds.java:
30: warning - Tag @link: Class or Package not found: NamedHandleObjectWithState

/builds/N/src/plugins/org.eclipse.platform.doc.isv/
../org.eclipse.core.commands/
src/org/eclipse/core/commands/Command.java:
581: warning - @param argument "state" is not a parameter name.

/builds/N/src/plugins/org.eclipse.platform.doc.isv/
../org.eclipse.jface/src/org/eclipse/jface/commands/RadioState.java:
46: warning - Tag @link: Class or Package not found: Boolean.TRUE

/builds/N/src/plugins/org.eclipse.platform.doc.isv/
../org.eclipse.jface/
src/org/eclipse/jface/commands/RadioState.java:
46: warning - Tag @link: Class or Package not found: Boolean.FALSE

/builds/N/src/plugins/org.eclipse.platform.doc.isv/
../org.eclipse.ui.workbench/Eclipse
    UI/org/eclipse/ui/handlers/RegistryRadioState.java:
51: warning - Tag @link: Class or Package not found: Boolean.FALSE
```

Figure 57 – *Bug Report* for the bug 117526 from Eclipse Platform UI.

  - 13: org/eclipse/ui/handlers/RegistryRadioState.java

- In folder: bundles/org.eclipse.jface/src/
  - 14: org/eclipse/jface/commands/RadioState.java

The patched files are also ranked in Ye et al. work (YE; BUNESCU; LIU, 2014), and we have shown the ranking positions above before the file paths. Without interpreting the bug report description and examining the patches, it is hard to detect we are dealing with a non-functional bug patch. Nevertheless, after comparing the buggy and the fixed files, it is clear that the patch applies only to the Javadoc lines, and functional code is not affected. Also, the automatic processing of this kind of bug is a challenge since it requires the system to deal with these particular cases. Figure 58 illustrates the changes in the files. Many other patches present the same or similar situations. Therefore, there is no AST generation for these patches, explaining the processing failure by ADD for these patches (or the absence of results).

In Chapter 6 we show that many patches in LR–dataset have exceptional properties, too far from most of the other dataset patches. For example, bug 60783 in the Eclipse

```
@@ -575,8 +575,9 @@
          * {@link IObjectWithState}.
          * </p>
          *
-         * @param state
-         *            The state to remove; must not be <code>null</code>.
+         * @param stateId
+         *            The identifier of the state to remove; must not be
+         *            <code>null</code>.
          */
         public void removeState(final String stateId) {
                 if (handler instanceof IObjectWithState) {
```

a) Command.java

```
@@ -13,7 +13,8 @@

 /**
  * <p>
- * State identifiers that are understood by {@link NamedHandleObjectWithState}.
+ * State identifiers that are understood by named handle objects that implement
+ * {@link IObjectWithState}.
  * </p>
  * <p>
  * Clients may implement or extend this class.
```

b) InamedHandleStateIds.java

```
@@ -22,7 +22,7 @@
  * A radio state that can be read from the registry. This stores a piece of
  * boolean state information that is grouped with other boolean state to form a
  * radio group. In a single radio group, there can be at most one state who
- * value is {@link Boolean#TRUE} all the others must be {@link Boolean.FALSE}.
+ * value is {@link Boolean#TRUE} all the others must be {@link Boolean#FALSE}.
  * </p>
  * <p>
  * When parsing from the registry, this state understands three parameters:
```

c) RegistryRadioState.java

```
@@ -23,8 +23,8 @@
 /**
  * <p>
  * A piece of boolean state grouped with other boolean states. Of these states,
- * only one may have a value of {@link Boolean.TRUE} at any given point in time.
- * The values of all other states must be {@link Boolean.FALSE}.
+ * only one may have a value of {@link Boolean#TRUE} at any given point in time.
+ * The values of all other states must be {@link Boolean#FALSE}.
  * </p>
  * <p>
  * If this state is registered using {@link IMenuStateIds#STYLE}, then it will
```

d) RadioState.java

Figure 58 – Patch for the bug 117526 in files from Eclipse project.

Platform UI project patches 16 files and involves 457 functional coding lines (168 added, 160 removed, 129 modified) distributed in 25 chunks. Another bug, 159857 in BIRT, exceeds other dimensions since it spreads in 66 chunks, with an accumulated distance between chunks of 882 lines (or 478 if we consider only non-empty code lines) and the

Table 17 – LR–dataset with and without outliers.

	#bugs reports	Outliers	No-Outliers
AspectJ	593	134 (22.6%)	459 (77.4%)
BIRT	4,178	948 (22.7%)	3,230 (77.3%)
Eclipse	6,495	1508 (23.2%)	4,987 (76.8%)
JDT	6,274	1342 (21.4%)	4,932 (78.6%)
SWT	4,151	938 (22.6%)	3,213 (77.4%)
Tomcat	1,056	234 (22.2%)	822 (77.8%)
Total	22,747	5,104 (22.4%)	17,643 (77.6%)

fixing of 46 files, 1 class, and 57 methods. We can find many other examples, including 1) bug 41254 in AspectJ (patches 75 files on 62 coding lines), 2) bug 47509 in JDT (patches 45 files on 261 coding lines), 3) bug 54426 in SWT (patches 8 files on 221 coding lines, spread in 30 chunks), 4) bug 49683 in Tomcat (patches 3 files on 159 coding-only lines, spread in 27 chunks). These exceptional patches would produce an artificial bias while evaluating some strategy over the dataset. To reduce the impact caused by these types of patches, we remove the outliers through the application of the discussed thresholds presented in Chapter 6 and enumerated next:

1. Patch size: from 1 to 60 lines
2. Chunks: from 1 to 20 chunks
3. Accumulated spreading between chunks: from 0 to 350 lines
4. Patched files: from 1 to 20 files

Table 17 shows the summary of patches for each project and the impact of the outlier removal. Even with the 22.4% overall reduction, many bugs (17,643) remain for the experiments and analysis.

Finally, considering all the five filtering criteria exposed in this subsection, Table 18 shows the relation between the bug reports in the original LR–dataset and two samplings set candidates for the experiment considering: 1. functional code only (FC); 2. mixed code (FC+NFC). For both sampling sets, we remove bugs and source code files in the following conditions: 1) fixed by testing files only; 2) files out of the project scope; 3) without baseline ranking results from Ye et al. study (YE; BUNESCU; LIU, 2014); 4) without results from ADD; 5) outliers. Considering FC and FC+NFC categories, the

Table 18 – LR–dataset sampling candidates.

	#bugs reports	Sampling candidates	
		1. FC	2. FC+NFC
AspectJ	593	135 (22.8%)	307 (51.8%)
BIRT	4,178	2,689 (64.4%)	2,919 (69.9%)
Eclipse	6,495	3,991 (61.5%)	4,459 (68.7%)
JDT	6,274	3,733 (59.5%)	4,400 (70.13%)
SWT	4,151	2,918 (70.3%)	2,922 (70.4%)
Tomcat	1,056	654 (61.9.2%)	761 (72.1%)
Total	22,747	14160 (62.25%)	15,768 (69.3%)

main difference is that FC+NFC can include patches fixed by testing code if at least one fixed file targets functional code.

7.2.2 Selected Settings

While our experimental package allows us to define many settings based on patch dimensions, the combinatorial explosion makes a comprehensive and complete analysis almost impractical. Therefore, we focused on some common repair patterns and their more common compositions of repair actions found in LR-dataset. The Chapter 6 details these common repair patterns composition. Additionally, the current state of the experimental package does not allow the processing of all the bug reports in LR–dataset, since some refactoring towards optimization is still required, and the time required for deep and complete processing is prohibitive. AspectJ and Tomcat have the smallest number of bugs in the LR–dataset. Therefore, we naturally choose these projects for the screening experiments and guide the selection and sampling over the other projects for comparison.

The detection of a repair pattern in a patch does not guarantee exclusivity for this pattern. So, different repair patterns in the same patch can imply a confounding factor and can produce some bias in the results and the analysis. Therefore, we should give special attention to the intersection between the samples and the co-occurrences of patterns to reduce these biases. As a mitigating strategy for this situation, we also tried to favor the selection (and the sampling) of bug patches where the repair pattern occurs exclusively or with a lower level of co-occurrences.

Table 19 – LR–dataset samples representativeness for AspectJ and Tomcat.

Repair Pattern	AspectJ #bugs	AspectJ Sample matches 1. FC	AspectJ Sample matches 2. FC+NFC	Tomcat #bugs	Tomcat Sample matches 1. FC	Tomcat Sample matches 2. FC+NFC
	593	164 (27.7%)	353 (59.5%)	1,056	752 (71.2%)	890 (84.3%)
codeMove	9	..	..	40	32 (80.0%)	36 (90.0%)
condBlockOthersAdd	137	28 (20.4%)	50 (36.5%)	187	50 (26.7%)	50 (26.7%)
condBlockRem	24	..	..	54	36 (66.7%)	46 (85.2%)
condBlockRetAdd	80	21 (26.3%)	50 (62.5%)	50	36 (72.0%)	48 (96.0%)
constChange	21	..	..	57	45 (78.9%)	50 (87.7%)
expLogicExpand	45	..	34 (75.6%)	61	46 (75.4%)	50 (82.0%)
expLogicMod	37	..	24 (64.9%)	52	40 (76.9%)	46 (88.5%)
missNullCheckN	53	24 (45.3%)	39 (73.6%)	53	45 (84.9%)	50 (94.3%)
missNullCheckP	33	..	27 (81.8%))	49	41 (83.7%)	46 (93.9%)
singleLine	89	26 (29.2%)	50 (56.2%)	208	50 (24.0%)	50 (24.0%)
wrapsIf	41	24 (58.5%)	31 (75.6%)	47	37 (78.7%)	44 (93.6%)
wrongMethodRef	66	..	40 (60.6%)	123	50 (40.7%)	50 (40.7%)
wrongVarRef	41	..	26 (63.4%)	90	50 (55.6%)	50 (55.6%)
Total sampled	459 (77.4%)	227 (38.3%)	294 (49.6%)	822 (77.8%)	557 (52.7%)	591 (56.0%)
Matched	395 (66.6%)	91 (15.3%)	230 (38.8%)	674 (63.8%)	367 (34.8%)	401 (38.0%)
Not Matched	64 (10.8%)	136 (22.9%)	64 (10.8%)	148 (14.0%)	190 (18.0%)	190 (18%)
Outliers	134 (22.6%)	35 (5.9%)	55 (9.3%)	234 (22.2%)	142 (13.5%)	173 (16.4%)
Not loaded by ADD	72 (12.1%)	6 (1.01%)	9 (1.52%)	100 (9.5%)	34 (3.2%)	44 (4.2%)

We covered most of the bugs found in AspectJ and Tomcat for the screening experiments. We focused the analysis on samples showing at least 20 to 30 bugs matched for each repair pattern. As an upper bound, we limited each sample to 50 bugs, hoping not to produce an unbalanced comparison between the bugs whose patches matched and those that did not match the repair patterns.

Table 19 show the representativeness of these sample sizes on the universe of bugs for each project segmented by repair pattern. Unfortunately, the FC category restricts the number of repair patterns alternatives because it does not provide enough bugs for some associated samples, especially for AspectJ. Therefore, we choose to run the screening experiments using the FC+NFC category.

7.2.3 Metrics Extracted and Hypothesis Tests

For each sample we extracted the metrics described in Section 2.3: MAP, MRR, Top-N for N in set {1, 5, 10}, and NDCG@k with k in the set {1, 5, 10}. We applied Statistical Non-Paired and Non-Parametric tests Mann-Whitney U to confirm if the found results have statistical significance (p-value = 0.05) and considering:

- H0 (=): The score ranking results ARE NOT SIGNIFICANTLY DIFFERENT when comparing samples of bugs with a matched repair pattern (or detected in the bug patches) and without this pattern.

- H1 (≠): The score ranking results ARE SIGNIFICANTLY DIFFERENT when comparing samples of bugs with a matched repair pattern (or detected in the bug patches) and without this pattern.

To define the impact caused by the repair patterns, we compute the difference between the metrics measures considering the samples where the repair pattern is *present* on the sample patches, m_p, against the correspondent measure of the *baseline*, m_b, i.e., when we do not differentiate when the pattern is present or absent, as defined in Equation 20.

$$d(p, b) = (m_p - m_b) * 100\ (\%) \tag{20}$$

We do the same with the measures for the samples where the repair pattern is *absent* in the patches, m_a, as defined in Equation 21.

$$d(a, b) = (m_a - m_b) * 100\ (\%) \tag{21}$$

7.2.4 Runtime Environment

We run the experiments on two servers with replicated Anaconda environment on Ubuntu 21.04 Operational System with the following essential software packages:

- Experimental package (Python and Java based software).
- Python 3.7.11
- Java 11 (OpenJDK 11.0.11)
- Maria DB 10.5.12
- Pony ORM 0.7.14
- ADD v1.0

The hardware settings were:

- Server 1, Lenovo ThinkServer TD340:

  CPU: 12-core Intel Xeon E5-2430 v2 @ 2.50 GHz

  GPU: GM206GL [Quadro M2000] NVIDIA

  RAM: 32 GB RAM (2 x 8GiB DIMM DDR3 1600 MHz 0.6 ns, 1x16 GiB DIMM DDR3 1600 MHz 0.6 ns)

  Hard Disk: 600 GB Seagate Savvio 10K.6 SAS 6GBS (ST600MM0006)

- Server 2, Cluster instance:

  CPU: 40 nodes of Intel Xeon E5620 @ 2.4GHz

  RAM: 20 GB RAM

  Hard Disk: 86 GB (Ext4 virtual partition)

7.3 Results

Here we present the obtained results that will support our analysis and answers to the research questions.

7.3.1 Screening of Repair Patterns

Considering FC+NFC bug patches for the AspectJ project, the comparisons between samples with matched versus not-matched repair patterns show some differences with statistical significance on the ranking score results for:

- ϕ_5: wrongVarRef, expLogicExpand, **wrongMethodRef**
- ϕ_7: **wrapsIf**, expLogicExpand, condBlockOthersAdd, **wrongMethodRef**, wrongVarRef
- ϕ_8: condBlockOthersAdd, **wrapsIf**, condBlockRetAdd, expLogicExpand, **wrongMethodRef**, wrongVarRef
- ϕ_{11}: singleLine, expLogicMod, condBlockOthersAdd, missNullCheckN
- ϕ_{15}: **wrapsIf**, **wrongMethodRef**
- ϕ_{19}: **wrongMethodRef**, singleLine, **wrapsIf**
- BLUiR: **wrapsIf**, singleLine, condBlockReturnAdd
- LR: condblockOthersAdd, **wrongMethodRef**

Considering FC+NFC bug patches for the Tomcat project, the comparisons between samples with matched versus not-matched repair patterns show some differences with statistical significance on the ranking score results for:

- ϕ_1: **missNullCheckN**, condBlockRem
- ϕ_4: **missNullCheckN**, expLogicExpand
- ϕ_5: **wrongVarRef**, codeMove, wrapsIf, expLogicMod, wrongMethodRef
- ϕ_6: **wrongVarRef**, missNullCheckP, wrongMethodRef
- ϕ_7: **missNullCheckN**, wrapsIf
- ϕ_8: **missNullCheckN**, wrapsIf, codeMove, expLogicMod
- ϕ_{11}: wrapsIf, expLogicMod, wrongMethodRef, codeMove, condBlockOthersAdd, expLogicExpand
- ϕ_{12}: wrapsIf, expLogicMod, **wrongVarRef**

- ❏ ϕ_{15}: missNullCheckP, constChange, **missNullCheckN**, wrongMethodRef, **wrongVarRef**
- ❏ ϕ_{16}: **missNullCheckN**, wrapsIf, **wrongVarRef**, expLogicMod
- ❏ ϕ_{17}: expLogicExpand, constChange, wrongMethodRef
- ❏ ϕ_{18}: **wrongVarRef**
- ❏ ϕ_{19}: **missNullCheckN**, singleLine, condBlockRem, wrongMethodRef
- ❏ BLUiR: **missNullCheckN**, wrapsIf, missNullCheckP, condBlockRetAdd
- ❏ LR: **missNullCheckN**

Next, we show the samples where we have some differences with statistical significance on the ranking score results for both AspectJ and Tomcat:

- ❏ ϕ_5: wrongVarRef, **wrongMethodRef**
- ❏ ϕ_7: **wrapsIf**
- ❏ ϕ_8: **wrapsIf**
- ❏ ϕ_{11}: expLogicMod, condBlockOthersAdd
- ❏ ϕ_{15}: **wrongMethodRef**
- ❏ ϕ_{19}: **wrongMethodRef**, singleLine
- ❏ BLUiR: **wrapsIf**, condBlockRetAdd

Some samples are bold considering their span across more score rankings: missNullCheckN and wrongVarRef for Tomcat; wrapsIf and wrongMethodRef for both AspectJ and Tomcat.

Based on the results with AspectJ and Tomcat, we decided to extend the analysis for Eclipse Platform UI and BIRT projects. However, to be viable, since the number of bugs for these projects is much higher and the processing time for all their bugs is prohibitive (with the actual implementation of the experimental package), we focus on the previously highlighted repair patterns. We will publish the complete analysis for Eclipse and BIRT in future studies and extend the studies for other repair patterns. Here we detail our results, especially for AspectJ and Tomcat.

7.3.2 Differences on ranking scores correlated to the repair pattern presence (or absence) and with statistical significance

We consider the hypothesis tests mentioned in Subsection 7.2.3 to verify the statistical significance of the score differences between samples with matched versus not-matched repair patterns. Table 20-a show the result of statistical test Mann-Whitney for each metric extracted from samples related to AspectJ. Each line contains the null hypothesis test result for a specific metric, considering the 19 features, BLUiR, and LR scores. The null hypothesis, H0, acceptance is represented by the $=$ symbol, while the $\neq$ symbol indicates the rejection. The results confirm the differences on some metrics for ϕ_4, ϕ_7, ϕ_8, ϕ_{15}, ϕ_{18}, ϕ_{19}, and BLUiR score rankings. Table 20-b shows the statistical tests for the wrapsIf in Tomcat. Similarly, the next tables present other results:

- ❑ Table 21: wrongMethodRef in Aspectj, Tomcat and BIRT;
- ❑ Table 22: wrongVariableRef in Aspectj, Tomcat and BIRT;
- ❑ Table 23: missNullCheckN in Aspectj, Tomcat and BIRT;

7.3.3 Impact of the differences correlated to the repair patterns

To give an idea about the level of impact caused by the repair pattern presence or absence, we compute the difference of the score results with the baseline, as defined in Equations 20 and 21. Table 24 shows the results for MAP measure. The samples associated with the repair patterns are in each column, starting with the results for the sample where the pattern is present (or matched in the sample patches), immediately followed by the results for the sample where the pattern is absent (or not matched in the sample patches). For example, the first column is related to the *Missing Not-Null Check* (MNC_N) repair pattern and next to the sample without this pattern represented with -MNC_N. Next columns are related to the repair patterns *Wraps with If* (W_If), *Wrong Method Reference* (WMR), *Wrong Variable Reference* (WVR). Only for sanity checking purposes, the last two columns represent the union of samples with the matched repair patterns followed by the union of the remaining patches without repair pattern matches. The color scale helps to highlight when the difference is positive (green tones, meaning the score is higher than the scores for the baseline), negative (red tones, meaning the score is below the baseline), or neutral (yellow tones, meaning the score is near the

Table 20 – *Wraps with If* in AspectJ and Tomcat, H0 result for Mann-Whitney (MW) test.

a) AspectJ

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$
MRR	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$
NDCG@1	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$
NDCG@5	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$\neq$	$=$
NDCG@10	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$
Top-1	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$
Top-5	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$\neq$	$=$
Top-10	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$

b) Tomcat

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	$=$	$=$	$\neq$	$=$	$\neq$	$=$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$=$	$\neq$	$=$	$=$	$=$	$\neq$	$=$
MRR	$=$	$=$	$\neq$	$=$	$\neq$	$=$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$=$	$\neq$	$=$	$=$	$=$	$\neq$	$=$
NDCG@1	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
NDCG@5	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
NDCG@10	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
Top-1	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
Top-5	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
Top-10	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$	$=$	$=$	$\neq$	$\neq$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$

baseline). We saturated the colors with the minimum (solid red), 0% (solid yellow), and maximum (solid green) shown in each table. The values are in the bold black font when there is statistical significance in the difference between the sample with the matched repair pattern and the sample not matched with the repair pattern, confirmed by null hypothesis rejection in at least one statistical test presented in Subsection 7.2.3. For the other metrics, we have:

- ❑ MRR score differences for AspectJ in Table 25-a and for Tomcat in Table 25-b;
- ❑ NDCG@1 score ranking differences for AspectJ in Table 26-a and for Tomcat in Table 26-b;
- ❑ NDCG@5 score ranking differences for AspectJ in Table 27-a and for Tomcat in Table 27-b;

Table 21 – *Wrong Method Reference* in AspectJ, Tomcat and BIRT: H0 result for Mann-Whitney (MW) test.

a) AspectJ

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	=	≠	=	≠	≠	=	=	=	=	=	=	=	=	=	=	=	=	=
MRR	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	=
NDCG@10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=
Top-5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	=
Top-10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	≠

b) Tomcat

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
MRR	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	≠	=	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	=	=	=	=	=
NDCG@10	=	=	=	=	≠	=	=	=	≠	=	≠	=	=	=	=	=	≠	=	=	=	=
Top-1	=	=	=	=	=	≠	=	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	≠	=	≠	=	=
Top-10	=	=	=	=	≠	=	=	=	≠	=	≠	=	=	=	=	=	≠	=	=	=	=

c) BIRT

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
MRR	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=

Table 22 – *Wrong Variable Reference* in AspectJ, H0 result for Mann-Whitney (MW) test.

a) AspectJ

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	=	≠	=	≠	≠	=	=	=	=	≠	=	=	=	=	=	=	=	=
MRR	=	=	=	=	≠	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=

b) Tomcat

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
MRR	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	≠	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	≠	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=

c) BIRT

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=
MRR	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	=	=	=

Table 23 – *Missing Not-Null Check* in AspectJ, H0 result for Mann-Whitney (MW) test.

a) AspectJ

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=
MRR	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	=	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=

b) Tomcat

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	≠	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	≠	=	=	=	≠	≠
MRR	≠	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	≠	=	=	=	≠	≠
NDCG@1	=	=	=	≠	=	=	≠	≠	=	=	=	=	=	=	≠	=	=	=	=	≠	=
NDCG@5	=	=	=	≠	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=	≠	≠	≠
NDCG@10	≠	=	=	≠	=	=	≠	≠	=	=	=	=	=	=	=	=	=	=	≠	≠	≠
Top-1	=	=	=	≠	=	=	≠	≠	=	=	=	=	=	=	≠	=	=	=	=	≠	=
Top-5	=	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	≠	≠	=
Top-10	≠	=	=	≠	=	=	≠	=	=	=	=	=	=	=	=	=	=	=	≠	=	=

c) BIRT

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}	ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}	ϕ_{16}	ϕ_{17}	ϕ_{18}	ϕ_{19}	BLUiR	LR
MAP	=	=	=	=	=	≠	=	=	=	=	=	=	=	=	≠	=	=	=	≠	=	=
MRR	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	≠	=	=
NDCG@1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
NDCG@10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-1	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-5	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Top-10	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=

- NDCG@10 score ranking differences for AspectJ in Table 28-a and for Tomcat in Table 28-b.
- Top-1 score ranking differences for AspectJ in Table 29-a and for Tomcat in Table 29-b;
- Top-5 score ranking differences for AspectJ in Table 30-a and for Tomcat in Table 30-b;
- Top-10 score ranking differences for AspectJ in Table 31-a and for Tomcat in Table 31-b.

7.3.4 Variation of the differences correlated to the repair patterns

Figure 59 shows the score rankings and highlights the range of the differences for the samples related to the wrapsIf pattern in AspectJ. The chart shows all 19 features from the LR approach, besides BLUiR and LR BL approaches. We consider three samples: 1) the sample where the repair pattern matches in patches (blue circle); 2) the sample where the repair pattern does not match in patches (red circle); 3) and the sample representing the baseline and containing the previous two samples, and samples with other repair patterns included in the experiment (black tick). In the baseline sample and similar to most of the previous BL approaches, the evaluations on the bug dataset do not differentiate matching or not-matching of repair patterns (or any other bug characteristic, discussed in Chapter 6). Figure 60 shows the score rankings for Tomcat analogous to what was shown for AspectJ. For the other repair patterns and projects, we have:

- *Wrong Method Reference* score ranking differences range for AspectJ in Figure 61 and for Tomcat in Figure 62;
- *Wrong Variable Reference* score ranking differences range for AspectJ in Figure 63 and for Tomcat in Figure 64;
- *Missing Not-Null Check* score ranking differences range for AspectJ in Figure 65 and for Tomcat in Figure 66.

Table 24 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the MAP scores.

a) MAP score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	2.00%	−0.60%	4.41%	[illegible]	0.52%	−1.82%	0.92%	4.60%	0.89%	−3.19%
f2	−0.65%	−0.01%	−0.07%	0.51%	−0.15%	1.29%	−0.67%	0.31%	−0.25%	0.90%
f3	**0.32%**	**−1.45%**	0.36%	[illegible]	−2.98%	−2.05%	[illegible]	3.29%	0.07%	−0.27%
f4	7.94%	2.85%	12.88%	−0.57%	−0.77%	[illegible]	1.60%	5.88%	1.91%	[illegible]
f5	−1.28%	−0.05%	[illegible]	[illegible]	**−0.36%**	**−1.62%**	**3.40%**	**0.86%**	−0.50%	1.78%
f6	4.25%	−0.28%	3.76%	−0.65%	−0.72%	0.48%	−0.62%	−0.94%	0.47%	−1.70%
f7	8.47%	3.05%	14.71%	−1.44%	**−1.91%**	**−5.19%**	**0.19%**	**4.51%**	1.67%	[illegible]
f8	7.99%	1.97%	14.92%	−0.37%	**−1.99%**	**−5.78%**	**6.99%**	**3.94%**	1.40%	[illegible]
f9	−0.31%	−0.41%	0.55%	0.67%	−0.95%	0.13%	2.96%	−0.64%	0.18%	−0.65%
f10	0.25%	0.09%	−0.66%	−0.45%	−0.11%	−0.07%	1.29%	−0.42%	−0.01%	0.02%
f11	3.94%	−2.79%	4.74%	−1.48%	−0.03%	2.22%	−3.04%	5.68%	0.07%	−0.26%
f12	−1.48%	−3.93%	2.33%	−0.77%	−1.18%	0.00%	−1.60%	0.65%	−0.02%	0.06%
f13	0.03%	0.40%	3.92%	1.87%	0.39%	0.52%	**−0.99%**	**−1.70%**	0.33%	−1.18%
f14	1.53%	−2.32%	0.89%	−0.78%	5.10%	−1.25%	2.31%	−0.55%	−0.13%	0.46%
f15	**2.29%**	**−0.86%**	2.15%	−1.07%	1.84%	−0.37%	−0.40%	1.06%	0.25%	−0.90%
f16	0.45%	−0.26%	0.65%	−0.19%	1.03%	−0.18%	−0.14%	0.74%	0.06%	−0.21%
f17	0.65%	−0.28%	0.93%	0.18%	0.16%	−0.26%	−0.02%	0.00%	0.15%	−0.55%
f18	1.01%	−0.51%	**1.51%**	**−0.46%**	2.23%	−0.26%	−0.18%	1.30%	0.18%	−0.64%
f19	−0.13%	0.99%	3.27%	[illegible]	5.93%	0.62%	−0.66%	0.28%	0.25%	−0.89%
BLUiR	8.75%	−0.81%	[illegible]	−0.41%	−0.36%	[illegible]	0.14%	7.84%	1.53%	[illegible]
LR	−3.52%	5.28%	3.33%	0.27%	3.47%	−1.37%	0.24%	5.07%	0.94%	−3.36%

b) MAP score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	**4.25%**	**−7.91%**	−3.50%	9.13%	8.57%	3.33%	−2.62%	0.57%	0.86%	−1.81%
f2	0.67%	−0.56%	−0.62%	−1.10%	−0.37%	−0.96%	0.80%	1.11%	−0.02%	0.03%
f3	5.44%	−1.05%	**−3.49%**	**1.50%**	−3.14%	−7.60%	5.20%	1.81%	0.26%	−0.55%
f4	**23.98%**	**−6.95%**	7.72%	14.48%	−1.61%	3.00%	1.87%	5.07%	−1.00%	2.12%
f5	3.98%	1.80%	**0.22%**	**−0.54%**	−0.27%	−1.70%	**−1.91%**	**2.79%**	0.05%	−0.11%
f6	−1.58%	−0.43%	5.29%	1.43%	2.93%	−3.53%	1.75%	−1.33%	−0.06%	0.12%
f7	**23.81%**	**−6.17%**	**5.02%**	**15.51%**	−1.27%	0.81%	1.84%	5.15%	−0.96%	2.03%
f8	10.09%	−2.41%	**−0.94%**	**11.79%**	2.03%	0.01%	4.45%	1.28%	0.37%	−0.78%
f9	−1.16%	−0.37%	**−0.02%**	**−1.04%**	2.15%	−1.20%	1.76%	−1.70%	0.27%	−0.58%
f10	−0.62%	−2.26%	**−2.30%**	**2.98%**	6.43%	0.28%	−0.53%	0.66%	0.33%	−0.69%
f11	1.01%	5.72%	**−1.44%**	**9.73%**	1.81%	−6.20%	−1.06%	2.83%	−0.27%	0.57%
f12	−1.66%	−1.39%	**−5.13%**	**4.56%**	−0.66%	−2.33%	−1.40%	3.97%	−0.71%	1.50%
f13	0.44%	−2.11%	**−3.19%**	**−2.06%**	−2.06%	1.13%	2.17%	2.30%	0.40%	−0.85%
f14	0.03%	−3.15%	**−3.37%**	**1.68%**	3.09%	0.99%	0.24%	5.58%	−0.45%	0.95%
f15	−3.67%	1.80%	2.92%	2.92%	3.92%	−1.51%	−1.07%	−1.21%	−0.47%	1.00%
f16	**−0.36%**	**0.28%**	**−0.17%**	**0.30%**	−0.02%	−0.13%	−0.31%	−0.22%	−0.09%	0.20%
f17	−1.20%	−0.54%	0.69%	0.69%	1.39%	−0.83%	−0.11%	−0.89%	−0.17%	0.36%
f18	−0.44%	0.47%	−0.21%	0.45%	−0.05%	−0.23%	−0.30%	−0.22%	−0.11%	0.23%
f19	−3.05%	0.68%	0.74%	5.01%	0.41%	−0.98%	−1.51%	−0.69%	−0.37%	0.78%
BLUiR	**22.75%**	**−7.91%**	**1.32%**	**14.74%**	3.12%	4.82%	7.12%	2.84%	0.18%	−0.39%
LR	13.38%	−4.85%	6.70%	6.86%	8.03%	5.28%	−5.23%	7.20%	0.55%	−1.16%

7.4 Analysis

Next, we analyze and discuss the results presented in the previous section. We analyzed the results for the patterns *Wraps with If*, *Wrong Method Reference*, *Wrong Variable Reference*, and *Missing Not-Null Check*. Overall we can find H0 rejection for metrics in all of them. Next, we present the analysis for each one.

Table 25 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the MRR scores.

a) MRR score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	0.64%	−1.95%	4.25%	[illegible]	1.93%	−2.78%	0.96%	4.76%	0.42%	−1.49%
f2	−0.73%	−0.07%	−0.13%	0.58%	−0.07%	1.09%	−0.61%	0.18%	−0.17%	0.61%
f3	−0.31%	−1.32%	0.74%	−1.73%	−2.17%	−2.41%	[illegible]	2.55%	0.19%	−0.67%
f4	6.36%	2.60%	11.22%	[illegible]	0.10%	[illegible]	1.55%	5.46%	1.72%	[illegible]
f5	−1.90%	−0.65%	−2.09%	[illegible]	**0.85%**	**−2.43%**	**3.43%**	**0.22%**	−0.38%	1.37%
f6	5.23%	−0.06%	2.98%	0.18%	−0.69%	0.52%	−1.33%	−1.24%	0.60%	−2.16%
f7	6.95%	2.99%	13.18%	[illegible]	−1.41%	[illegible]	0.57%	3.65%	1.54%	[illegible]
f8	6.90%	0.85%	13.83%	−1.33%	−1.87%	[illegible]	5.99%	3.22%	1.27%	−4.58%
f9	−0.59%	0.37%	1.97%	0.52%	−1.01%	1.14%	2.55%	−0.89%	0.23%	−0.84%
f10	0.32%	0.06%	−0.59%	−0.41%	0.04%	−0.19%	1.14%	−0.37%	0.02%	−0.09%
f11	2.66%	−2.86%	6.96%	[illegible]	−0.71%	2.58%	−3.13%	4.67%	−0.17%	0.62%
f12	−1.63%	−4.15%	2.56%	−0.57%	−0.77%	−0.29%	−1.88%	0.48%	−0.03%	0.12%
f13	−0.55%	−0.13%	5.11%	3.10%	2.14%	−0.11%	**−1.52%**	**−2.30%**	0.41%	−1.49%
f14	0.67%	−3.14%	0.33%	−1.14%	5.47%	−1.62%	1.52%	−0.48%	−0.38%	1.35%
f15	3.06%	−0.90%	2.00%	−1.11%	3.06%	−1.01%	−0.38%	1.53%	0.36%	−1.30%
f16	0.24%	−0.34%	0.40%	−0.17%	0.85%	0.28%	−0.18%	0.51%	−0.05%	0.19%
f17	0.78%	−0.28%	0.63%	0.45%	0.55%	−0.34%	−0.29%	−0.13%	0.02%	−0.08%
f18	0.68%	−0.62%	**1.13%**	**−0.36%**	2.06%	−0.23%	−0.50%	0.98%	0.11%	−0.41%
f19	−0.62%	0.80%	4.02%	[illegible]	9.18%	−0.37%	0.53%	0.35%	0.48%	−1.72%
BLUiR	6.51%	−2.53%	[illegible]	[illegible]	0.49%	[illegible]	−1.85%	7.53%	0.96%	−3.45%
LR	[illegible]	2.23%	1.46%	0.12%	**4.47%**	**−6.96%**	−3.38%	−3.80%	0.52%	−1.86%

b) MRR score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	**2.66%**	**−7.89%**	[illegible]	8.28%	[illegible]	2.99%	−1.15%	−1.20%	1.14%	[illegible]
f2	−0.08%	−1.19%	−1.18%	−1.64%	−0.30%	−1.55%	2.06%	0.50%	0.11%	−0.24%
f3	3.89%	2.19%	**−4.54%**	**−0.02%**	−3.94%	[illegible]	5.64%	2.78%	0.44%	−0.94%
f4	**23.23%**	**−8.75%**	6.32%	[illegible]	−1.52%	1.91%	2.08%	3.64%	−1.13%	[illegible]
f5	3.72%	1.96%	**−0.03%**	**−0.77%**	−0.14%	−1.90%	**−2.14%**	**2.63%**	0.15%	−0.31%
f6	−3.06%	−1.51%	5.21%	3.44%	1.86%	−5.61%	3.29%	−1.76%	−0.36%	0.75%
f7	**22.99%**	**−7.72%**	**3.65%**	**13.00%**	−1.74%	0.84%	2.36%	3.71%	−1.25%	[illegible]
f8	9.36%	−3.42%	**−0.92%**	**10.96%**	3.70%	0.01%	5.19%	0.18%	0.29%	−0.60%
f9	−1.62%	0.61%	**−0.48%**	**−1.42%**	2.19%	−1.57%	1.39%	−2.11%	0.36%	−0.77%
f10	−1.26%	−2.23%	**−2.77%**	**3.26%**	6.73%	−0.65%	−0.59%	0.07%	0.37%	−0.79%
f11	0.22%	6.70%	**−2.12%**	**10.35%**	1.44%	[illegible]	0.22%	2.50%	−0.40%	0.84%
f12	−1.13%	0.68%	**−5.60%**	**4.07%**	−1.06%	−2.52%	−0.39%	3.50%	−0.82%	1.73%
f13	−0.09%	−0.91%	**−3.56%**	**−2.46%**	−2.54%	0.80%	1.75%	1.94%	0.35%	−0.75%
f14	−1.31%	0.70%	**−3.71%**	**1.46%**	1.66%	0.62%	0.85%	5.12%	−0.56%	1.19%
f15	−5.32%	1.96%	2.61%	4.50%	3.13%	−3.18%	0.33%	−1.88%	−0.74%	1.58%
f16	**−0.54%**	**0.21%**	**−0.28%**	**0.38%**	−0.08%	−0.16%	−0.37%	−0.36%	−0.18%	0.38%
f17	−1.67%	−0.53%	0.52%	1.00%	1.00%	−1.30%	0.54%	−1.07%	−0.28%	0.58%
f18	−0.64%	0.43%	−0.35%	0.56%	−0.14%	−0.31%	−0.33%	−0.38%	−0.20%	0.43%
f19	−4.39%	0.94%	0.00%	6.70%	−0.25%	−2.36%	−0.35%	−1.10%	−0.62%	1.32%
BLUiR	**21.52%**	**−7.23%**	**−0.67%**	**12.92%**	3.35%	2.61%	[illegible]	1.64%	0.12%	−0.24%
LR	**9.69%**	**−4.16%**	5.13%	6.83%	5.23%	2.68%	[illegible]	4.03%	0.49%	−1.04%

7.4.1 Wraps with If Repair Pattern

In AspectJ, *Wraps with If* samples present many differences in the scores for both cases (with and without the repair pattern), that is evidenced in Figure 59, but also in Table 24-a to Table 31-a. For most features and also for the BLUiR, when *Wraps with If* is present, we have higher scores than in the baseline. On the other hand, the

Table 26 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@1 scores.

a) NDCG@1 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	−0.48%	−0.17%	3.49%	−2.17%	−0.79%	−4.17%	1.88%	3.45%	0.60%	−2.17%
f2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f3	−0.89%	−0.02%	−0.82%	[illegible]	−1.02%	−3.24%	7.89%	3.98%	−0.06%	0.23%
f4	7.55%	2.83%	12.16%	−0.40%	−0.52%	[illegible]	−0.35%	6.28%	1.89%	[illegible]
f5	−1.55%	0.45%	−1.55%	−1.55%	−0.02%	−1.55%	2.30%	0.45%	−0.44%	1.58%
f6	1.36%	−0.21%	−0.21%	1.02%	−0.21%	1.02%	−0.21%	−0.21%	0.06%	−0.21%
f7	[illegible]	2.52%	**13.87%**	**−2.71%**	−2.18%	−4.56%	−0.15%	3.29%	1.52%	−5.47%
f8	8.05%	2.96%	**15.54%**	**−1.04%**	−2.04%	−5.04%	6.86%	2.96%	1.42%	−5.11%
f9	−0.89%	0.34%	1.09%	1.11%	−0.89%	0.34%	2.96%	−0.89%	0.25%	−0.89%
f10	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f11	−1.62%	−0.96%	2.99%	−2.19%	0.81%	3.04%	−1.83%	3.81%	−0.34%	1.23%
f12	−2.59%	−2.59%	0.64%	−1.36%	−1.06%	−0.59%	−2.59%	−0.59%	−0.15%	0.54%
f13	−0.97%	0.26%	2.99%	2.26%	0.57%	0.26%	−0.97%	−0.97%	0.27%	−0.97%
f14	0.81%	−1.75%	0.22%	−1.75%	**4.78%**	**−1.75%**	0.60%	−0.82%	0.18%	−0.65%
f15	0.88%	−0.32%	−0.32%	−0.32%	0.85%	−0.32%	−0.32%	0.62%	0.09%	−0.32%
f16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f18	−0.34%	−0.34%	−0.34%	−0.34%	2.16%	−0.34%	−0.34%	1.66%	0.09%	−0.34%
f19	−1.42%	2.58%	**3.79%**	**−1.42%**	5.32%	0.58%	0.94%	1.52%	0.39%	−1.42%
BLUiR	7.52%	−0.43%	**15.38%**	**−0.43%**	−0.55%	[illegible]	−0.38%	7.19%	1.43%	−5.13%
LR	−4.78%	6.54%	2.06%	−2.02%	0.33%	−3.90%	−3.41%	4.89%	1.31%	−4.70%

b) NDCG@1 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	2.64%	[illegible]	**−5.01%**	**10.30%**	7.78%	3.37%	−2.80%	0.92%	1.17%	−2.47%
f2	−0.55%	−0.55%	−0.55%	−0.55%	−0.55%	−0.55%	−0.08%	−0.55%	−0.10%	0.22%
f3	4.49%	1.19%	−3.34%	1.57%	−2.43%	[illegible]	4.04%	2.51%	0.10%	−0.21%
f4	**25.36%**	**−7.70%**	9.28%	11.52%	−0.84%	2.75%	3.20%	4.75%	−0.84%	1.77%
f5	3.35%	−1.09%	1.18%	−1.09%	−1.09%	−1.09%	−1.09%	0.91%	0.02%	−0.04%
f6	−2.57%	1.43%	5.32%	0.84%	**3.43%**	**−2.57%**	−1.89%	−1.34%	−0.42%	0.88%
f7	**23.21%**	**−6.37%**	6.02%	[illegible]	−1.15%	−0.21%	−0.70%	[illegible]	−1.30%	2.75%
f8	**11.85%**	**−2.15%**	0.06%	9.85%	1.49%	1.07%	3.07%	−0.15%	0.48%	−1.02%
f9	−0.86%	0.37%	1.41%	−0.86%	1.14%	−0.86%	1.14%	−0.86%	0.16%	−0.33%
f10	2.42%	−1.02%	−1.97%	0.54%	**5.76%**	**−4.24%**	−0.24%	−0.24%	1.11%	−2.34%
f11	1.25%	3.81%	−0.87%	7.19%	**0.59%**	**−5.41%**	1.19%	2.59%	−0.39%	0.83%
f12	−2.47%	−0.22%	−3.83%	2.17%	−1.83%	−0.61%	−1.38%	2.17%	−0.92%	1.93%
f13	0.22%	−1.32%	−2.00%	−2.00%	−2.00%	0.00%	2.00%	2.00%	0.36%	−0.77%
f14	−1.10%	−1.99%	−3.22%	0.23%	2.23%	1.46%	1.46%	[illegible]	−0.40%	0.84%
f15	**−3.06%**	**2.16%**	2.55%	2.35%	2.94%	−1.06%	−2.38%	−1.84%	−0.68%	1.44%
f16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f17	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	0.36%
f18	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f19	−2.05%	−0.83%	0.22%	5.36%	−2.05%	−0.05%	−1.37%	−0.83%	−0.30%	0.63%
BLUiR	**27.22%**	**−9.35%**	3.41%	12.75%	3.61%	1.97%	6.43%	3.97%	0.16%	−0.34%
LR	14.57%	−5.60%	8.52%	8.11%	6.19%	4.19%	[illegible]	[illegible]	0.06%	−0.12%

sample without *Wraps with If* show scores below or near to the baseline. Considering higher scores, most visible and consistent variations between metrics against baseline occur with ϕ_4 (+13.16% in NDCG@5 when the pattern is present, and −2.93% in Top-5 and Top-10 when the pattern is absent), ϕ_7 (+13.87% in NDCG@1 when present, −3.86% in Top-1 when absent), ϕ_8 (+15.54% in NDCG@1 when present, −1.82% in

Table 27 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@1 scores.

a) NDCG@5 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	2.11%	−0.89%	5.55%	[illegible]	1.18%	−0.52%	0.57%	4.88%	0.99%	−3.56%
f2	−0.38%	−0.38%	−0.38%	0.88%	−0.38%	0.94%	−0.38%	0.39%	−0.18%	0.65%
f3	1.15%	−1.31%	1.22%	−1.12%	−3.30%	−1.58%	[illegible]	3.07%	0.29%	−1.03%
f4	8.19%	2.96%	**13.16%**	**−1.04%**	−0.62%	[illegible]	1.68%	6.07%	2.03%	[illegible]
f5	−1.90%	0.10%	[illegible]	[illegible]	0.12%	−1.90%	**3.61%**	**0.10%**	−0.47%	1.69%
f6	5.55%	0.36%	4.56%	−1.55%	−2.44%	−0.67%	−1.23%	−1.12%	0.53%	−1.89%
f7	8.01%	3.32%	**14.68%**	**−1.42%**	−1.62%	[illegible]	0.11%	4.75%	1.76%	[illegible]
f8	7.60%	1.71%	13.52%	−0.20%	−1.69%	[illegible]	6.32%	4.18%	1.42%	−5.11%
f9	−0.12%	0.12%	0.87%	0.89%	−1.11%	0.64%	2.74%	−1.11%	0.31%	−1.11%
f10	0.75%	0.47%	−0.53%	−0.53%	−0.53%	−0.53%	1.66%	−0.53%	0.15%	−0.53%
f11	**5.52%**	**−2.76%**	6.12%	[illegible]	−1.92%	2.59%	−2.89%	5.90%	0.21%	−0.75%
f12	−1.59%	−4.20%	3.16%	−0.59%	−0.65%	−0.17%	−1.77%	−0.20%	0.08%	−0.29%
f13	0.67%	0.06%	4.51%	1.71%	1.09%	0.06%	−1.94%	−1.94%	0.44%	−1.57%
f14	2.45%	−1.85%	1.01%	−1.58%	5.43%	−1.65%	2.21%	−0.89%	−0.16%	0.58%
f15	1.64%	−0.67%	**2.18%**	**−1.45%**	**2.47%**	**−1.45%**	−0.53%	1.82%	0.30%	−1.08%
f16	0.73%	−0.37%	1.02%	−0.37%	1.20%	−0.30%	−0.37%	0.89%	0.09%	−0.31%
f17	0.83%	−0.42%	0.57%	0.62%	−0.68%	−0.68%	−0.68%	0.62%	0.05%	−0.19%
f18	0.96%	−0.65%	1.38%	−0.25%	1.85%	−0.65%	−0.65%	1.35%	0.18%	−0.65%
f19	−0.75%	0.99%	3.03%	−1.38%	**7.26%**	**−0.58%**	−1.43%	1.07%	0.55%	−1.96%
BLUiR	8.16%	−1.45%	**16.92%**	**−1.69%**	−0.19%	[illegible]	−0.99%	8.35%	1.65%	[illegible]
LR	−0.66%	3.26%	5.55%	−0.63%	0.00%	−2.95%	−1.71%	4.65%	1.33%	−4.78%

b) NDCG@5 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	3.79%	[illegible]	−4.74%	8.53%	9.87%	3.71%	−3.47%	2.41%	0.50%	−1.06%
f2	0.98%	−0.65%	−0.55%	−1.43%	−0.47%	−1.43%	1.35%	1.87%	0.06%	−0.12%
f3	4.87%	−1.08%	−4.53%	1.08%	−2.94%	[illegible]	5.77%	1.76%	0.27%	−0.58%
f4	**23.56%**	**−7.14%**	6.33%	14.74%	−2.16%	3.27%	1.01%	5.21%	−1.35%	2.84%
f5	3.90%	2.43%	0.32%	−0.69%	−0.16%	−1.95%	**−1.95%**	**3.35%**	0.01%	−0.02%
f6	−1.87%	−1.53%	6.51%	1.19%	2.91%	−3.90%	**5.09%**	**−2.18%**	−0.07%	0.14%
f7	**24.12%**	**−6.25%**	4.78%	16.55%	−1.48%	1.32%	2.76%	4.26%	−0.99%	2.10%
f8	**9.90%**	**−3.02%**	−0.39%	12.19%	1.31%	−0.11%	6.56%	2.47%	0.39%	−0.83%
f9	−0.78%	−1.11%	−0.33%	−0.83%	2.30%	−1.34%	**1.40%**	**−2.60%**	0.34%	−0.72%
f10	−0.82%	−3.59%	−1.73%	4.04%	5.85%	0.65%	−1.26%	1.43%	−0.02%	0.03%
f11	1.76%	7.50%	**−1.96%**	**10.00%**	**2.80%**	**−6.22%**	−2.16%	2.48%	−0.20%	0.42%
f12	−0.45%	−1.12%	**−5.23%**	**6.67%**	−1.41%	−2.85%	−1.62%	4.50%	−0.86%	1.82%
f13	−0.08%	−2.31%	−3.85%	−2.12%	−1.84%	1.28%	1.54%	2.93%	0.53%	−1.11%
f14	0.07%	−2.53%	−3.39%	1.23%	4.17%	−0.04%	−1.42%	5.94%	−0.15%	0.32%
f15	−3.88%	1.70%	2.61%	3.71%	**4.24%**	**−2.28%**	−0.14%	−1.02%	−0.45%	0.94%
f16	−0.07%	−0.07%	−0.07%	−0.07%	−0.07%	−0.07%	−0.07%	−0.07%	−0.07%	0.14%
f17	−1.53%	−0.24%	1.88%	1.59%	2.13%	−1.49%	−0.03%	−1.10%	−0.38%	0.80%
f18	−0.11%	−0.11%	−0.11%	0.07%	−0.11%	−0.11%	−0.11%	−0.11%	−0.11%	0.24%
f19	**−4.28%**	**1.37%**	0.10%	5.32%	0.63%	−2.28%	−1.23%	−0.42%	−0.30%	0.62%
BLUiR	**21.91%**	**−9.21%**	1.52%	[illegible]	2.78%	7.15%	7.82%	2.86%	0.12%	−0.25%
LR	**12.32%**	**−5.51%**	6.07%	8.16%	8.35%	5.35%	−2.45%	6.49%	0.51%	−1.08%

Top-1 when absent), and BLUiR (+18.98% in Top-5 when present, −3.73% in Top-5 when absent). Other features like ϕ_1, ϕ_6, ϕ_{11}, ϕ_{12}, ϕ_{15}, ϕ_{19} are generally consistent with higher results than baseline in the presence of the pattern, but in a smaller range. The remaining features and the LR approach have minimal or inconsistent differences from the baseline. According to Table 20, not all the results confirm statistical difference, but

Table 28 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the NDCG@10 scores.

a) NDCG@10 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	2.20%	-1.17%	4.79%	[illegible]	1.81%	-1.67%	0.97%	4.21%	1.02%	-3.68%
f2	-0.68%	0.70%	0.47%	0.58%	-0.68%	1.55%	-0.68%	0.81%	-0.18%	0.64%
f3	1.22%	-2.14%	0.52%	-1.62%	-4.71%	-1.92%	[illegible]	3.09%	0.06%	-0.23%
f4	8.19%	2.96%	**13.16%**	**-1.04%**	-0.62%	[illegible]	1.68%	6.07%	2.03%	[illegible]
f5	-0.61%	-0.32%	-1.24%	[illegible]	-0.30%	-2.32%	4.03%	1.73%	-0.35%	1.27%
f6	5.56%	-0.17%	4.82%	-0.75%	-1.73%	0.49%	0.61%	-0.65%	0.68%	-2.45%
f7	7.97%	3.28%	**14.63%**	**-1.46%**	-1.36%	[illegible]	0.07%	4.95%	1.77%	[illegible]
f8	7.48%	1.59%	**14.55%**	**-0.32%**	-1.81%	[illegible]	6.20%	4.78%	1.45%	[illegible]
f9	0.44%	-0.24%	0.51%	0.64%	-0.58%	0.29%	2.38%	-0.80%	0.26%	-0.94%
f10	0.75%	0.47%	-0.53%	-0.53%	-0.53%	-0.53%	1.66%	-0.53%	0.15%	-0.53%
f11	4.97%	-3.73%	5.73%	-0.42%	-0.06%	0.97%	-3.63%	6.24%	0.11%	-0.40%
f12	-1.67%	-4.45%	2.31%	-0.77%	-1.50%	0.25%	-2.62%	1.72%	0.00%	0.01%
f13	0.50%	0.49%	4.97%	2.14%	0.92%	-0.11%	-2.11%	-2.11%	0.35%	-1.27%
f14	2.26%	-2.56%	0.29%	0.56%	4.85%	-1.27%	4.67%	-0.29%	-0.10%	0.37%
f15	3.97%	-0.33%	**4.01%**	**-1.26%**	**3.28%**	**-1.25%**	0.72%	2.48%	0.62%	-2.25%
f16	0.66%	-0.44%	0.95%	-0.16%	1.14%	-0.36%	-0.44%	0.82%	0.11%	-0.38%
f17	1.03%	-0.91%	0.99%	1.16%	0.64%	-0.94%	1.15%	-0.48%	0.23%	-0.84%
f18	0.63%	-0.69%	1.05%	-0.01%	2.23%	-0.04%	0.17%	1.01%	0.18%	-0.65%
f19	1.68%	0.70%	2.52%	[illegible]	**7.89%**	**-0.19%**	-0.56%	0.40%	0.55%	-1.97%
BLUiR	9.03%	-1.24%	[illegible]	-0.17%	-0.81%	[illegible]	-0.26%	8.32%	1.25%	-4.40%
LR	-2.59%	4.55%	5.04%	[illegible]	0.76%	-2.26%	2.48%	5.61%	1.02%	-3.68%

b) NDCG@10 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	**4.71%**	**-10.02%**	0.31%	7.46%	9.42%	3.45%	[illegible]	0.48%	0.88%	-1.85%
f2	1.21%	-0.44%	-0.86%	-1.93%	-0.26%	-1.33%	1.05%	1.53%	0.18%	-0.38%
f3	5.41%	-2.06%	-2.92%	1.35%	-2.87%	[illegible]	5.38%	3.27%	0.43%	-0.90%
f4	**23.15%**	**-6.78%**	6.22%	[illegible]	-1.86%	2.85%	1.72%	5.37%	-1.21%	2.56%
f5	4.37%	2.98%	0.10%	-0.28%	**0.45%**	**-2.18%**	**-2.18%**	**3.12%**	0.07%	-0.15%
f6	-1.07%	-1.54%	5.14%	2.33%	1.54%	-3.96%	**4.54%**	**-1.82%**	0.04%	-0.09%
f7	**24.04%**	**-6.33%**	4.70%	[illegible]	-1.56%	1.57%	2.68%	4.18%	-1.02%	2.15%
f8	**9.19%**	**-4.18%**	**-1.09%**	**12.56%**	3.22%	-0.46%	5.81%	1.08%	0.18%	-0.39%
f9	-1.40%	-0.46%	-0.29%	-1.44%	**1.80%**	**-1.96%**	2.16%	-1.97%	0.48%	-1.02%
f10	-1.93%	-3.00%	-2.52%	4.59%	7.78%	1.39%	-0.47%	1.38%	0.04%	-0.08%
f11	0.54%	5.72%	**-2.40%**	**11.85%**	**3.56%**	**-6.82%**	-1.50%	3.11%	-0.32%	0.68%
f12	-1.28%	-1.26%	**-6.05%**	**5.38%**	-0.50%	-3.74%	-1.61%	3.74%	-0.66%	1.40%
f13	-0.58%	-2.55%	-3.58%	-2.62%	-2.34%	1.45%	1.71%	2.43%	0.40%	-0.84%
f14	-0.66%	-3.47%	-4.06%	2.62%	2.61%	1.63%	0.28%	5.71%	-0.52%	1.10%
f15	-4.15%	1.43%	2.67%	3.44%	4.57%	-1.97%	-0.41%	-1.29%	-0.39%	0.83%
f16	-0.16%	-0.16%	-0.16%	0.01%	-0.16%	0.25%	-0.16%	-0.16%	-0.13%	0.28%
f17	-2.20%	-0.91%	1.90%	1.14%	**2.84%**	**-1.45%**	0.23%	-1.78%	-0.32%	0.68%
f18	-0.24%	0.43%	-0.24%	-0.06%	-0.24%	0.13%	-0.24%	-0.24%	-0.21%	0.45%
f19	**-3.49%**	**2.10%**	0.80%	4.76%	0.67%	-1.72%	-1.08%	0.05%	-0.42%	0.89%
BLUiR	**20.60%**	**-6.80%**	0.57%	[illegible]	1.65%	5.68%	[illegible]	3.13%	0.04%	-0.09%
LR	12.02%	-3.55%	5.33%	7.36%	9.13%	6.20%	[illegible]	7.52%	0.52%	-1.09%

we have the H0 rejection for some metrics in the scores for ϕ_4, ϕ_7, ϕ_8, ϕ_{15}, ϕ_{19}, BLUiR, many of them between those we just highlighted.

From the obtained results, we have some evidence that bugs that require the *Wraps with If* repair pattern would be easier to find than bugs that would not require this type of patch on the samples for the AspectJ dataset using the BLUiR approach. However,

Table 29 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-1 scores.

a) Top-1 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	−1.83%	−1.52%	3.38%	[illegible]	0.48%	[illegible]	2.01%	4.48%	0.04%	−0.15%
f2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f3	−1.67%	−0.80%	−0.35%	[illegible]	−1.80%	−2.80%	[illegible]	3.20%	0.15%	−0.55%
f4	6.04%	2.10%	10.68%	−1.90%	0.60%	[illegible]	−0.37%	6.10%	1.57%	[illegible]
f5	−2.04%	−0.04%	−2.04%	−2.04%	0.46%	−2.04%	1.81%	−0.04%	−0.30%	1.08%
f6	2.22%	−0.34%	−0.34%	1.66%	**−0.34%**	**1.66%**	−0.34%	−0.34%	0.09%	−0.34%
f7	[illegible]	2.14%	**12.72%**	**−3.86%**	−2.36%	−3.86%	1.67%	2.14%	1.44%	−5.18%
f8	[illegible]	2.18%	**14.76%**	**−1.82%**	−2.82%	[illegible]	7.56%	2.18%	1.31%	−4.70%
f9	−1.02%	0.98%	2.21%	0.98%	−1.02%	0.98%	2.83%	−1.02%	0.28%	−1.02%
f10	0.00%	0.00%	0.00%	0.00%	**0.00%**	**0.00%**	0.00%	0.00%	0.00%	0.00%
f11	−2.20%	−0.76%	4.92%	[illegible]	**0.24%**	**3.24%**	−0.92%	3.24%	−0.41%	1.49%
f12	−2.72%	−2.72%	0.50%	−0.72%	−0.22%	−0.72%	−2.72%	−0.72%	−0.11%	0.40%
f13	−1.36%	0.64%	5.09%	2.64%	1.14%	0.64%	−1.36%	−1.36%	0.38%	−1.36%
f14	−0.16%	−2.72%	0.50%	−2.72%	**4.78%**	**−2.72%**	1.13%	−0.72%	−0.11%	0.40%
f15	1.88%	−0.68%	−0.68%	−0.68%	1.82%	−0.68%	−0.68%	1.32%	0.19%	−0.68%
f16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f17	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f18	−0.34%	−0.34%	−0.34%	−0.34%	2.16%	−0.34%	−0.34%	1.66%	0.09%	−0.34%
f19	−2.04%	1.96%	**4.41%**	**−2.04%**	7.96%	−0.04%	1.81%	1.96%	0.57%	−2.04%
BLUiR	5.70%	−2.24%	**13.56%**	**−2.24%**	0.26%	[illegible]	−0.71%	7.76%	0.80%	−2.87%
LR	[illegible]	4.71%	0.97%	−1.29%	3.71%	[illegible]	−4.37%	−3.29%	0.45%	−1.61%

b) Top-1 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	1.96%	[illegible]	**−6.52%**	**9.30%**	9.30%	3.30%	−0.70%	−0.70%	1.73%	−3.65%
f2	−1.02%	−1.02%	−1.02%	−1.02%	−1.02%	−1.02%	0.98%	−1.02%	−0.02%	0.04%
f3	4.07%	4.29%	−4.61%	0.29%	−3.71%	[illegible]	4.29%	2.29%	0.51%	−1.07%
f4	**24.76%**	**−8.80%**	7.84%	9.20%	−0.80%	1.20%	3.20%	3.20%	−0.87%	1.84%
f5	3.26%	−1.18%	1.09%	−1.18%	−1.18%	−1.18%	−1.18%	0.82%	0.06%	−0.13%
f6	−3.55%	0.45%	5.54%	2.45%	**2.45%**	**−3.55%**	−1.55%	−1.55%	−0.81%	1.71%
f7	**22.45%**	**−8.00%**	5.28%	[illegible]	−2.00%	0.00%	0.00%	[illegible]	−1.55%	3.27%
f8	**11.20%**	**−2.80%**	0.29%	9.20%	3.20%	1.20%	3.20%	−0.80%	0.43%	−0.90%
f9	−1.02%	0.98%	1.26%	−1.02%	0.98%	−1.02%	0.98%	−1.02%	0.23%	−0.49%
f10	2.10%	−0.57%	−2.30%	1.43%	**5.43%**	**−4.57%**	−0.57%	−0.57%	1.17%	−2.46%
f11	0.74%	4.08%	−1.38%	8.08%	**0.08%**	**−5.92%**	2.08%	2.08%	−0.44%	0.92%
f12	−2.01%	1.77%	−4.23%	1.77%	−2.23%	−0.23%	−0.23%	1.77%	−0.99%	2.09%
f13	0.02%	−0.20%	−2.20%	−2.20%	−2.20%	−0.20%	1.80%	1.80%	0.29%	−0.62%
f14	−1.96%	1.37%	−4.08%	−0.63%	1.37%	1.37%	1.37%	5.37%	−0.65%	1.37%
f15	**−4.23%**	**1.77%**	2.59%	3.77%	1.77%	−2.23%	−2.23%	−2.23%	−0.99%	2.09%
f16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f17	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	0.36%
f18	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
f19	−2.88%	−0.88%	−0.60%	7.12%	−2.88%	−0.88%	−0.88%	−0.88%	−0.63%	1.33%
BLUiR	**26.09%**	**−9.24%**	2.30%	10.76%	4.76%	0.76%	[illegible]	2.76%	0.19%	−0.40%
LR	11.25%	−4.53%	5.74%	7.47%	3.47%	1.47%	[illegible]	5.47%	0.10%	−0.22%

the LR BL approach does not extend these results since the extracted metrics have no consistent or significant difference. The more positive impacted features were ϕ_4, ϕ_7, ϕ_8, and BLUiR. In addition, the observed change in the score for BLUiR approximate or even overcomes the scores with LR (initially higher), as occurs with metric NDCG@1, where the new score is 15.38 percentual points beyond the baseline.

Table 30 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-1 scores.

a) Top-5 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	2.83%	−1.69%	8.12%	0.31%	4.81%	4.31%	−2.30%	4.31%	1.44%	[illegible]
f2	−1.36%	−1.36%	−1.36%	0.64%	−1.36%	2.64%	−1.36%	0.64%	−0.49%	1.76%
f3	2.80%	−2.59%	3.54%	1.41%	−2.59%	−0.59%	[illegible]	1.41%	0.46%	−1.65%
f4	7.59%	3.07%	12.88%	−2.93%	−0.43%	[illegible]	2.46%	5.07%	1.86%	[illegible]
f5	−3.06%	−1.06%	−3.06%	−3.06%	1.94%	−3.06%	8.48%	−1.06%	−0.45%	1.63%
f6	8.24%	0.86%	5.76%	−3.14%	−2.14%	−3.14%	−3.30%	−3.14%	1.12%	−4.02%
f7	5.70%	3.76%	13.56%	−2.24%	0.26%	[illegible]	−0.71%	5.76%	1.67%	[illegible]
f8	5.70%	−0.24%	10.34%	−0.24%	0.26%	−8.24%	3.14%	5.76%	1.23%	−4.40%
f9	0.86%	0.30%	1.53%	0.30%	−1.70%	2.30%	2.15%	−1.70%	0.47%	−1.70%
f10	1.54%	0.98%	−1.02%	−1.02%	−1.02%	−1.02%	2.83%	−1.02%	0.28%	−1.02%
f11	8.97%	−4.67%	9.14%	−2.67%	[illegible]	3.33%	[illegible]	5.33%	−0.14%	0.52%
f12	−0.65%	[illegible]	7.12%	0.22%	−0.78%	0.22%	−1.94%	0.22%	0.30%	−1.09%
f13	1.73%	−1.40%	6.28%	2.60%	4.10%	−1.40%	−3.40%	−3.40%	0.51%	−1.84%
f14	3.64%	−3.18%	0.49%	−1.18%	5.82%	−1.18%	2.35%	−1.18%	−0.49%	1.75%
f15	1.73%	−1.40%	6.28%	−3.40%	6.60%	−3.40%	0.44%	2.60%	0.51%	−1.84%
f16	1.54%	−1.02%	2.21%	−1.02%	1.48%	0.98%	−1.02%	0.98%	−0.15%	0.54%
f17	2.41%	−0.72%	0.50%	1.28%	−2.72%	−2.72%	−2.72%	1.28%	−0.11%	0.40%
f18	1.20%	−1.36%	1.87%	0.64%	1.14%	−1.36%	−1.36%	0.64%	0.38%	−1.36%
f19	−0.13%	−1.82%	1.85%	−1.82%	12.18%	−1.82%	−3.98%	0.18%	0.44%	−1.57%
BLUiR	5.91%	−3.73%	18.98%	−3.73%	0.27%	[illegible]	−4.34%	8.27%	1.14%	−4.16%
LR	0.47%	−4.50%	2.79%	3.50%	6.50%	[illegible]	−8.50%	−8.50%	1.06%	−3.82%

b) Top-5 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	1.97%	[illegible]	[illegible]	7.53%	11.53%	3.53%	[illegible]	3.53%	−0.58%	1.21%
f2	1.40%	−1.05%	−0.77%	−3.05%	0.95%	−3.05%	2.95%	2.95%	0.45%	−0.94%
f3	2.30%	−0.37%	[illegible]	−0.37%	−0.37%	−4.37%	7.63%	5.63%	0.57%	−1.21%
f4	20.38%	−9.39%	3.51%	14.61%	−3.39%	4.61%	−1.39%	6.61%	−2.23%	4.71%
f5	3.79%	[illegible]	−0.60%	−0.88%	1.12%	−2.88%	−2.88%	5.12%	0.12%	−0.24%
f6	[illegible]	[illegible]	8.37%	2.19%	2.19%	[illegible]	14.19%	−3.81%	0.41%	−0.87%
f7	24.11%	−5.67%	2.69%	14.33%	−1.67%	2.33%	4.33%	0.33%	−1.25%	2.64%
f8	[illegible]	[illegible]	−1.45%	12.37%	2.37%	−1.63%	12.37%	2.37%	0.07%	−0.15%
f9	−0.63%	−1.08%	−2.80%	−1.08%	2.92%	−3.08%	0.92%	−5.08%	0.41%	−0.87%
f10	[illegible]	[illegible]	−3.70%	6.94%	6.94%	0.94%	−1.06%	2.94%	−1.09%	2.31%
f11	1.75%	[illegible]	−4.56%	11.53%	7.53%	−6.47%	[illegible]	1.53%	−0.27%	0.58%
f12	1.49%	0.16%	[illegible]	12.16%	−1.84%	[illegible]	−1.84%	4.16%	−1.12%	2.37%
f13	−1.65%	[illegible]	[illegible]	−2.09%	−2.09%	1.91%	−0.09%	3.91%	0.64%	−1.35%
f14	−0.50%	1.73%	−4.64%	1.73%	3.73%	−2.27%	−2.27%	5.73%	−0.07%	0.15%
f15	[illegible]	0.86%	2.23%	6.86%	4.86%	−5.14%	4.86%	−1.14%	−0.16%	0.34%
f16	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	−0.17%	0.36%
f17	[illegible]	−0.94%	4.43%	5.06%	3.06%	−4.94%	1.06%	−2.94%	−0.95%	2.01%
f18	−0.51%	−0.51%	−0.51%	1.49%	−0.51%	−0.51%	−0.51%	−0.51%	−0.51%	1.07%
f19	−8.12%	3.88%	−1.30%	7.88%	1.88%	−6.12%	1.88%	−0.12%	−0.14%	0.30%
BLUiR	14.84%	−9.38%	−1.92%	12.62%	0.62%	8.62%	12.62%	0.62%	0.25%	−0.54%
LR	2.75%	[illegible]	2.10%	9.64%	9.64%	1.64%	3.64%	3.64%	1.22%	−2.57%

In Tomcat, we have more samples with statistically significant scores' differences. Still, the impact correlated to *Wraps with If* is almost opposite to what we have found in AspectJ. Overall, samples with the pattern show scores near the baseline (or slightly below), and samples without the pattern have higher scores than the baseline. Most impacted are for features ϕ_1 (-6.52% in Top-1 when present, $+10.30\%$ in NDCG@1

Table 31 – Impact of the presence and absence of the repair patterns in AspectJ and Tomcat based on the Top-10 scores.

a) Top-10 score differences from baseline in AspectJ

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	**1,49%**	**−4,15%**	4,88%	−2,15%	5,85%	−0,15%	−1,07%	1,85%	1,07%	−3,84%
f2	−2,38%	1,62%	0,84%	−0,38%	−2,38%	3,62%	−2,38%	1,62%	−0,64%	2,31%
f3	3,51%	−5,01%	−0,88%	0,99%	[illegible]	−1,01%	9,92%	0,99%	−0,49%	1,74%
f4	**7,59%**	**3,07%**	[illegible]	[illegible]	−0,43%	[illegible]	2,46%	5,07%	1,86%	[illegible]
f5	1,05%	−2,08%	−0,86%	[illegible]	**0,92%**	**−4,08%**	**7,46%**	**3,92%**	−0,17%	0,61%
f6	9,13%	−1,95%	5,41%	−1,95%	−1,45%	2,05%	**1,44%**	**−1,95%**	1,27%	−4,67%
f7	**5,70%**	**3,76%**	[illegible]	−2,24%	0,26%	−6,24%	−0,71%	5,76%	1,67%	−5,99%
f8	5,36%	−0,59%	**13,22%**	**−0,59%**	−0,09%	[illegible]	2,80%	7,41%	1,33%	−4,77%
f9	2,07%	−1,06%	0,16%	0,94%	**1,94%**	**0,94%**	0,78%	−1,06%	0,42%	−1,50%
f10	1,54%	0,98%	−1,02%	−1,02%	−1,02%	−1,02%	2,83%	−1,02%	0,28%	−1,02%
f11	5,44%	[illegible]	**7,59%**	**4,11%**	**2,11%**	**−1,89%**	[illegible]	6,11%	−0,07%	0,23%
f12	−1,49%	[illegible]	**3,72%**	**−1,18%**	−4,18%	0,82%	−5,94%	6,82%	−0,49%	1,75%
f13	1,05%	−0,08%	8,82%	3,92%	3,42%	−2,08%	−4,08%	−4,08%	0,27%	−0,96%
f14	3,14%	−6,24%	−2,67%	7,76%	5,26%	1,76%	10,83%	−0,24%	−0,51%	1,82%
f15	9,11%	1,16%	10,51%	−0,84%	**11,16%**	**−2,84%**	2,69%	3,16%	1,59%	−5,72%
f16	0,86%	−1,70%	1,53%	0,30%	0,80%	0,30%	−1,70%	0,30%	0,04%	−0,14%
f17	2,77%	−1,48%	2,19%	4,52%	**5,02%**	**−1,48%**	4,06%	−3,48%	−0,09%	0,33%
f18	−0,84%	−1,40%	−0,18%	0,60%	1,60%	2,60%	0,44%	−1,40%	−0,36%	1,29%
f19	**7,59%**	**−0,93%**	−0,02%	−0,93%	**14,57%**	**−0,93%**	−1,39%	−2,93%	0,55%	−1,99%
BLUiR	9,18%	−4,15%	[illegible]	1,85%	−1,65%	[illegible]	−1,07%	7,85%	−0,24%	0,85%
LR	[illegible]	2,23%	−0,80%	−1,77%	**10,73%**	**−7,77%**	1,31%	−3,77%	−0,90%	3,23%

b) Top-10 score differences from baseline in Tomcat

	MNC_N	-MNC_N	W_If	-W_If	WMR	-WMR	WVR	-WVR	ALL_M	-ALL_M
f1	**5.18%**	**−16.82%**	11.09%	3.18%	11.18%	3.18%	[illegible]	−0.82%	0.79%	−1.66%
f2	1.59%	−1.08%	−0.53%	−5.08%	0.92%	−3.08%	2.92%	2.92%	0.91%	−1.92%
f3	3.55%	−3.78%	−0.23%	0.22%	0.22%	−5.78%	6.22%	10.22%	1.14%	−2.41%
f4	**18.52%**	**−7.26%**	3.93%	14.74%	−3.26%	2.74%	2.74%	6.74%	−1.85%	3.90%
f5	5.00%	8.11%	−1.62%	0.11%	**4.11%**	**−3.89%**	**−3.89%**	**4.11%**	0.35%	−0.73%
f6	−1.39%	−4.72%	3.46%	5.28%	−2.72%	[illegible]	**13.28%**	**−2.72%**	0.74%	−1.56%
f7	**23.43%**	**−6.35%**	2.02%	13.85%	−2.35%	3.65%	3.65%	−0.35%	−1.43%	3.02%
f8	4.35%	−8.53%	**−4.08%**	**15.47%**	7.47%	−2.53%	9.47%	−2.53%	−0.84%	1.78%
f9	−3.00%	0.55%	−2.90%	−3.45%	**2.55%**	**−5.45%**	2.55%	[illegible]	1.03%	−2.18%
f10	−8.13%	−5.69%	−5.51%	8.31%	12.31%	2.31%	0.31%	4.31%	−1.24%	2.63%
f11	−2.46%	7.10%	**−6.45%**	**17.10%**	**9.10%**	**−8.90%**	−2.90%	3.10%	−0.72%	1.52%
f12	−1.20%	−0.75%	**−9.93%**	**9.25%**	−0.75%	[illegible]	−0.75%	1.25%	−0.54%	1.14%
f13	−3.51%	−1.95%	[illegible]	−3.95%	−3.95%	2.05%	0.05%	2.05%	0.03%	−0.06%
f14	−2.82%	−3.04%	[illegible]	6.96%	−1.04%	2.96%	2.96%	4.96%	−1.10%	2.33%
f15	−8.10%	−0.32%	3.31%	5.68%	5.68%	−4.32%	3.68%	−2.32%	−0.10%	0.20%
f16	−0.85%	−0.85%	−0.85%	1.15%	−0.85%	1.15%	−0.85%	−0.85%	−0.60%	1.26%
f17	−6.91%	−3.14%	4.50%	4.86%	**4.86%**	**−5.14%**	2.86%	[illegible]	−0.91%	1.92%
f18	−1.18%	0.82%	−1.18%	0.82%	−1.18%	0.82%	−1.18%	−1.18%	−0.94%	1.97%
f19	**−6.22%**	**7.34%**	0.70%	5.34%	1.34%	−4.66%	1.34%	1.34%	−0.68%	1.45%
BLUiR	9.26%	−0.96%	−2.96%	13.04%	−2.96%	3.04%	11.04%	3.04%	0.16%	−0.33%
LR	2.34%	−1.66%	−0.39%	6.34%	12.34%	4.34%	4.34%	6.34%	1.39%	−2.93%

when absent), ϕ_8 (-4.08% in Top-10 when present, $+15.47\%$ in Top-10 when absent), ϕ_{11} (-6.45% in Top-10 when present, $+17.10\%$ in MAP when absent), ϕ_{12} (-5.60% in MAP when present, $+6.67\%$ in NDCG@5 when absent) and for BLUiR (-1.92% in Top-5 when present, $+14.74\%$ in MAP when absent. ϕ_7 is more impacted (with some inconsistency): both scores are above the baseline with the highest difference of $+15.63\%$

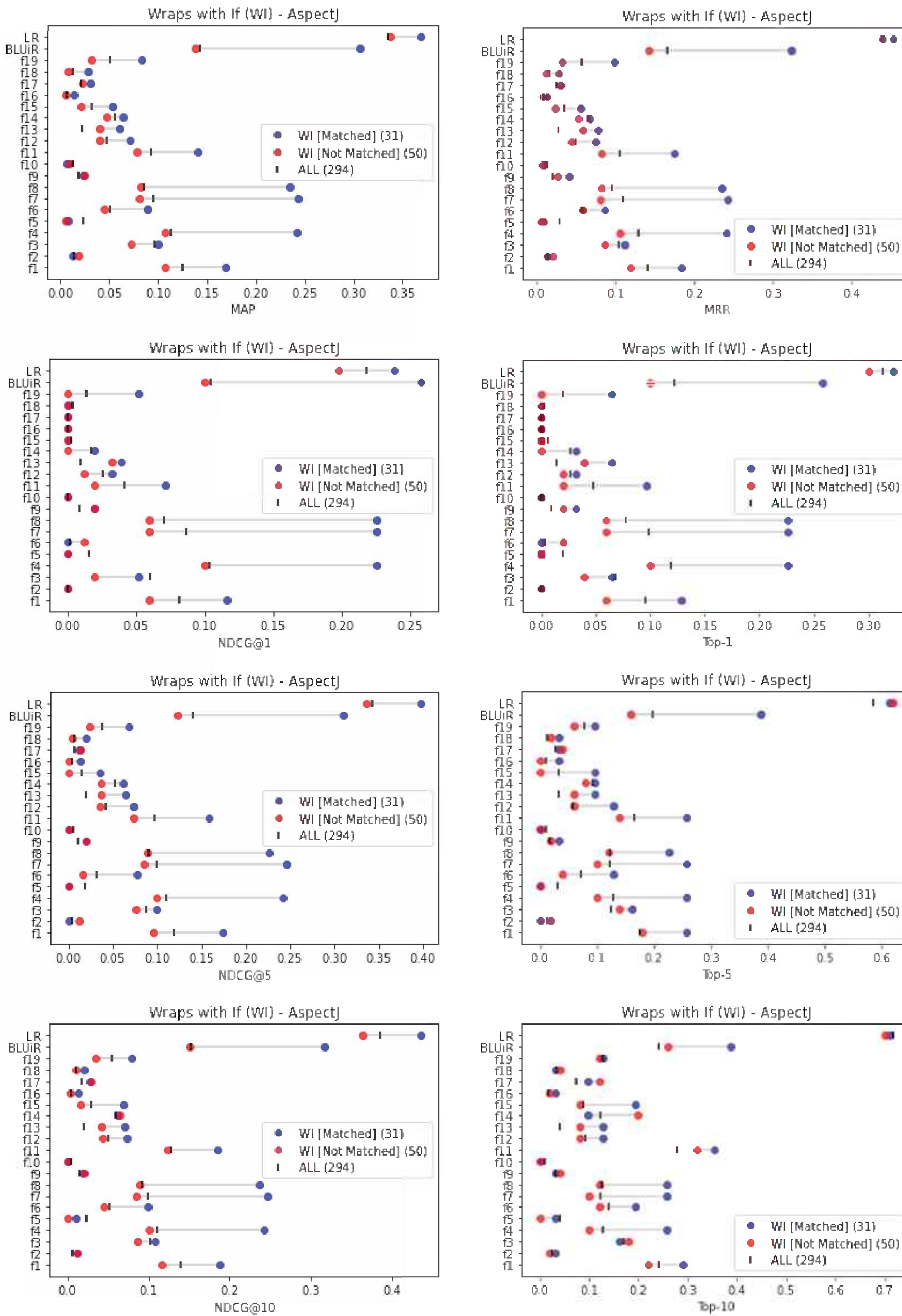


Figure 59 – Score rankings differences for *Wraps with If* repair pattern in AspectJ (FC+NFC).

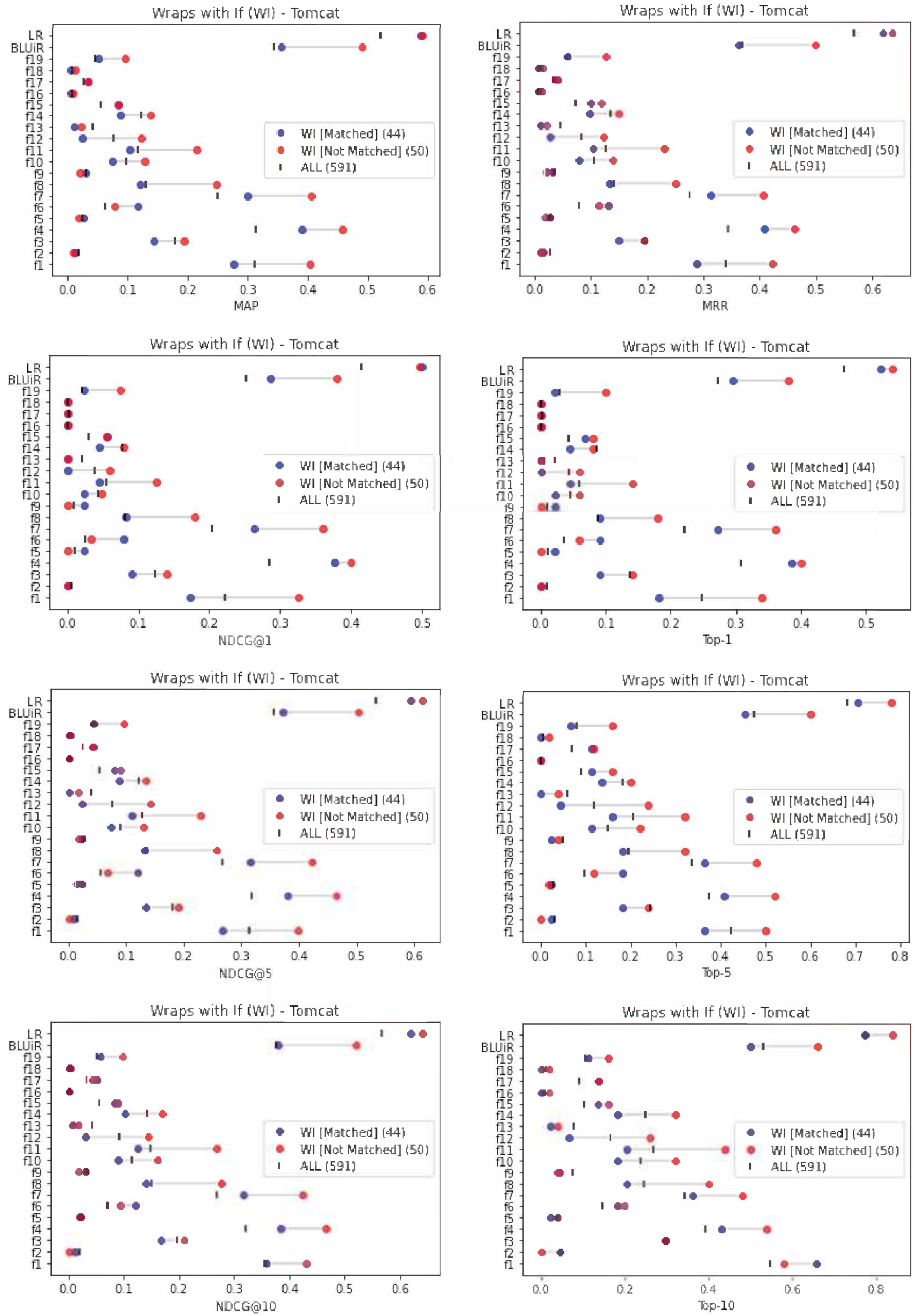


Figure 60 – Score rankings differences for *Wraps with If* repair pattern in Tomcat (FC+NFC).

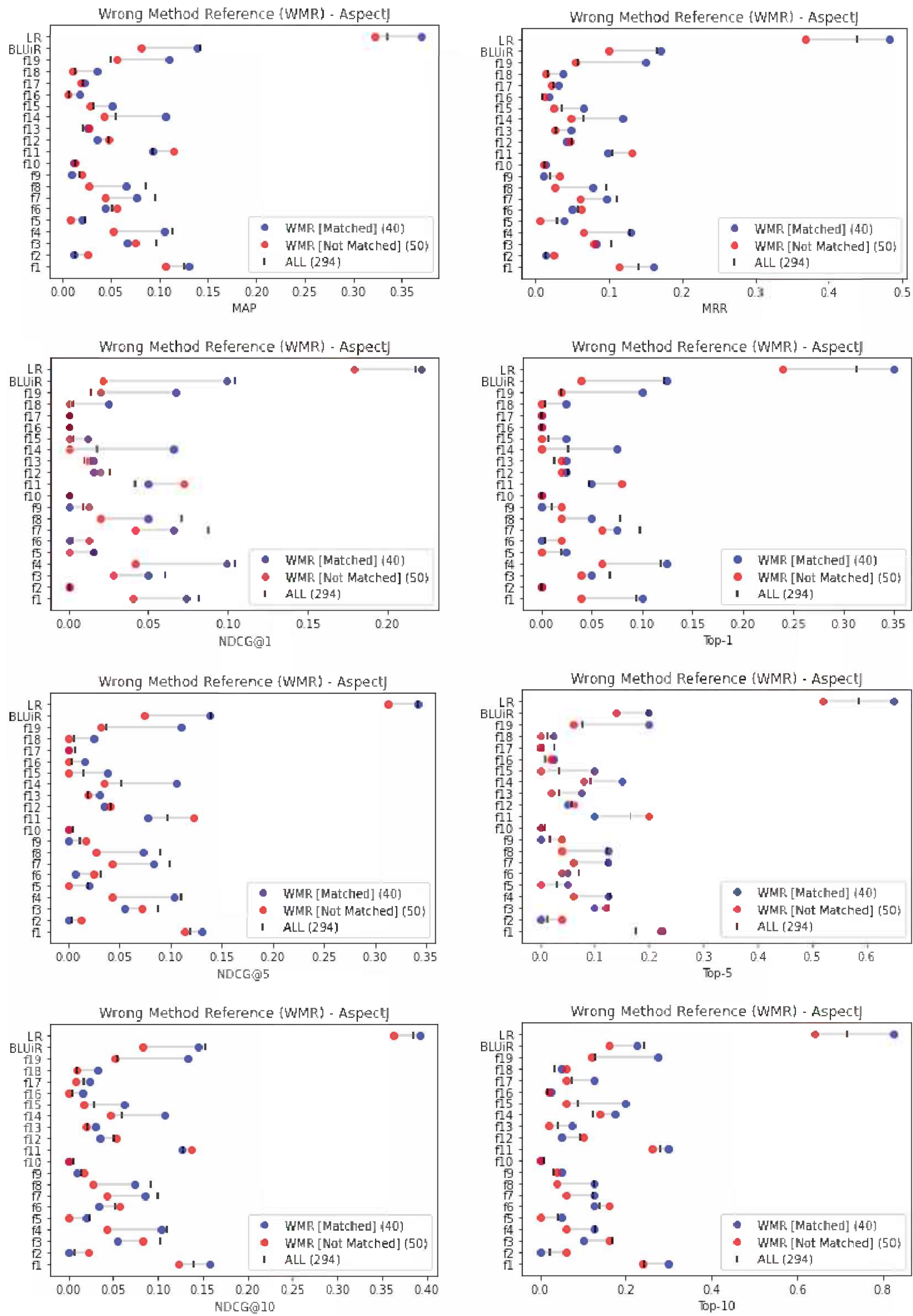


Figure 61 – Score rankings differences for *Wrong Method Reference* repair pattern in AspectJ (FC+NFC).

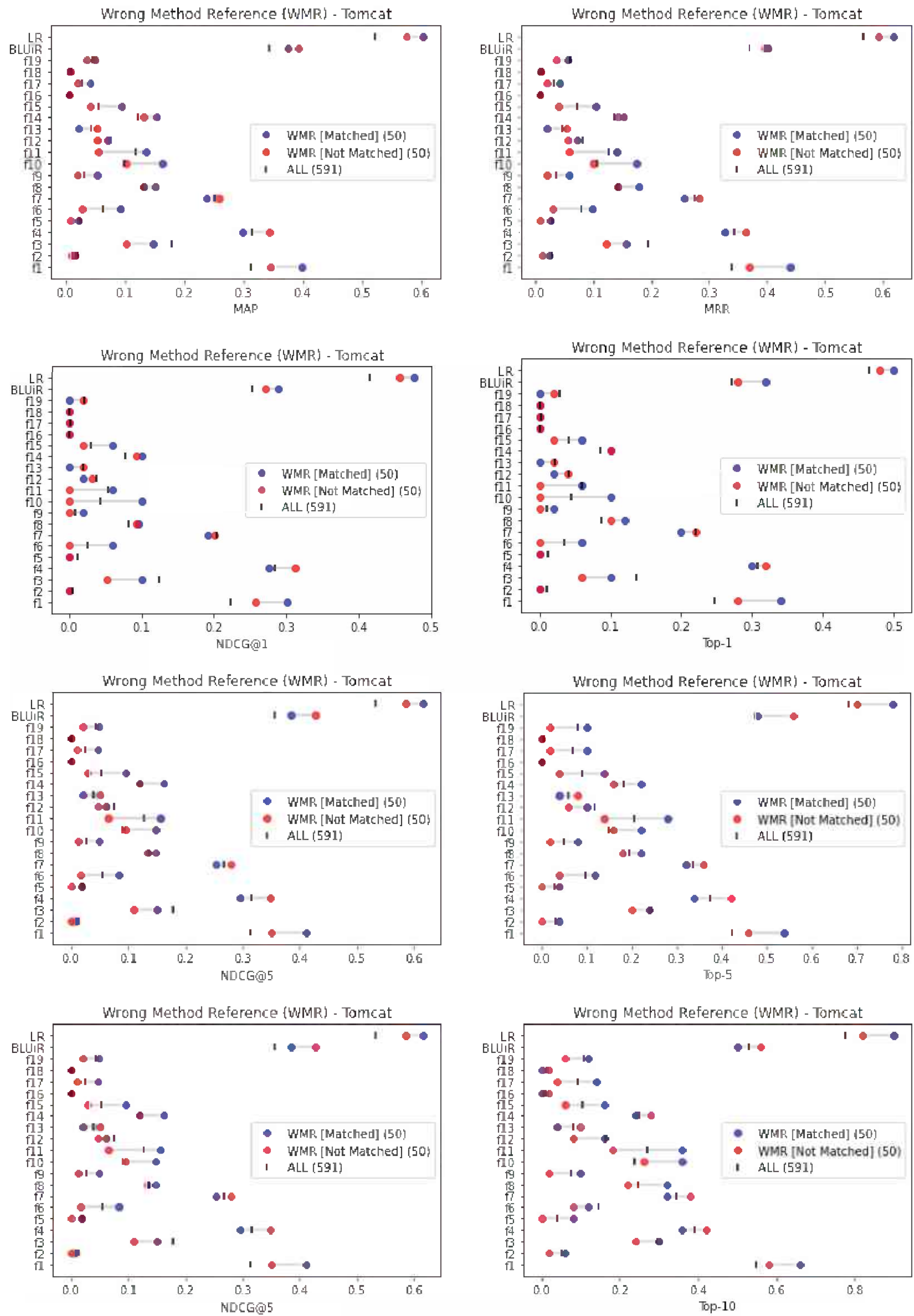


Figure 62 – Score rankings differences for *Wrong Method Reference* repair pattern in Tomcat (FC+NFC).

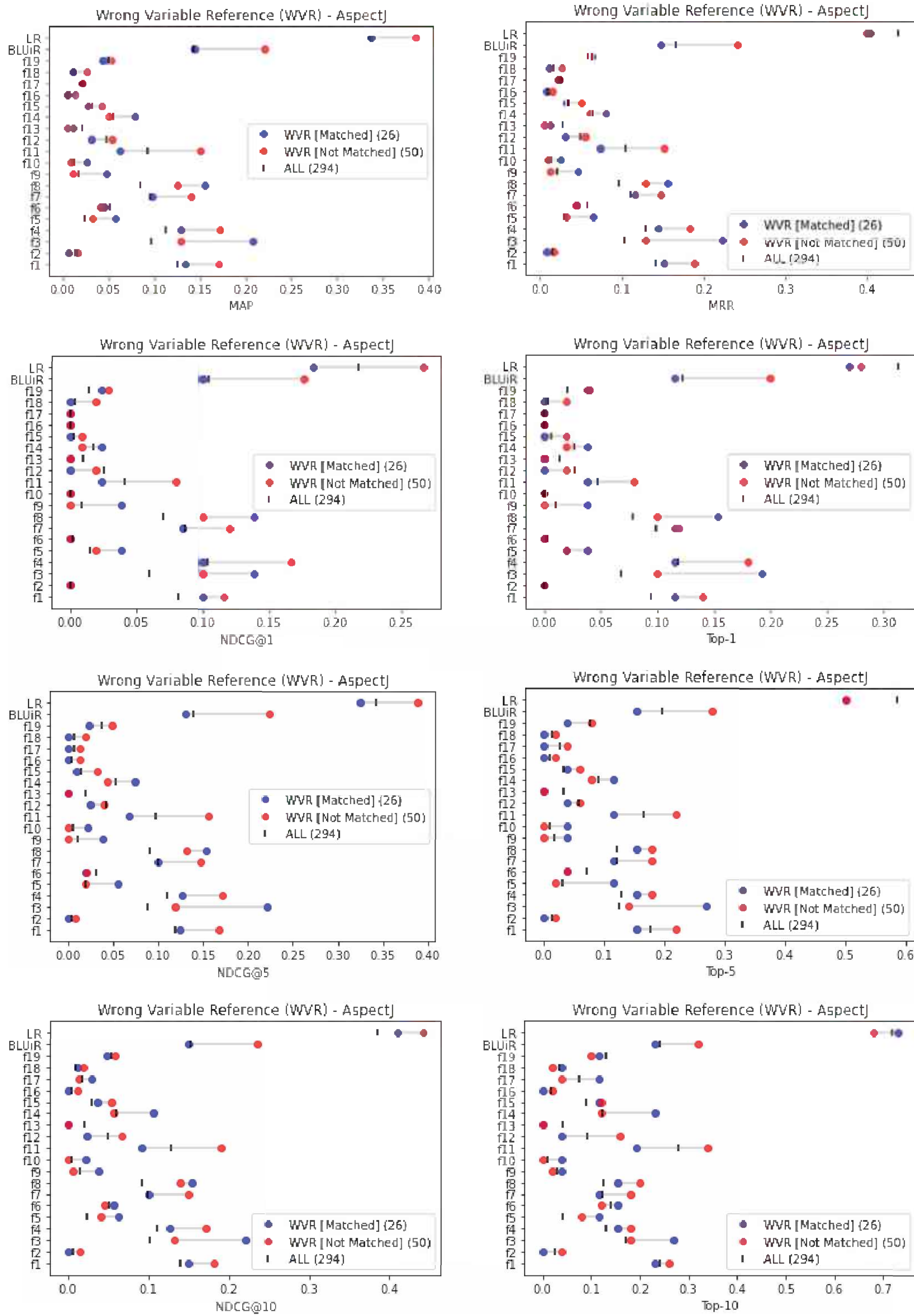


Figure 63 – Score rankings differences for *Wrong Var Reference* repair pattern in AspectJ (FC+NFC).

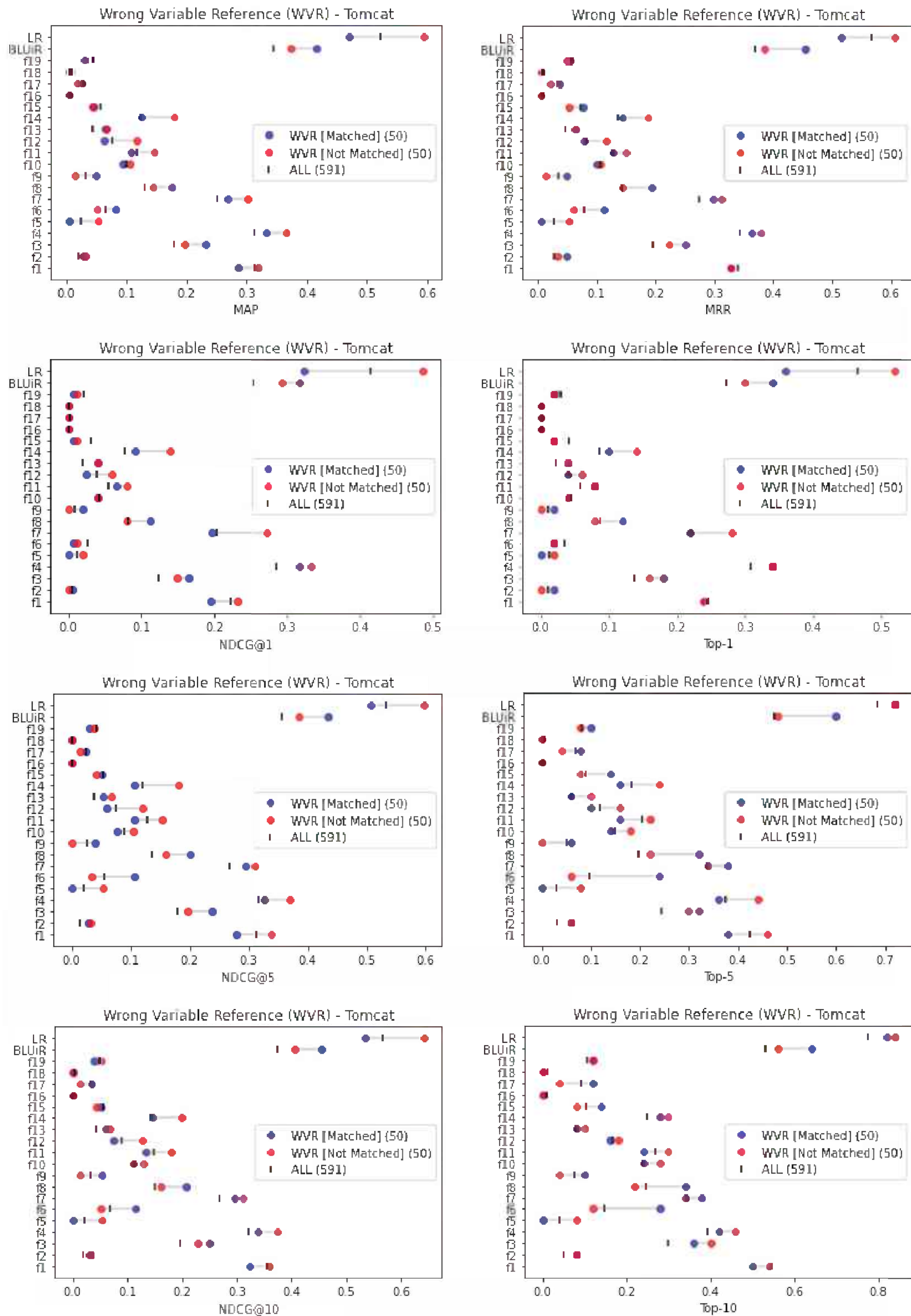


Figure 64 – Score rankings differences for *Wrong Var Reference* repair pattern in Tomcat (FC+NFC).

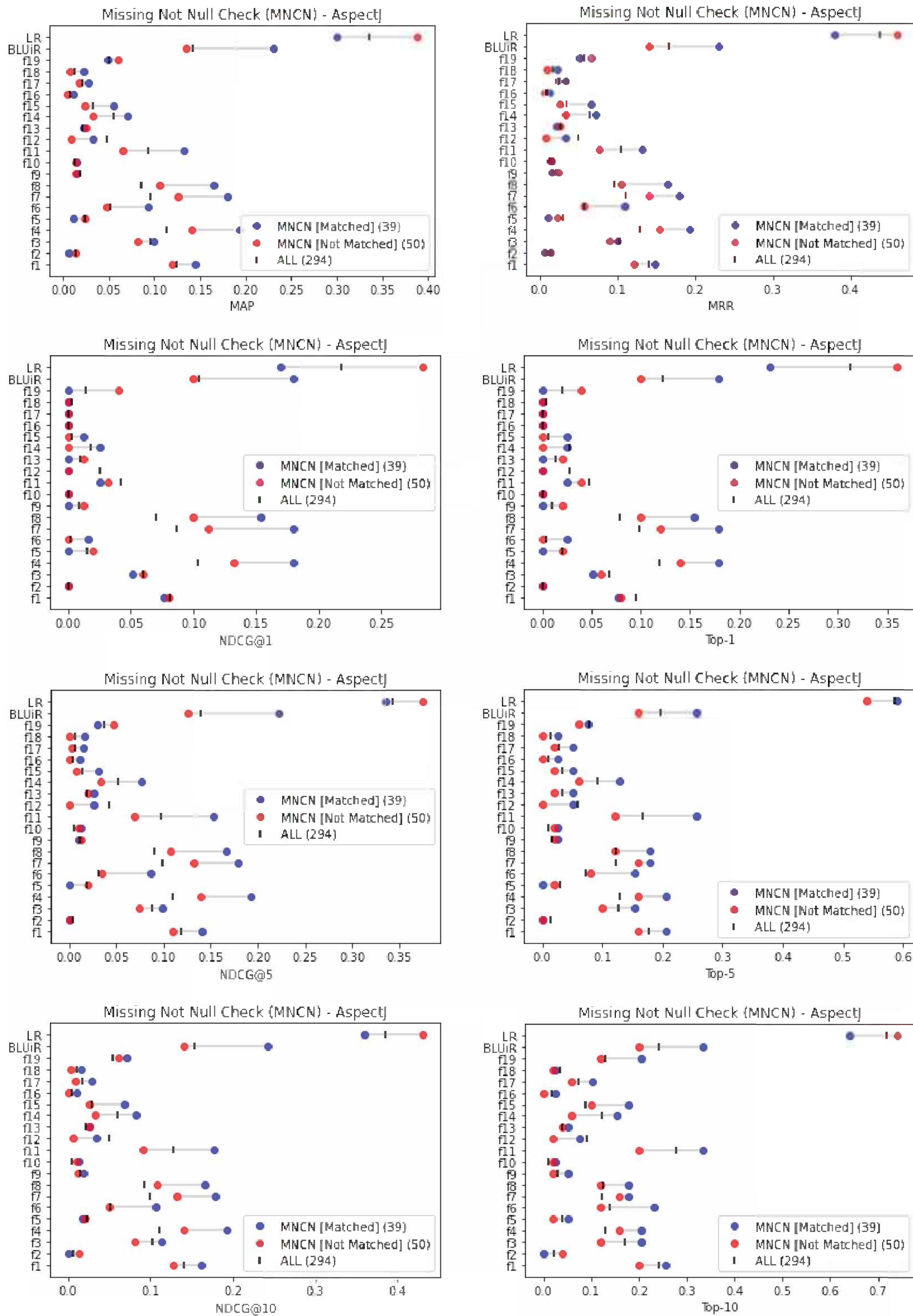


Figure 65 – Score rankings differences for *Missing Not-Null Check Reference* repair pattern in AspectJ (FC+NFC).

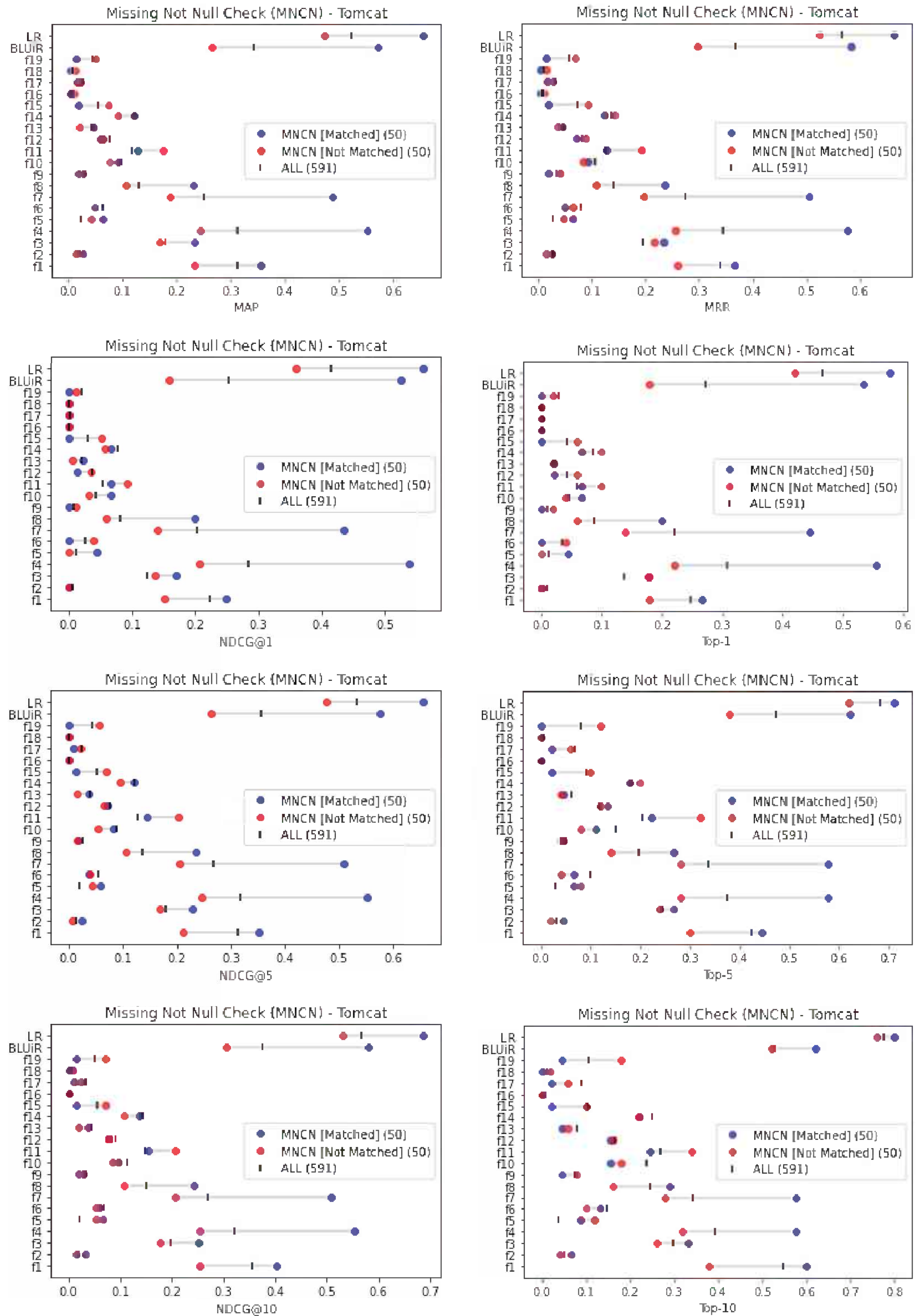


Figure 66 – Score rankings differences for *Missing Not-Null Check Reference* repair pattern in Tomcat (FC+NFC).

in NDCG@1 when the pattern is absent, and the lowest difference of +2.69% in Top-5 when the pattern is present. Other affected scores but in a lower degree are ϕ_3, ϕ_4, ϕ_6, ϕ_{10}, ϕ_{14}, ϕ_{19}.

As with AspectJ, the samples in Tomcat show score differences with statistical significance. The pattern presence/absence is correlated to score changes for features and the BLUiR strategy. Nevertheless, this time, bugs whose patches do not contain the pattern would be associated with the easiest bug localization, and bugs requiring patches with the pattern would be near the baseline score.

7.4.2 Wrong Method Reference Repair Pattern

In AspectJ and for *Wrong Method Reference*, we have scores differences statistically significant, but the range of variation is smaller than those observed in the *Wraps with If*. Considering statistically significant differences, we highlight scores for features ϕ_{14} (+4.78% in NDCG@1 when the repair pattern is present, −1.75% in NDCG@1 when absent), ϕ_{19} (+7.89% in NDCG@10 when present and −0.58% in NDCG@5 when absent) and also for LR (+10.73% in Top-10 when present, −7.77% in Top-10 when absent). Other features as ϕ_5 and ϕ_{15} have scored with statistically significant differences but with more minor variations. Overall, the scores are higher than the baseline score when the pattern is present. When the pattern is absent, the scores are near or lower than the baseline score. Again, as with *Wraps with If*, the results for AspectJ suggest that patches requiring this pattern would be easier to localize than when the pattern is absent compared to the baseline.

In Tomcat, we also have score differences for *Wrong Method Reference*. We can highlight the differences for ϕ_6 (+3.46% in NDCG@1 when present, −2.57% in NDCG@1 when absent), ϕ_{10} (+5.76% in NDCG@1 when present, −4.57% in Top-1 when absent), ϕ_{11} (+7.53% in Top-5 when present, −6.82% in NDCG@10 when absent), and ϕ_{15} (+4.86% in Top-5 when present, −5.14% in Top-5 when absent). This time results from Tomcat suggest that the presence of the pattern in the patches would indicate an easier bug localization than for the bugs where patches do not contain the pattern.

7.4.3 Wrong Variable Reference Repair Pattern

In AspectJ, we can perceive from Figure 63 that many scores have apparent differences when compared to the baseline and also between scores of matched versus not-matched samples. Some examples with the more significant differences are ϕ_3, ϕ_{11},

and BLUiR. Other features as ϕ_4, ϕ_5, ϕ_7, ϕ_9, ϕ_{12} also present some differences. Despite that, few of these features have scores rejecting H0 for some metrics. Examples are ϕ_5 (+8.48% in Top-5 when present, −1.06% in Top-5 when absent), ϕ_7 (+0.19% in MAP when present, +4.51% in MAP when absent), and ϕ_8 (+6.99% in MAP when present, +3.94% in MAP when absent). The inconsistency between the score differences and the failure to reject the null hypothesis in statistical tests does not give enough confidence for insights about this pattern in AspectJ. Another factor contributing to the inconsistent results may be the number of instances for the matched sample, 26, a small number and almost half of the cases on the not matched side.

In Tomcat, we can see some score differences for *Wrong Variable Reference*, but most are small ones, and few features show consistent and statistical significance: ϕ_5 (−3.89% in Top-10 when present, +5.12% in Top-5 when absent), ϕ_6 (+14.19% in Top-5 when present, −3.81% in Top-5 when absent), and ϕ_9 (+1.4% in NDCG@5 when present, −5.08% in Top-5 when absent). LR score also show clear visual difference in Figure 64, but this results is not confirmed with hypothesis tests. ϕ_5 has a small contribution since the score range is below .10 values for almost all the metrics and is a feature with statistically different scores. The sample where the pattern is present has a lower score than the baseline. When the pattern is absent, the score is higher. We would infer from this that BL strategies based on this feature would produce betters scores on samples where the bug patches do not require fixings like the *Wrong Variable Reference* repair pattern. Nonetheless, it is still possible to have higher scores for features like ϕ_6 and ϕ_9 when this repair pattern is present.

7.4.4 Missing Not-Null Check Repair Pattern

In AspectJ, many scores show sharp differences, according to Figure 65. For example, ϕ_4, ϕ_7, ϕ_8, BLUiR, and LR. But curiously, Table 23-a shows scores rejection of H0 hypothesis for some metrics only in ϕ_{11} (+8.97% in Top-5 when present, −4.87% in NDCG@5 when absent) and ϕ_{15} (+2.29% in MAP when present, −0.86% in MAP when absent). Overall, according to Tables 24-a to Tables 31-a, most of the scores are higher than baseline for bugs whose patches contain the pattern and lower than baseline scores when the pattern is absent.

In Tomcat and compared to the other analyzed samples and patterns, *Missing Not-Null Check* is possibly the repair pattern with more shreds of evidence of score differences spanning for more features and impacting both BL strategies. We can confirm the evidence by analyzing Figure 66, Tables 24-b to 31-b, and also Table 23-b. The

more significant differences with statistical significance occur in features ϕ_1 (+5.18% in Top-10 when present, −16.82% in Top-10 when absent), ϕ_4 (+25.36% in NDCG@1 when present, −9.39% in Top-5 when absent), ϕ_7 (+24.12% in NDCG@5 when present, −8.00% in Top-1 when absent), ϕ_8 (+11.85% in NDCG@1 when present, −4.18% in NDCG@10 when absent), and in both BL strategies, BLUiR (+27.22% in NDCG@1 when present, −9.38% in Top-5 when absent) and LR (+14.57% in NDCG@1 when present, −5.60% in NDCG@1 when absent). Other features as ϕ_{15} and ϕ_{19} also present statistically significant differences but with a smaller range. Overall, scores with substantial differences are higher than baseline when the *Missing Not-Null Check* is present in patches. In comparison, patches without the pattern show lower scores compared to baseline. The situation is not so different with Tomcat. Nevertheless, the impact observed on the scores is variate: positive for some features as ϕ_6 and ϕ_9, and negative for ϕ_5.

7.5 Answers for the Research Questions

Here we synthesize the results and analysis previously presented to answer the initially proposed research questions.

7.5.1 RQ6: on the existence of differences correlated to the presence versus absence of repair patterns in patches

When we compare a sample of bugs where the respective patches match a given repair pattern against another sample of bugs where this pattern is not present, is there any difference in the measured metrics targeting the ranking of bug suspects? Are these differences statistically significant?

Based on the results in Section 7.3 and the analysis of Section 7.4 we found many cases where the score of features and the BL strategies differ. We compare scores extracted from a sample with bug reports whose patches contain one of the repair patterns to scores from a sample where this pattern is not present on the respective patches. We found different situations for the differences: 1) sometimes the scores for the sample matched with the pattern is higher than the baseline (suggesting an easier BL for these cases), and the scores for samples without the pattern is near to the baseline, usually bellow; 2) sometimes the first situation is inverted; 3) sometimes we have the baseline scores in the middle, while matched versus not-matched samples scores are far in one of the extremes. We have found differences with statistical significance (H0 rejected), but

not all the differences show the significance for most metrics. The results for *Missing Not-Null Check* repair pattern in Tomcat have the most substantial shreds of evidence for the differences, especially for the features ϕ_4, ϕ_7 and the BL strategy, BLUiR.

7.5.2 RQ7: on the type of impact correlated to the presence versus absence of repair patterns in patches

What type of impact is associated with the evaluated metric's score rankings by the presence of a repair pattern in the patches of a bug sample? Moreover, when the repair pattern is absent?

Again, and somehow aligned to RQ6, we have found different types of impact. We find a positive impact on the score rankings when it is higher than the baseline score on the presence of the repair pattern and is lower or near to baseline, on the absence of the repair pattern. This occurs for *Wrong Method Reference* and *Missing Not-Null Check* in AspectJ and Tomcat. For *Wraps with If* the positive impact is viewed for AspectJ. On the other hand, the same pattern in Tomcat shows the opposite effect: the absence of the repair pattern presents a higher score than the baseline. For this case, while there are significant differences, we cannot associate the same tendency (or type of impact) to the repair pattern presence/absence on different projects. The lack of H0 rejection in statistical tests for *Wrong Variable Reference* samples makes it harder to state with sure the type of impact observed in AspectJ. However, Tomcat's situation is not different since few features present statistical significance in score differences, which are small overall.

7.5.3 RQ8: on the degree of the impact correlated to the presence versus absence of repair patterns in patches

What is the degree of the impact correlated to the repair pattern's presence or absence on the metrics measured?

The degree of impact observed for features and BL strategies variate a lot. For example, for *Missing Not-Null Check* and considering results with statistical significance, the score increase can reach 27.22 percentual points above the baseline (e.g., in BLUiR with NDCG@1 when the pattern is present in Tomcat). At the same time, the score decrease can reach 16.82% percentual points below the baseline (e.g., in ϕ_1 with Top-10 when the repair pattern is absent in Tomcat). For *Wraps with If* the increase reaches 18.98 percentual points above the baseline (e.g., in BLUiR with Top-5 when the repair

pattern is present in AspectJ), while the decrease goes 9.93 percentual points below the baseline (e.g., in ϕ_{12} with Top-10 in Tomcat when the repair pattern is present in Tomcat. Interesting to note that there is no transfer of influence between projects. The last example illustrates this, observing that for AspectJ, we can observe a correlation between the repair pattern presence and the increase in the BL score and, consequently, the ease of localizing the bugs compared to the baseline scores. On the other hand, in Tomcat, the increase in the BL score is correlated to the repair pattern absence (especially for BLUiR, since for LR, the score is almost the same between metrics). *Wrong Method Reference* and *Wrong Variable Refence* seems to be the less impacted repair patterns. The smaller number of H0 rejections on the score differences reinforces this observation. However, even considering the reduced impact, we can point the differences. The score increase in *Wrong Method Reference* reaches 14.57 percentual points above the baseline (e.g., in ϕ_{19} with Top-10 when the repair patterns are present in AspectJ). In contrast, the decrease reaches 7.77 below the baseline (e.g., in LR with Top-10 when the repair pattern is absent in AspectJ). In *Wrong Variable Reference*, the score increase reaches 14.19 percentual points (e.g., for ϕ_6 with Top-5 when the repair pattern is present in Tomcat). In comparison, the decrease gets 6.08 percentual points (e.g., for ϕ_9 with Top-5 when the repair pattern is absent in Tomcat). The lack of H0 rejections makes the findings for these two last repair patterns not sound like the first two.

Finally, we can observe there is no impact on some features by a repair pattern's presence or absence, especially when the baseline score is already low. One of the most notable examples is the ϕ_2 that presents small variations from the baseline for almost all the repair patterns and metrics in both projects. ϕ_2 is a special case, already with a low score in the baseline. Consequently, the feature does not contribute too much to the localization in both BL strategies. Since the feature depends on the presence of Javadoc API in code, and it is not unusual to have code without this type of documentation, it is reasonable that the scores will not be so high, regardless. A similar situation also occurs for features ϕ_{15} to ϕ_{19}. The reasons are slightly different but proceed from the already lower baseline scores. These are query-independent features, and the produced scores do not depend on the bug report content but only from the graph/structural characteristics of the affected source code file (i.e., the number of in/out dependencies for other source codes, PageRank and HITS scores).

7.6 Threats to Validity

The co-occurrences impose a challenge to analyze and extract conclusions about the influence of individual repair patterns. It is hard to design and set up the ideal experimental conditions since isolating these patterns and getting significant and representative samples may be impossible for some cases. Even considering that we can select bugs whose patches match single repair patterns, this can be a false positive. The patch would still contain repair patterns not formally defined, not identified, or undetected by the ADD tool. We would also consider ADD tool's repair pattern detection precision and recall since false positives and false negatives would impact our results. While the manual inspection was applied to define the ADD tool precision and recall against Defects4J bugs, a large dataset as LR–dataset makes an inspection too time-consuming and feasible only to punctual cases. Finally, we should extend the study to analyze more bugs and more projects since we concentrate on only two projects from the six present in the LR–dataset. Therefore, the extension also applies to the analysis of more datasets.

7.7 Limitations and Future Work

A coverage measure exposing the patch parts related to each pattern would help regulate and filter patches more representative of each repair pattern. For example, a patch containing a repair pattern associated with 100% of the patch code lines is a more authentic representative of this pattern than a patch with only 10% of its extension associated with the same repair pattern. This extension would help to increase the accuracy of the results correlating repair patterns with changes in the scores.

The idea about an environment capable of experimenting and comparing different BL approaches was started here with the developed experimental package. This package contains some of the conceptual ideas and guidelines exposed in Chapter 4. However, much work still needs to be done for a more complete and practical framework that would potentially emerge with the development of the package. The most critical improvement is to increase the scalability of the experimental package. The current implementation does not allow optimum computational power and resources usage (memory and disk included). We based our current model on relational databases and an ORM technology that does not allow parallelization in a viable way. Furthermore, the volume of data on tables is enormous (some of them with a dozen GB). Therefore, applying some optimization strategies (e.g., better indexing, data compression, query optimization) was not enough to improve the processing time and memory usage. Therefore, the

refactoring towards a better architecture based on distributed processing and storage (e.g., Spark) would be our next step in this direction.

7.8 Related Work

Liu et al. (2018) point out that "the prerequisite for further advancing state-of-the-art APR techniques is to acquire all-round and detailed understanding about real-world patches". Therefore, the study tried to expose intrinsic characteristics on human-written patches that would help to tune APR strategies and help to synthesize better machine-generated patches. The study goes beyond the statement level and proposes an even finer-grained knowledge about the code elements involved in a patch, making the automatically generated patches more realistic and applicable. Here we also show that the type of patches required to fix a reported bug can influence the results obtained by a given evaluated BL strategy because a bug dataset contains a heterogeneous set of bugs, and many works do not consider or differentiate it while testing the BL approaches on these datasets. Furthermore, since the localization is a required step in APR, it also would help to explain some vies caused by a non-characterized dataset applied in the evaluation of the APR strategies.

The reproducibility study conducted by Lee et al. (2018) reviews past approaches for Information Retrieval-based Bug Localization (IRBL) under new conditions and settings, proposing Bench4BL as a new benchmark for strategies evaluation. In our experimental preparation, we consider some issues tested by Lee et al. (2018) study as influential factors for BL, e.g., testing of BL approaches with larger datasets, maintenance of consistency between project version and the bug report, and exclusion of testing files. Lee et al. (2018) also shows there is room to improve the performance of BL strategies, even considering past approaches, also confirmed with our results.

DeMarco et al. (2014) proposes Nopol, an APR tool focused on automatically repair buggy IF conditions and missing preconditions. In our contexts, we can relate it with the repair patterns *Missing Null Check* and *Conditional Expression* variations. Nopol is a direct example of how the characteristics of the patches can impact the design and evaluation of debugging tools since it is a real representation of a tool focused on specific classes of bugs. Still, Nopol reinforces the importance of characterizing a bug dataset to accurately define the performance results derived from an evaluation because there is no sense to test a tool like Nopol with bugs of different classes it can handle.

Liu et al. (2019) investigated the influence of bug localization "tweaking" on APR

approaches, confirming the presence of bias while conducting evaluations and comparisons in strategies targeting the automation of debugging tasks. Furthermore, the work shows that research should attempt to clarify and qualify the applied benchmarks (and settings) to avoid misleading conclusions about the tested approaches, typically relying only upon direct comparisons of obtained performance results. Similarly, we propose the evaluation of BL approaches accounting for the dataset/benchmark characterization and differentiating experimental samples by their bugs' nature (e.g., bug patches characteristics in our case) so the analysis, comparisons and conclusions about the tested approaches would be more informed and accurate.

Böhme et al. (2017) propose DBGBench, as the "the first human-generated benchmark for the qualitative evaluation of automated fault localization, bug diagnosis, and repair techniques", considering the gap between the research proposals to support debugging and the industrial practice needs. In the study, the conduction of debugging tasks by professional developers contrasts with the research proposals to support these tasks. The DBGBench is one of the contributions derived by Böhme et al. (2017) study and serves as a ground-truth for evaluation of typical debugging tasks, as occurs with BL. Similar to our considerations about the bugs' nature, DBGBench reflects the importance of the knowledge about the benchmarks/dataset contents and how it can help clarify and separate what is acceptable from what does not reflect the reality or the practical concerns for the debugging task automation (or the automated support for debugging tasks).

7.9 Final Considerations

The obtained results are promising since different evaluation results would be produced, even for the already known approaches like BLUiR and LR. The same occurs for some features that show other scores depending on the sample. A comprehensive revision of the already published works and approaches under the perspective of the bugs' nature and characteristics would unveil how and what strategies would better fit (or not) to localize each type of bug. Additionally, we should consider conducting research to characterize bug datasets better, quantifying and defining what kind of bugs are present, in what proportion, and how representative these bugs are. Furthermore, since the approaches usually apply different base features, we should study the impact on the features and how it influences the BL approaches considering the bug nature. Finally, APR approaches would be better targeted at specific types of bugs, considering

datasets bug characteristics for sampling and possibly leading to more informed, fair and accurate evaluations.

Chapter 8

Conclusion

Despite the active research and perceived advances in the area of BL (or even APR), much still needs to be done to make approaches more accurate, reliable, and apt to be widely used in software production. The BL using the bug reports as initial query, and using essentially static information to the ranking of suspicious software components, is one of the branches of this research area. This work proposes the development and application of strategies to contribute to BL, leveraging the advances in ML and IR approaches, and providing ideas to experiment with past approaches, under the lens of dataset analysis and with the characterization of its bugs.

We have experimented with LtR techniques (e.g., LambdaMART, SVMRank, and other algorithms), focusing on the tuning and pre-processing of data to improve the results in the state-of-the-art. Many of the previously proposed approaches in the literature do not indicate clearly and punctually why they fail or succeed in the BL process, focusing on highlighting only the overall gains in evaluation performance metrics. Our experiments and results suggest that the analysis of the bugs in the dataset would contribute to 1) the tuning of the algorithms, 2) the approaches implementation guidance, and 3) the understanding of the approaches' capacity. Hence, it would be easier to identify the proposed strategies' competence, weaknesses, and ideal context.

We also defined a new taxonomy to refer to bugs characteristics, especially those related to its patches. Initially, the taxonomy was presented in the Defects4J Dissection study, extended through a tool (ADD) to automate the information extraction and support the characterization, and applied here to characterize LR–dataset, a larger dataset used on the work of Ye et al. (YE; BUNESCU; LIU, 2014), and also object of our study. Finally, we show how and what influence we would observe when we sample a bug dataset according to the bugs' characteristics, focusing on the performance results.

We find significant differences in features scores and also on BL scores, depending on the bug characteristics used as selection criteria for sampling (e.g., the repair patterns we concentrated our effort).

8.1 Final considerations on the research

We developed an experimental package based on Ye, Bunescu e Liu (2016) approach for experimentation with BL alternative strategies, integrating multiple information sources. The Ye, Bunescu e Liu (2016) approaches originally combine a total of 19 distinct features, many of them based on previous work, as occurs with BLUiR (SAHA et al., 2013). As a proof of concept, we conducted a partial replication of Ye et al.'s work and implemented some additional strategies, looking for BL scores improvements. Chapter 4 presented some of the experimented ideas. Preliminary experiments show that, although each feature can contribute to the overall performance of the algorithms, the ability to design and select good features, besides combining and to weight than properly, is essential. Moreover, not all features are crucial to obtain top performance, e.g., introducing the Entropy feature in the preliminarily tested configurations is marginal: 2.21% to 4.9% with MAP; and 0.25% to 3.69% with MRR. By the way, while analyzing overall performance, we observed a role of Entropy in the reduction of overfitting.

Our preliminary results with LtR algorithms reinforce the need to proceed with careful tuning of the approaches, and we also perceive the influence of the dataset used in the process. When compared with our baseline configurations, we have obtained better results in RQ2 from Chapter 5. Our best-tuned setting had a MAP value of 42.75% and an MRR value of 51.36%, while baseline configuration was 40.0% for MAP and 46% for MRR.

We also experimented with pre-processing of bug reports. Results discussed in RQ3 from Chapter 5 present some potential improvement while testing data balance strategies for generating input data to the learning process. We have consistent gains applying different methods of data balance with LambdaMART, obtained increases of 52.3% (QuickRank) and 70.86% (RankLib) over the baseline configuration (poor performing because of the overfitting). Finally, we confirm the previous knowledge about the bugs characteristics and the informed selection of bugs before proceeding with an assessment can contribute to BL, especially on the understanding about over what type of bugs the approach would perform better or worst.

We first studied Defects4J (SOBREIRA et al., 2018), a relatively small dataset (and

benchmark) usually applied in research for BL and APR. We found many recurrences in patches for bugs in Defects4J that culminated in taxonomy for what we refer to as repair actions and repair patterns. Beyond the patterns we have studied, other dimensions associated with patch size in coding lines and spreading give a dimension of the patch's shape, complexity, and span in the source codebase. The Defects4J study served as the basis for ADD tool development (MADEIRAL et al., 2018), targeting the automatic extraction of the patches characterization information, applied in other studies as done by Durieux et al. (2019). In this thesis, we proceed with analyzing and characterizing the LR-dataset, a large dataset of bugs proposed by Ye, Bunescu e Liu (2014). We applied the dimensions defined in our first study with Defects4J with the help of ADD tool to extract the information about repair actions, repair patterns, and other size dimensions associated with the bug patches. Our results confirm the presence of many of the actions and patterns already found in Defects4J. Chapter 6 details this analysis. We also extend the work with the Defects4J dissection, exploring additional information related to the co-occurrences of repair actions and repair patterns that would influence the analysis. Section 6.5 enumerates some of the references to the Defects4J Dissection study, confirming its applicability in different areas, reinforcing our initial hypothesis, and validating the potential application of our findings.

As an unfould of the dataset dissection studies, we argue that bug datasets should not be considered like a black-box while evaluating BL approaches, as occurs in most of the previous approaches before our dissection studies. Therefore, we propose the study of the bugs' nature and characteristics. This knowledge would help to optimize the selection of the techniques for BL, to understand the performance variations, to obtain more practical insights to improve the approaches, and to propose new methods for BL based on more informed decisions. Our work in (SOBREIRA et al., 2018) started to fill the gap in this direction, providing a kind of framework for bugs characterization through its patches. The exposing of the bug characteristics in a bug dataset is the first step towards the enlightenment for a better understanding of how different BL approaches work against bugs of diverse nature. We show that the bugs have distinct characteristics, and this is a relevant issue while reporting results of BL approaches. In Chapter 7 we have confirmed the influence of the sampling selection based on the characteristics of the bug. We focus mainly on repair patterns' presence, based on the most common repair actions associated with these patterns. When contrasting samples with and without a given repair pattern, we found significant statistical differences on scores produced for individual features, and for specific BL approaches, as BLUiR (SAHA et al., 2013), and the LtR–based approach from Ye et al. (YE; BUNESCU; LIU, 2016), considering

the projects AspectJ and Tomcat from LR-dataset (YE; BUNESCU; LIU, 2014). For example, assuming a sample of bugs from Tomcat with *Missing Not-Null Check* repair pattern and applying the BLUiR approach, we found a score increase of 27.22 percentual points in the NDCG@1 metric. Otherside, in a sample without the last pattern, we can perceive a decrease of 16.82 in another measure, Top-10 metric, for the feature ϕ_1. Other analogous findings were detailed in Chapter 7, confirming more differences.

Based on the experience with all the experimental processes, we confirm the importance and the need for an integrated environment for experimenting with BL strategies. Chapter 4 raises many of the factors that would influence the success of a BL strategy, and some ideas partially implemented in our experimental package were applied and discussed in Chapters 5 and 7. Even considering previous approaches implement some of these ideas (most isolated), an integrated environment for experimentation continues as an open problem. While we start some development in this direction with our experimental package and with ideas on Chapter 4 and all the experiments, we have considerable work to do and many challenges to overcome: facilitate the reproduction and comparisons between approaches, to handle the massive amount of data to process and stay scalable, to integrate feature extraction process with ML-based methods, and many others.

8.2 Main Contributions

The main contributions of this work are in:

- Chapter 5, where we discuss most of the proof of concept contributions, especially in sections with answers for the research questions:
  1. RQ2 shows the application of a new feature based on Code Entropy and LtR to produce BL scores;
  2. RQ3 shows the application of different data balance strategies on the training with LtR algorithms to produce BL scores;
  3. RQ4 shows the influence of parameters tuning on LtR–based BL scores;
  4. With the preliminary experiments of Chapter 5 we observe some gains in the performance of LtR algorithms for BL, applying some of the strategies raised in Chapter 4.
- Chapter 4 and Chapter 7, where we present:

1. A new approach to deal with the assessment of BL methods using bug datasets and benchmarks, guided by bug characteristics;
2. A proof of concept showing the influence of the bug types composing a dataset in the assessment of research approaches on typical software development tasks like BL;
3. The experimental package (in development) briefly summarized in Chapter 4: this implementation was essential to test ideas and obtain results for analysis and comparisons of BL approaches;
4. A proof of concept on the integration of diverse sources of information for BL based on (YE; BUNESCU; LIU, 2016) work and proposing the leverage of the knowledge about the bugs' nature composing the target dataset.

- Chapter 6, where we present:

1. A new approach to deal with assessment in datasets and benchmarks for BL: in the dissection of Defects4J, we observed there are commonalities and variabilities in the bug datasets that we should explore to improve BL and other research areas (e.g., APR).
2. We expanded the work with Defects4J Dissection, applying some of those ideas on a larger dataset, confirming the findings in Defects4J, and complementing with new co-occurrences analysis involving repair actions and repair patterns.

- The papers (SOBREIRA et al., 2018) and (MADEIRAL et al., 2018) that qualifies as collaboration work was fundamental for many achievements and previous contributions, especially:

1. A taxonomy to characterize bug datasets in terms of their patches composition;
2. A tool to extract patch characteristics from a bug dataset, e.g., repair action, repair patterns, and size dimensions;

8.3 Future Work

We can conduct additional experiments considering: 1) the same approach from Chapter 7, but with the characteristics of the bugs to explore more repair patterns, repair

actions, and the other dimensions from the dissection analysis of a dataset; 2) exploring different datasets to show if our findings would sustain more contexts; 3) extending the experiments from Chapter 5 for better coverage of LR-dataset; 4) increment the previous extension with the procedures from Chapter 7, so we can test the influence of bugs' characteristics on LtR strategies; 5) incorporating many of the ideas raised ideas in Chapter 4 in the experimental package, e.g., developing a module for classifying and pre-processing bug reports through the analysis of the bug report content and with query rewriting techniques.

More development, refactoring, and architecting are needed so the experimental package would become a practical framework for BL, using most of the raised ideas in Chapter 4 and allowing support assessment of past and new approaches. The scalability, primarily to improve processing time, memory usage, and storage requirements, is the main issue in the experimental package's new architecture. After, we should make it available for further development by the research community.

Some extensions to this work are in progress or planned as 1) a paper to publish the analysis in Chapter 6, showing the extension of Defects4J dissection to LR-dataset, 2) another paper with extensions and additional data from Chapter 7 after the completion of additional experiments to confirm the influence of Repair Patterns and Repair Actions on BL, covering a more significant part of LR-dataset, 3) the additional refactoring and optimization of the experimental package would make it viable to complement and extend the experiments in Chapter 5, especially with considerations about the bug characteristics, similar to the experiments from Chapter 7. We plan another extension 4) to publish a paper exposing the experimental package in detail, complementing what we briefly summarize in Chapter 4, but only after some refactoring and re-architecting to scale better with large datasets and to polish the codebase.

8.4 Bibliographical Production

Our work in (SOBREIRA et al., 2018) exposes many intrinsic properties of bug patches in the Defects4J dataset. Therefore, we can use the insights from that study to clarify how approaches to BL behave depending on the bugs' nature. Until the end of 2021, this paper accounts for more than 80 citations.

Our subsequent work in (MADEIRAL et al., 2018) makes it possible to extend part of the analysis done in (SOBREIRA et al., 2018) to other datasets of bugs beyond Defects4J through the automatic detection of repair patterns. This work was named the

best paper in the 6th Workshop on Software Visualization, Evolution and Maintenance (VEM 2018), co-located with the 9th Brazilian Conference on Software: Theory and Practice (CBSoft'18).

Bibliography

ABREU, R. et al. Refining spectrum-based fault localization rankings. **Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09**, p. 409, 2009. Disponível em: <https://doi.org/10.1145/1529282.1529374>.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. V. On the accuracy of spectrum-based fault localization. In: **Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART-Mutation 2007)**. Windsor, UK: IEEE, 2007. p. 89–98. ISBN 0769529844. ISSN 0014-0139. Disponível em: <https://doi.org/10.1109/TAIC.PART.2007.13>.

AKBAR, S. A.; KAK, A. C. A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization. In: **17th International Conference on Mining Software Repositories**. Seoul, Republic of Korea: ACM, 2020. p. 21–31. ISBN 9781450375177. Disponível em: <https://doi.org/10.1145/3379597.3387474>.

ALMHANA, R. et al. Recommending Relevant Classes for Bug Reports Using Multi-objective Search. In: **31st IEEE/ACM International Conference on Automated Software Engineering**. Singapore: ACM, 2016. (ASE 2016), p. 286–295. ISBN 978-1-4503-3845-5. Disponível em: <http://doi.acm.org/10.1145/2970276.2970344>.

ARCEGA, L.; FONT, J.; CETINA, C. Evolutionary algorithm for bug localization in the reconfigurations of models at runtime. In: **21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018**. Copenhagen, Denmark: ACM, 2018. (MODELS '18), p. 90–100. ISBN 9781450349499. Disponível em: <https://doi.org/10.1145/3239372.3239392>.

ARCEGA, L. et al. Bug Localization in Model-Based Systems in the Wild. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 31, n. 1, 2021. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3472616>.

______. An approach for bug localization in models using two levels: model and metamodel. **Software and Systems Modeling**, v. 18, n. 6, p. 3551–3576, 2019. Disponível em: <https://doi.org/10.1007/s10270-019-00727-y>.

ARTZI, S. et al. Fault localization for dynamic web applications. **IEEE Transactions on Software Engineering**, v. 38, n. 2, p. 314–335, 2012. ISSN 00985589. Disponível em: <https://doi.org/10.1109/TSE.2011.76>.

ASKARUNISA, A.; MANJU, T.; BABU, B. G. Fault Localization for Java Programs Using Probabilistic Program Dependence Graph. **International Journal of Computer Science Issues**, v. 8, n. 6, p. 224–232, 2011. Disponível em: <https://doi.org/10.48550/arXiv.1201.3985>.

BAAH, G.; PODGURSKI, A.; HARROLD, M. Mitigating the confounding effects of program dependences for effective fault localization. In: **Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. Szeged, Hungary: ACM, 2011. p. 146–156. ISBN 9781450304436. Disponível em: <https://doi.org/10.1145/2025113.2025136>.

BALL, T.; LARUS, J. R. Optimally profiling and tracing programs. **ACM Transactions on Programming Languages and Systems**, v. 16, n. 4, p. 1319–1360, 1994. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/183432.183527>.

BARBOSA, J. R. et al. BULNER: BUg Localization with word embeddings and NEtwork Regularization. In: **VII Workshop on Software Visualization, Evolution and Maintenance (VEM '19)**. Porto Alegre, RS, Brasil: [s.n.], 2019. p. 9–16. Disponível em: <https://doi.org/10.48550/arXiv.1908.09876>.

BETTENBURG, N. et al. What Makes a Good Bug Report? In: **16th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. Atlanta, Georgia: ACM, 2008. (SIGSOFT '08/FSE-16), p. 308–318. ISBN 978-1-59593-995-1. Disponível em: <http://doi.acm.org/10.1145/1453101.1453146>.

BHAGWAN, R. et al. Orca: Differential Bug Localization in Large-Scale Services. In: **13th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2018. (OSDI'18), p. 493–509. ISBN 9781931971478. Disponível em: <https://www.usenix.org/conference/osdi18/presentation/bhagwan>.

BÖHME, M. et al. Where is the bug and how is it fixed? an experiment with practitioners. **11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017**, p. 117–128, 2017. Disponível em: <https://doi.org/10.1145/3106237.3106255>.

BREIMAN, L. E. O. Random Forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001. ISSN 08856125. Disponível em: <https://doi.org/10.1023/A:1010933404324>.

BRIAND, L. C.; LABICHE, Y.; LIU, X. Using Machine Learning to Support Debugging with Tarantula. In: **18th IEEE International Symposium on Software Reliability (ISSRE '07)**. Trollhattan, Sweden: IEEE, 2007. (ISSRE'07, January), p. 137–146. ISBN 978-0-7695-3024-6. ISSN 1071-9458. Disponível em: <https://doi.org/10.1109/ISSRE.2007.36>.

BURGES, C. et al. Learning to rank using gradient descent. In: **22nd international conference on Machine learning (ICML'05)**. Bonn, Germany: [s.n.], 2005. p. 89–96. ISBN 1595931805. ISSN 00243205. Disponível em: <https://doi.org/10.1145/1102351.1102363>.

BURGES, C. J. C.; RAGNO, R.; LE, Q. V. Learning to Rank with Nonsmooth Cost Functions. In: **19th International Conference on Neural Information Processing Systems**. MIT Press Cambridge, 2006. v. 19, p. 193–200. ISBN 0262195682. ISSN 10495258. Disponível em: <https://proceedings.neurips.cc/paper/2006/file/af44c4c56f385c43f2529f9b1b018f6a-Paper.pdf>.

CAMPOS, J. et al. GZoltar: an eclipse plug-in for testing and debugging. In: **27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012**. Essen, Germany: Association for Computing Machinery, 2012. p. 378–381. ISBN 9781450312042. Disponível em: <https://doi.org/10.1145/2351676.2351752>.

CAO, J. et al. BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs. In: **35th IEEE/ACM International Conference on Automated Software Engineering**. Virtual Event, Australia: Association for Computing Machinery, 2020. p. 1214–1218. ISBN 9781450367684. Disponível em: <https://doi.org/10.1145/3324884.3418934>.

CAO, Z. et al. Learning to rank: from pairwise approach to listwise approach. In: **24th International Conference on Machine Learning**. Corvalis, Oregon, USA: ACM, 2007. p. 129–136. ISBN 9781595937933. ISSN 1595937935. Disponível em: <https://doi.org/10.1145/1273496.1273513>.

CELLIER, P. et al. Multiple Fault Localization with Data Mining. In: **23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011)**. Miami, USA: HAL, 2011. p. 238–243. ISBN 1891706292. Disponível em: <https://hal.archives-ouvertes.fr/hal-01119562>.

CHAKRABORTY, S. et al. Entropy Guided Spectrum Based Bug Localization Using Statistical Language Model. 2018. Disponível em: <https://doi.org/10.48550/arXiv.1802.06947>.

CHAPARRO, O.; FLOREZ, J. M.; MARCUS, A. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. **Empirical Software Engineering**, v. 24, p. 2947–3007, 2019. ISSN 15737616. Disponível em: <https://doi.org/10.1007/s10664-018-9672-z>.

CHILIMBI, T. M. et al. HOLMES : Effective Statistical Debugging via Efficient Path Profiling. In: **31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2008. (ICSE'09), p. 34–44. ISBN 978-1-4244-3453-4. Disponível em: <https://doi.org/10.1109/ICSE.2009.5070506>.

CHOI, S.-s.; CHA, S.-h. A survey of Binary similarity and distance measures. **Journal of Systemics, Cybernetics and Informatics**, p. 43–48, 2010. Disponível em: <http://www.iiisci.org/journal/sci/Abstract.asp?var=&id=GS315JG>.

CHRISTI, A. et al. Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization. In: **2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. Memphis, TN, USA: IEEE, 2018. p. 184–191. Disponível em: <https://doi.org/10.1109/ISSREW.2018.00005>.

COUSOT, P.; COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: **4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages**. Los Angeles, California: ACM, 1977. (POPL '77), p. 238–252. ISSN 00900036. Disponível em: <http://doi.acm.org/10.1145/512950.512973>.

COUTANT, D. S. D. et al. DOC: A practical approach to source-level debugging of globally optimized code. **ACM SIGPLAN Notices**, v. 23, n. 7, p. 125–134, 1988. ISSN 15581160. Disponível em: <https://doi.org/10.1145/960116.54003>.

DALLMEIER, V.; LINDIG, C.; ZELLER, A. Lightweight Bug Localization with AMPLE. In: **6th International Symposium on Automated Analysis-driven Debugging**. Monterey, California, USA: ACM, 2005. (AADEBUG'05), p. 99–104. ISBN 1-59593-050-7. Disponível em: <https://doi.org/10.1145/1085130.1085143>.

DALLMEIER, V.; ZIMMERMANN, T. Extraction of bug localization benchmarks from history. In: **22nd IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2007. (ASE '07), p. 433–436. ISBN 978-1-59593-882-4. Disponível em: <https://doi.org/10.1145/1321631.1321702>.

DAO, T.; ZHANG, L.; MENG, N. How Does Execution Information Help with Information Retrieval Based Bug Localization? An Extensive Study. In: **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. Buenos Aires, Argentina: IEEE, 2017. p. 241–250. ISBN 1234567245. Disponível em: <https://doi.org/10.1109/ICPC.2017.29%0A>.

DEMARCO, F. et al. Automatic repair of buggy if conditions and missing preconditions with SMT. In: **Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis - CSTVA 2014**. Hyderabad, India: ACM, 2014. p. 30–39. ISBN 9781450328470. Disponível em: <https://doi.org/10.1145/2593735.2593740>.

DIGIUSEPPE, N.; JONES, J. A. Fault density, fault types, and spectra-based fault localization. **Empirical Software Engineering**, v. 20, n. 4, p. 928–967, 2015. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-014-9304-1>.

DILSHENER, T. **Improving Information Retrieval Based Bug Localisation Using Contextual Heuristics**. Tese (Doutorado) — The Open University, 2016. Disponível em: <https://doi.org/10.21954/ou.ro.0000c41c>.

DILSHENER, T.; WERMELINGER, M.; YU, Y. Locating Bugs Without Looking Back. In: **13th International Conference on Mining Software Repositories**. Austin, Texas: ACM, 2016. (MSR '16), p. 286–290. ISBN 978-1-4503-4186-8. Disponível em: <https://doi.org/10.1145/2901739.2901775>.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. **Empirical Software Engineering**, v. 10, n. 4, p. 405–435, 2005. ISSN 13823256. Disponível em: <https://doi.org/10.1007/s10664-005-3861-2>.

DURIEUX, T. et al. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In: **27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. Tallinn, Estonia: ACM, 2019. p. 302–313. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3338911>.

EDWARDS, J. C. **Method, system, and program for logging statements to monitor execution of a program**. 2003. Disponível em: <https://patents.google.com/patent/US6539501B1/en>.

FALLERI, J.-R. et al. Fine-grained and accurate source code differencing. In: **ACM/IEEE International Conference on Automated Software Engineering, ASE'14**. Vasteras, Sweden: ACM/IEEE, 2014. p. 313–324. ISBN 9781450330138. Disponível em: <http://doi.acm.org/10.1145/2642937.2642982>.

FREUND, Y. et al. An Efficient Boosting Algorithm for Combining Preferences. **The Journal of Machine Learning Research**, v. 4, p. 933–969, 2003. ISSN 0003-6951. Disponível em: <https://dl.acm.org/doi/10.5555/945365.964285>.

FRIEDMAN, J. H. . Greedy Function Approximation: A Gradient Boosting Machine. **The Annals of Statistics**, v. 29, n. 5, p. 1189–1232, 2001. Disponível em: <https://doi.org/10.1214/aos/1013203451>.

GAZZOLA, L.; MICUCCI, D.; MARIANI, L. Automatic Software Repair: A Survey. **IEEE Transactions on Software Engineering**, IEEE, v. 45, n. 1, p. 34–67, 2019. ISSN 19393520. Disponível em: <https://doi.org/10.1109/TSE.2017.2755013>.

GONG, C. et al. Effects of Class Imbalance in Test Suites: An Empirical Study of Spectrum-Based Fault Localization. In: **36th Annual Computer Software and Applications Conference Workshops**. Izmir, Turkey: IEEE, 2012. p. 470–475. ISBN 978-1-4673-2714-5. Disponível em: <https://doi.org/10.1109/COMPSACW.2012.89>.

GOUES, C. L. et al. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. **IEEE Transactions on Software Engineering**, v. 41, n. 12, p. 1236–1256, 2015. ISSN 00985589. Disponível em: <https://doi.org/10.1109/TSE.2015.2454513>.

HAMILL, M.; GOSEVA-POPSTOJANOVA, K. Analyzing and predicting effort associated with finding and fixing software faults. **Information and Software Technology**, v. 87, p. 1–18, 2017. ISSN 09505849. Disponível em: <https://doi.org/10.1016/j.infsof.2017.01.002>.

HE, H.; GARCIA, E. A. Learning from imbalanced data. **IEEE Transactions on Knowledge and Data Engineering**, v. 21, n. 9, p. 1263–1284, 2009. ISSN 10414347. Disponível em: <https://doi.org/10.1109/TKDE.2008.239>.

HELLENDOORN, V. J. Are Deep Neural Networks the Best Choice for Modeling Source Code? In: **11th Joint Meeting on Foundations of Software Engineering**. Paderborn, Germany: ACM, 2017. p. 763–773. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3106290>.

HOANG, T. V. D. et al. Network-Clustered Multi-Modal Bug Localization. **IEEE Transactions on Software Engineering**, v. 45, p. 1002–1023, 2018. ISSN 00985589. Disponível em: <https://doi.org/10.1109/TSE.2018.2810892>.

HUO, X.; LI, M. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: **26th International Joint Conference on Artificial Intelligence**. Melbourne, Australia: IJCAI, 2017. p. 1909–1915. ISBN 9780999241103. ISSN 10450823. Disponível em: <https://doi.org/10.24963/ijcai.2017/265>.

HUO, X.; LI, M.; ZHOU, Z. H. Learning unified features from natural and programming languages for locating buggy source code. In: **25h International Joint Conference on Artificial Intelligence**. New York, New York, USA: AAAI Press, 2016. p. 1606–1612. ISBN 978-1-57735-770-4. ISSN 10450823. Disponível em: <https://dl.acm.org/doi/10.5555/3060832.3060845>.

HUO, X. et al. Deep Transfer Bug Localization. **IEEE Transactions on Software Engineering**, IEEE, v. 47, n. 7, p. 1368 – 1380, 2019. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2019.2920771>.

HUTCHINS, M. et al. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: **16th International Conference on Software Engineering**. Sorrento, Italy: IEEE, 1994. p. 191–200. Disponível em: <https://doi.org/10.1109/ICSE.1994.296778>.

ISO/IEC/IEEE. **Systems and software engineering – Vocabulary**. 2010. 410 p. Disponível em: <https://www.iso.org/standard/71952.htmlhttp://www.iso.org/iso/catalogue_detail.htm?csnumber=50518>.

JANSSEM, T.; ABREU, R.; GEMUND, A. J. C. V. Zoltar: A toolset for automatic fault localization. In: **24th IEEE/ACM International Conference on Automated Software Engineering**. Auckland, New Zealand: IEEE, 2009. p. 662–664. ISBN 9780769538914. ISSN 1938-4300. Disponível em: <https://doi.org/10.1109/ASE.2009.27>.

JÄRVELIN, K.; KEKÄLÄINEN, J. Cumulated Gain-Based Evaluation of IR Techniques. **ACM Transactions on Information Systems**, v. 20, n. 4, p. 422–446, 2002. Disponível em: <https://doi.org/10.1145/582415.582418>.

JEFFREY, D. et al. BugFix: A learning-based tool to assist developers in fixing bugs. In: **17th International Conference on Program Comprehension**. Vancouver, BC, Canada: IEEE, 2009. p. 70–79. ISBN 9781424439973. ISSN 1063-6897. Disponível em: <https://doi.org/10.1109/ICPC.2009.5090029>.

JOACHIMS, T. Training Linear SVMs in Linear Time. In: **12th ACM SIGKDD international conference on Knowledge discovery and data mining**. Philadelphia, Pennsylvania, USA: ACM, 2006. p. 217–226. ISBN 1595933395. Disponível em: <https://doi.org/10.1145/1150402.1150429>.

JONES, J. A. J. J. a.; HARROLD, M. J. M. Empirical evaluation of the tarantula automatic fault-localization technique. In: **20th IEEE/ACM International Conference on Automated Software Engineering**. Long Beach, CA, USA: ACM, 2005. (ASE '05), p. 282–292. ISBN 1581139934. Disponível em: <http://doi.acm.org/10.1145/1101908.1101949http://portal.acm.org/citation.cfm?id=1101949>.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: **2014 International Symposium on Software Testing and Analysis**. San Jose, CA, USA: ACM, 2014. (ISSTA 2014), p. 437–440. ISBN 978-1-4503-2645-2. Disponível em: <https://doi.org/10.1145/2610384.2628055>.

JUST, R. et al. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In: **27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018**. Amsterdam, Netherlands: ACM, 2018. p. 287–297. ISBN 9781450356992. Disponível em: <https://doi.org/10.1145/3213846.3213870>.

KAUFMAN, S. et al. Leakage in data mining: Formulation, detection, and avoidance. **ACM Transactions on Knowledge Discovery from Data**, v. 6, n. 4, p. 1–21, 2012. ISSN 15564681. Disponível em: <https://doi.org/10.1145/2382577.2382579>.

KEYHANIPOUR, A. H.; MOEINI, A. Learning to rank with click-through features in a reinforcement learning framework. **International Journal of Web Information Systems**, v. 12, n. 4, p. 448–476, 2016. Disponível em: <https://doi.org/10.1108/IJWIS-12-2015-0046>.

KHAN, T. A.; SULLIVAN, A.; WANG, K. AlloyFL: A Fault Localization Framework for Alloy. In: **29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. Athens, Greece: ACM, 2021. (ESEC/FSE 2021), p. 1535–1539. ISBN 9781450385626. Disponível em: <https://dl.acm.org/doi/10.1145/3468264.3473116>.

KHATIWADA, S.; TUSHEV, M.; MAHMOUD, A. On Combining IR Methods to Improve Bug Localization. In: **28th International Conference on Program Comprehension**. Seoul, Korea: ACM, 2020. p. 252–262. ISBN 9781450379588. Disponível em: <https://doi.org/10.1145/3387904.3389280>.

KIDWELL, P. A. Stalking the Elusive Computer Bug. **IEEE Ann. Hist. Comput.**, IEEE Educational Activities Department, USA, v. 20, n. 4, p. 5–9, 1998. ISSN 1058-6180. Disponível em: <https://doi.org/10.1109/85.728224>.

KIM, D. et al. Where should we fix this bug? A two-phase recommendation model. **IEEE Transactions on Software Engineering**, v. 39, n. 11, p. 1597–1610, 2013. ISSN 00985589. Disponível em: <https://doi.org/10.1109/TSE.2013.24>.

KIM, M.; LEE, E. Poster: Are information retrieval-based bug localization techniques trustworthy? In: **40th International Conference on Software Engineering: Companion Proceeedings**. Gothenburg, Sweden: ACM, 2018. p. 248–249. ISBN 9781450356633. ISSN 02705257. Disponível em: <https://doi.org/10.1145/3183440.3194954>.

______. A novel approach to automatic query reformulation for IR-based bug localization. In: **34th ACM/SIGAPP ACM Symposium on Applied Computing**. Limassol, Cyprus: ACM, 2019. p. 1752–1759. ISBN 9781450359337. Disponível em: <https://doi.org/10.1145/3297280.3297451>.

KOCHHAR, P. S. et al. Practitioners' expectations on automated fault localization. In: **25th International Symposium on Software Testing and Analysis - ISSTA 2016**. Saarbrücken, German: ACM, 2016. p. 165–176. ISBN 9781450343909. Disponível em: <https://doi.org/10.1145/2931037.2931051>.

KOYUNCU, A. et al. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. 2019. Disponível em: <https://doi.org/10.48550/arXiv.1902.02703>.

KUMA, T. et al. Improving the Accuracy of Spectrum-Based Fault Localization for Automated Program Repair. In: **Proceedings of the 28th International Conference on Program Comprehension**. Seoul, Korea: ACM, 2020. p. 376–380. ISBN 9781450379588. Disponível em: <https://doi.org/10.1145/3387904.3389290>.

LADYZYNSKI, P.; ZBIKOWSKI, K.; GRZEGORZEWSKI, P. Stock trading with random forests, trend detection tests and force index volume indicators. **Artificial Intelligence and Soft Computing**, Berlin, Heidelberg, v. 7895, p. 441–452, 2013. ISSN 03029743. Disponível em: <https://doi.org/10.1007/978-3-642-38610-7_41>.

LAM, A. N. et al. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports. In: **30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. Lincoln, NE, USA: IEEE, 2015. p. 476–481. ISBN 9781509000258. Disponível em: <https://doi.org/10.1109/ASE.2015.73>.

______. Bug Localization with Combination of Deep Learning and Information Retrieval. In: **25th International Conference on Program Comprehension (ICPC)**. Buenos Aires, Argentina: IEEE, 2017. p. 218–229. ISBN 9781538605356. ISSN 1524-4563. Disponível em: <https://doi.org/10.1109/ICPC.2017.24>.

LE, T.-D. B. et al. A learning-to-rank based fault localization approach using likely invariants. In: **25th International Symposium on Software Testing and Analysis**. Saarbrücken Germany: ACM, 2016. p. 177–188. ISBN 9781450343909. Disponível em: <https://doi.org/10.1145/2931037.2931049>.

LE, T.-D. B.; OENTARYO, R. J.; LO, D. Information Retrieval and Spectrum Based Bug Localization: Better Together. In: **10th Joint Meeting on Foundations of Software Engineering**. Bergamo, Italy: ACM, 2015. (ESEC/FSE 2015, 65), p. 579–590. ISBN 978-1-4503-3675-8. Disponível em: <http://doi.acm.org/10.1145/2786805.2786880>.

LEE, J. et al. Bench4BL: Reproducibility Study on the Performance of IR-Based Bug Localization. In: **27th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Amsterdam, Netherlands: ACM, 2018. (ISSTA 2018, 2), p. 61–72. ISBN 9781450356992. Disponível em: <https://doi.org/10.1145/3213846.3213856>.

LEX, A. et al. UpSet: Visualization of intersecting sets. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 20, n. 12, p. 1983–1992, 2014. ISSN 10772626. Disponível em: <https://doi.org/10.1109/TVCG.2014.2346248>.

LI, X. et al. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In: **28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Beijing, China: ACM, 2019. p. 284–295. ISBN 9781450362245. Disponível em: <https://doi.org/10.1145/3339068>.

LI, X.; ZHANG, L. Transforming Programs and Tests in Tandem for Fault Localization. **Proceedings of the ACM on Programming Languages**, v. 1, n. OOPSLA, p. 92:1–92:30, 2017. Disponível em: <https://doi.org/10.1145/3133916>.

LI, Y.; WANG, S.; NGUYEN, T. N. Fault Localization with Code Coverage Representation Learning. In: **IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. Madrid, ES: IEEE Press, 2021. p. 661–673. ISBN 9781450390859. Disponível em: <https://doi.org/10.1109/ICSE43902.2021.00067>.

LIANG, H. et al. Deep Learning With Customized Abstract Syntax Tree for Bug Localization. **IEEE Access**, IEEE, v. 7, p. 116309–116320, 2019. Disponível em: <https://doi.org/10.1109/ACCESS.2019.2936948>.

LIU, K. et al. A Closer Look at Real-World Patches. In: **34th IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Madrid, Spain: IEEE, 2018. p. 275–286. Disponível em: <https://doi.org/10.1109/ICSME.2018.00037>.

_____. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. **12th IEEE Conference on Software Testing, Validation and Verification (ICST)**, IEEE, Xi'an, China, n. January 2019, p. 102–113, 2019. Disponível em: <https://doi.org/10.1109/ICST.2019.00020>.

LIU, T.-Y. Learning to Rank for Information Retrieval. **Foundations and Trends in Information Retrieval**, v. 3, n. 3, p. 225–331, 2009. ISSN 1554-0669. Disponível em: <https://doi.org/10.1561/1500000016>.

LOU, Y. et al. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In: **29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Virtual Event, USA: ACM, 2020. (ISSTA 2020), p. 75–87. ISBN 9781450380089. Disponível em: <https://doi.org/10.1145/3395363.3397351>.

_____. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning. In: **29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. Athens, Greece: ACM, 2021. (ESEC/FSE 2021), p. 664–676. ISBN 9781450385626. Disponível em: <https://doi.org/10.1145/3468264.3468580>.

LOYOLA, P.; GAJANANAN, K.; SATOH, F. Bug localization by learning to rank and represent bug inducing changes. **27th ACM International Conference on Information and Knowledge Management**, ACM, Torino, Italy, p. 657–665, 2018. Disponível em: <https://doi.org/10.1145/3269206.3271811>.

LU, S. et al. BugBench: Benchmarks for Evaluating Bug Detection Tools. **Proc of the Workshop on the Evaluation of Software Defect Detection Tools**, n. 3, p. 1–5, 2005. Disponível em: <https://researchr.org/publication/Lu05bugbench%3Abenchmarks>.

LUCCHESE, C. et al. Selective Gradient Boosting for Effective Learning to Rank. In: **41st International ACM SIGIR Conference on Research & Development in Information Retrieval**. Ann Arbor, MI, USA: ACM, 2018. p. 155–164. ISBN 9781450356572. Disponível em: <https://doi.org/10.1145/3209978.3210048>.

LUCIA, L. et al. Are Faults Localizable? In: **9th IEEE Working Conference on Mining Software Repositories**. Zurich, Switzerland: IEEE Press, 2012. (MSR '12), p. 74–77. ISBN 978-1-4673-1761-0. ISSN 21601852. Disponível em: <https://dl.acm.org/doi/10.5555/2664446.2664457>.

MADEIRAL, F. et al. Towards an automated approach for bug fix pattern detection. In: **VI Workshop on Software Visualization, Evolution and Maintenance (VEM '18)**. São Carlos, SP, Brazil: SBC, 2018. Disponível em: <https://vem2018.github.io/proceedings/VEM2018-Proceedings.pdf>.

MAGOUN, A. B.; ISRAEL, P. Did You Know? Edison Coined the Term "Bug". **The Institute, The latest news about IEEE, its members, tech history, and new offerings**, 8 2013. Disponível em: <https://spectrum.ieee.org/did-you-know-edison-coined-the-term-bug>.

MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **An Introduction to Information Retrieval**. 1st. ed. Cambridge, England: Cambridge University Press, 2008. 482 p. ISBN 978-0521865715.

MCCAULEY, R. et al. Debugging: a review of the literature from an educational perspective. **Computer Science Education**, v. 18, n. 2, p. 67–92, 2008. ISSN 0899-3408. Disponível em: <https://doi.org/10.1080/08993400802114581>.

METZLER, D.; CROFT, W. B. Linear Feature-Based Models for Information Retrieval. **Information Retrieval**, Kluwer Academic Publishers, v. 10, n. 3, p. 257–274, 2006. Disponível em: <https://doi.org/10.1007/s10791-006-9019-z>.

MILLS, C. et al. Are Bug Reports Enough for Text Retrieval-Based Bug Localization? In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Madrid, Spain: IEEE, 2018. p. 381–392. ISBN 978-1-5386-7870-1. Disponível em: <https://doi.org/10.1109/ICSME.2018.00046>.

MITCHELL, R. L. **Y2K: The good, the bad and the crazy**. Computer World, 2009. Disponível em: <http://www.computerworld.com/article/2522197/it-management/y2k--the-good--the-bad-and-the-crazy.html>.

MONPERRUS, M. Automatic Software Repair: A Bibliography. **ACM Computing Surveys**, ACM, v. 51, n. 1, p. 24, 2018. ISSN 10488251. Disponível em: <https://doi.org/10.1145/3105906>.

MOON, S. et al. Ask the Mutants: Mutating faulty programs for fault localization. **7th International Conference on Software Testing, Verification and Validation (ICST 2014)**, IEEE, Cleveland, OH, USA, p. 153–162, 2014. ISSN 2159-4848. Disponível em: <https://doi.org/10.1109/ICST.2014.28>.

MORENO, L. et al. On the use of stack traces to improve text retrieval-based bug localization. In: **30th International Conference on Software Maintenance and Evolution, ICSME 2014**. Victoria, BC, Canada: IEEE, 2014. p. 151–160. ISBN 9780769553030. ISSN 1063-6773. Disponível em: <https://doi.org/10.1109/ICSME.2014.37>.

MOTWANI, M. et al. Do automated program repair techniques repair hard and important bugs? **Empirical Software Engineering**, v. 23, n. 5, p. 2901–2947, 2018. ISSN 15737616. Disponível em: <https://doi.org/10.1007/s10664-017-9550-0>.

NAISH, L.; LEE, H. J.; RAMAMOHANARAO, K. A model for spectra-based software diagnosis. **ACM Transactions on Software Engineering and**

Methodology, ACM, v. 20, n. 3, p. 1–32, 2011. ISSN 1049331X. Disponível em: <https://doi.org/10.1145/2000791.2000795>.

NAMIN, A. S. Statistical Fault Localization Based on Importance Sampling. In: **14th International Conference on Machine Learning and Applications (ICMLA)**. Miami, FL, USA: [s.n.], 2015. p. 58–63. ISBN 978-1-5090-0287-0. Disponível em: <https://doi.org/10.1109/ICMLA.2015.91>.

NAYROLLES, M.; HAMOU-LHADJ, A. Towards a Classification of Bugs to Facilitate Software Maintainability Tasks. In: **1st International Workshop on Software Qualities and Their Dependencies**. Gothenburg, Sweden: ACM, 2018. p. 25–32. ISBN 9781450357418. Disponível em: <https://doi.org/10.1145/3194095.3194101>.

NGUYEN, A. T. et al. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In: **26th IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 263–272. ISBN 978-1-4577-1638-6. Disponível em: <https://doi.org/10.1109/ASE.2011.6100062>.

NICHOLS, B. D. Augmented Bug Localization Using Past Bug Information. In: **48th Annual Southeast Regional Conference**. Oxford, Mississippi: ACM, 2010. (ACM SE '10), p. 61:1–61:6. ISBN 978-1-4503-0064-3. Disponível em: <https://doi.org/10.1145/1900008.1900090>.

PAN, S. J.; FELLOW, Q. Y. A Survey on Transfer Learning. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 22, n. 10, p. 1–15, 2009. Disponível em: <https://doi.org/10.1109/TKDE.2009.191>.

PAPADAKIS, M.; TRAON, Y. L. Metallaxis-FL: mutation-based fault localization Mike. **Software Testing Verification and Reliability**, v. 25, n. JUN., p. 605–628, 2015. ISSN 0008350X. Disponível em: <https://doi.org/10.1002/stvr.1509>.

PARNIN, C.; ORSO, A. Are Automated Debugging Techniques Actually Helping Programmers? In: **2011 International Symposium on Software Testing and Analysis**. Toronto, Ontario, Canada: ACM, 2011. (ISSTA'11), p. 199–209. ISBN 978-1-4503-0562-4. Disponível em: <https://doi.org/10.1145/2001420.2001445>.

PAWLAK, R. et al. SPOON: A library for implementing analyses and transformations of Java source code. **Software: Practice and Experience**, v. 46, n. 9, p. 1155–1179, 2015. Disponível em: <https://doi.org/10.1002/spe.2346>.

PEARSON, S. et al. Evaluating & improving fault localization techniques. In: **IEEE/ACM 39th International Conference on Software Engineering (ICSE 2017)**. Buenos Aires, Argentina: IEEE, 2017. ISBN 9781538638682. Disponível em: <https://doi.org/10.1109/ICSE.2017.62>.

PEREZ, A. et al. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. In: **IEEE/ACM 39th International Conference**

on Software Engineering, ICSE 2017. Buenos Aires, Argentina: IEEE, 2017. p. 1558–1225. ISBN 9781538638682. ISSN 1872-5392. Disponível em: <https://doi.org/10.1109/ICSE.2017.66%0A>.

POLISETTY, S.; MIRANSKYY, A.; BA\CSAR, A. On Usefulness of the Deep-Learning-Based Bug Localization Models to Practitioners. In: **15th International Conference on Predictive Models and Data Analytics in Software Engineering**. Recife, Brazil: ACM, 2019. (PROMISE'19), p. 16–25. ISBN 9781450372336. Disponível em: <https://doi.org/10.1145/3345629.3345632>.

POSHYVANYK, D. et al. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. **IEEE Transactions on Software Engineering**, v. 33, n. 6, p. 420–432, 2007. Disponível em: <https://doi.org/10.1109/TSE.2007.1016>.

PRADEL, M. et al. Scaffle: Bug Localization on Millions of Files. In: **29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Virtual Event, USA: ACM, 2020. (ISSTA 2020), p. 225–236. ISBN 9781450380089. Disponível em: <https://doi.org/10.1145/3395363.3397356>.

PYTLIK, B. et al. Automated Fault Localization Using Potential Invariants. In: **Fifth Int. Workshop on Automated and Algorithmic Debugging**. Ghent, Belgium: Arxiv, 2003. p. 273–276. ISBN 158113472X. Disponível em: <http://arxiv.org/abs/cs/0310040>.

QI, B. et al. DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization. **IEEE Transactions on Reliability**, v. 71, n. 1, p. 235 – 249, 2022. Disponível em: <https://doi.org/10.1109/TR.2021.3104728>.

QI, Z. et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: **2015 International Symposium on Software Testing and Analysis (ISSTA 2015)**. Baltimore, MD, USA: ACM, 2015. p. 24–36. ISBN 9781450336208. Disponível em: <https://doi.org/10.1145/2771783.2771791>.

RAHMAN, F. et al. Comparing Static Bug Finders and Statistical Prediction. In: **36th International Conference on Software Engineering**. Hyderabad, India: ACM, 2014. (ICSE 2014), p. 424–434. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568269>.

RAHMAN, M. M.; ROY, C. K. Improving IR-Based Bug Localization with Context-Aware Query Reformulation. In: **40th International Conference on Software Engineering: Companion Proceeedings**. Gothenburg, Sweden: ACM, 2018. p. 348–349. ISBN 9781450355735. Disponível em: <https://doi.org/10.1145/3183440.3195003>.

RASELIMO, M.; FISCHER, B. Spectrum-Based Fault Localization for Context-Free Grammars. In: **12th ACM SIGPLAN International Conference on Software**

Language Engineering. Athens, Greece: ACM, 2019. (SLE 2019), p. 15–28. ISBN 9781450369817. Disponível em: <https://doi.org/10.1145/3357766.3359538>.

RATH, M.; LO, D.; MÄDER, P. Analyzing requirements and traceability information to improve bug localization. In: **15th International Conference on Mining Software Repositories (MSR '18)**. Gothenburg, Sweden: ACM, 2018. p. 442–453. ISBN 9781450357166. Disponível em: <https://doi.org/10.1145/3196398.3196415>.

RATH, M.; MÄDER, P. Influence of structured information in bug report descriptions on ir-based bug localization. In: **44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018**. Prague, Czech Republic: IEEE, 2018. p. 26–32. ISBN 9781538673829. Disponível em: <https://doi.org/10.1109/SEAA.2018.00014>.

RAY, B. et al. On the "Naturalness" of Buggy Code. In: **38th International Conference on Software Engineering**. Austin, Texas: ACM, 2016. ISBN 9781450339001. ISSN 02705257. Disponível em: <https://doi.org/10.1145/2884781.2884848>.

RIPLEY, B. D.; HJORT, N. L. **Pattern Recognition and Neural Networks**. 1st. ed. New York, NY, USA: Cambridge University Press, 1995. ISBN 0521460867.

ROSENBLUM, S.; ROSENBLUM, D. S. A Practical Approach to Programming With Assertions. **IEEE Transactions on Software Engineering**, v. 21, n. 1, p. 19–31, 1995. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/32.341844>.

SAHA, R. K. et al. Improving bug localization using structured information retrieval. In: **28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)**. Silicon Valley, CA, USA: IEEE, 2013. p. 345–355. ISBN 9781479902156. Disponível em: <https://doi.org/10.1109/ASE.2013.6693093>.

SAHA, R. K.; SAHA, A. K.; PERRY, D. E. Toward understanding the causes of unanswered questions in software information sites: a case study of stack overflow. In: **9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)**. Saint Petersburg, Russia: ACM, 2013. p. 663. ISBN 9781450322379. Disponível em: <https://doi.org/10.1145/2491411.2494585>.

SASSO, T. D.; MOCCI, A.; LANZA, M. What Makes a Satisficing Bug Report? In: **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. Vienna, Austria: IEEE, 2016. p. 164–174. ISBN 9781509041275. Disponível em: <https://doi.org/10.1109/QRS.2016.28>.

SHI, P. et al. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization XIAOYUAN. **ACM Transactions on Software Engineering and Methodology**, v. 22, n. 4, p. 1–31, 2013. ISSN 10069313. Disponível em: <https://doi.org/10.1145/2522920.2522924>.

SHI, Z. et al. Comparing learning to rank techniques in hybrid bug localization. **Applied Soft Computing Journal**, Elsevier B.V., v. 62, p. 636–648, 2018. ISSN 15684946. Disponível em: <https://doi.org/10.1016/j.asoc.2017.10.048%0A>.

SHI, Z.; KEUNG, J.; SONG, Q. An Empirical Study of BM25 and BM25F Based Feature Location Techniques. In: **International Workshop on Innovative Software Development Methodologies and Practices**. Hong Kong, China: ACM, 2014. (InnoSWDev 2014), p. 106–114. ISBN 978-1-4503-3226-2. Disponível em: <https://doi.org/10.1145/2666581.2666594>.

SIEGMUND, B. et al. Studying the advancement in debugging practice of professional software developers. **Software Quality Journal**, Springer US, v. 25, p. 83–110, 2014. ISSN 15731367. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6983851>.

SILVA-JUNIOR, D. et al. Data-Flow-Based Evolutionary Fault Localization. In: **35th Annual ACM Symposium on Applied Computing**. Brno Czech Republic: Association for Computing Machinery, 2020. p. 1963–1970. ISBN 9781450368667. Disponível em: <https://doi.org/10.1145/3341105.3373946>.

SINHA, V. S.; MANI, S.; MUKHERJEE, D. Is text search an effective approach for fault localization: A practitioners perspective. In: **ACM Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12)**. Tucson, Arizona, USA: ACM, 2012. p. 159–170. ISBN 9781450315630. Disponível em: <https://doi.org/10.1145/2384716.2384770>.

SISMAN, B.; KAK, A. C. Incorporating version histories in Information Retrieval based bug localization. In: **9th IEEE Working Conference on Mining Software Repositories**. Zurich, Switzerland: IEEE, 2012. (MSR '12), p. 50–59. ISBN 9781467317610. ISSN 21601852. Disponível em: <https://doi.org/10.1109/MSR.2012.6224299>.

SOBREIRA, V. et al. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In: **25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Campobasso, Italy: IEEE, 2018. p. 130–140. ISBN 9781538649695. Disponível em: <https://doi.org/10.1109/SANER.2018.8330203>.

SOHN, J. et al. Assisting Bug Report Assignment Using Automated Fault Localisation: An Industrial Case Study. In: **2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)**. Porto de Galinhas, Brazil: IEEE, 2021. p. 284–294. Disponível em: <https://doi.org/10.1109/ICST49551.2021.00041>.

SOHN, J.; YOO, S. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In: **26th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Santa Barbara, CA, USA: Association for Computing Machinery, 2017. (ISSTA 2017), p. 273–283. ISBN 9781450350761. Disponível em: <https://doi.org/10.1145/3092703.3092717>.

SOTO, M. et al. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In: **13th International Conference on Mining Software Repositories**. Austin, Texas: ACM, 2016. (MSR '16), p. 512–515. ISBN 978-1-4503-4186-8. Disponível em: <http://doi.acm.org/10.1145/2901739.2903495>.

SPEER, R.; CHIN, J.; HAVASI, C. ConceptNet 5.5: An Open Multilingual Graph of General Knowledge. In: **31th AAAI Conference on Artificial Intelligence (AAAI-17)**. San Francisco, California, USA: AAAI, 2017. p. 4444–4451. ISBN 0142-9612 (Print). ISSN 0378-7753. Disponível em: <https://dl.acm.org/doi/10.5555/3298023.3298212>.

SUMNER, W. N.; ZHANG, X. Comparative Causality: Explaining the Differences between Executions. In: **2013 International Conference on Software Engineering**. San Francisco, CA, USA: IEEE, 2013. (ICSE'13), p. 272–281. ISBN 9781467330749. Disponível em: <https://doi.org/10.1109/ICSE.2013.6606573>.

TAN, S. H. et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In: **IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)**. Buenos Aires, Argentina: IEEE, 2017. p. 180–182. Disponível em: <https://doi.org/10.1109/ICSE-C.2017.76>.

TASHMAN, L. J. Out-of-sample tests of forecasting accuracy: An analysis and review. **International Journal of Forecasting**, v. 16, n. 4, p. 437–450, 2000. ISSN 01692070. Disponível em: <https://doi.org/10.1016/S0169-2070(00)00065-0>.

THOMPSON, G.; SULLIVAN, A. K. ProFL: A Fault Localization Framework for Prolog. In: **Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Virtual Event, USA: Association for Computing Machinery, 2020. (ISSTA 2020), p. 561–564. ISBN 9781450380089. Disponível em: <https://doi.org/10.1145/3395363.3404367>.

TIAN, Y.; LO, D. A Comparative Study on the Effectiveness of Part-of-Speech Tagging Techniques on Bug Report. In: **22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. Montreal, QC, Canada: IEEE, 2015. p. 570–574. ISBN 9781479984695. Disponível em: <https://doi.org/10.1109/SANER.2015.7081879>.

TOMASSI, D. A. et al. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In: **41st International Conference on Software Engineering (ICSE)**. Montreal, QC, Canada: IEEE, 2019. v. 2019-May. ISBN 9781728108698. ISSN 02705257. Disponível em: <https://doi.org/10.1109/ICSE.2019.00048>.

TU, F. et al. Be Careful of When: An Empirical Study on Time-Related Misuse of Issue Tracking Data. In: **ESEC/FSE 2018**. Lake Buena Vista, FL, USA: ACM, 2018. p. 307–318. ISBN 9781450355735. Disponível em: <https://doi.org/10.1145/3236024.3236054>.

TU, Z.; SU, Z.; DEVANBU, P. On the localness of software. In: **22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014**. Hong Kong, China: ACM, 2014. p. 269–280. ISBN 9781450330565. Disponível em: <https://doi.org/10.1145/2635868.2635875>.

UNENO, Y.; MIZUNO, O.; CHOI, E.-h. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In: **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. Vienna, Austria: IEEE, 2016. ISBN 9781509041275. Disponível em: <https://doi.org/10.1109/QRS.2016.30>.

WANG, Q.; PARNIN, C.; ORSO, A. Evaluating the Usefulness of IR-based Fault Localization Techniques. In: **2015 International Symposium on Software Testing and Analysis**. Baltimore, MD, USA: ACM, 2015. (ISSTA 2015), p. 1–11. ISBN 978-1-4503-3620-8. Disponível em: <https://doi.org/10.1145/2771783.2771797>.

WANG, S. et al. Bugram: Bug Detection with N-gram Language Models. In: **31st IEEE/ACM International Conference on Automated Software Engineering**. Singapore: ACM, 2016. (ASE 2016), p. 708–719. ISBN 978-1-4503-3845-5. Disponível em: <http://doi.acm.org/10.1145/2970276.2970341>.

WANG, S.; LO, D. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In: **22nd International Conference on Program Comprehension**. Hyderabad, India: ACM, 2014. (ICPC 2014), p. 53–63. ISBN 9781450328791. Disponível em: <http://doi.acm.org/10.1145/2597008.2597148>.

______. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. **Journal of Software: Evolution and Process**, v. 28, n. 10, 2016. ISSN 20477481. Disponível em: <https://doi.org/10.1002/smr.1801>.

WANG, Y. et al. Bug Patterns Localization Based on Topic Model for Bugs in Program Loop. In: **18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018**. Lisbon, Portugal: IEEE, 2018. ISBN 9781538678398. Disponível em: <https://doi.org/10.1109/QRS-C.2018.00070>.

______. DrDebug: Deterministic Replay based Cyclic Debugging with Dynamic Slicing Categories and Subject Descriptors. In: **IEEE/ACM International Symposium on Code Generation and Optimization**. Orlando, FL, USA: ACM, 2014. (CGO'14), p. 98:98–98:108. ISBN 9781450326704. Disponível em: <http://doi.acm.org/10.1145/2544137.2544152>.

WEN, M.; WU, R.; CHEUNG, S.-C. Locus: Locating Bugs from Software Changes. In: **31st IEEE/ACM International Conference on Automated Software Engineering**. Singapore: ACM, 2016. (ASE 2016), p. 262–273. ISBN 978-1-4503-3845-5. ISSN 1090-3801. Disponível em: <https://doi.org/10.1145/2970276.2970359>.

WONG, C.-P. et al. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In: **International Conference on**

Software Maintenance and Evolution (ICSME). Victoria, BC, Canada: IEEE, 2014. p. 181–190. ISBN 978-1-4799-6146-7. ISSN 1063-6773. Disponível em: <https://doi.org/10.1109/ICSME.2014.40>.

WONG, W. E. et al. The DStar Method for Effective Software Fault Localization. **IEEE Transactions on Reliability**, v. 63, n. 1, p. 290–308, 2014. ISSN 0018-9529. Disponível em: <https://doi.org/10.1109/TR.2013.2285319>.

_____. A Survey on Software Fault Localization. **IEEE Transactions on Software Engineering**, PP, n. 99, p. 1, 2016. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2016.2521368>.

WU, Q. et al. Adapting boosting for information retrieval measures. **Information Retrieval**, v. 13, n. 3, p. 254–270, 2010. ISSN 13864564.

XIAO, Y. et al. Improving bug localization with word embedding and enhanced convolutional neural networks. **Information and Software Technology**, Elsevier, v. 105, n. July 2018, p. 17–29, 2019. ISSN 09505849. Disponível em: <https://doi.org/10.1016/j.infsof.2018.08.002>.

_____. Improving Bug Localization with an Enhanced Convolutional Neural Network. In: **24th Asia-Pacific Software Engineering Conference (APSEC)**. Nanjing, China: IEEE, 2017. p. 338–347. ISBN 9781538636817. ISSN 15301362. Disponível em: <https://doi.org/10.1109/APSEC.2017.40>.

_____. Bug Localization with Semantic and Structural Features using Convolutional Neural Network and Cascade Forest. In: **22nd International Conference on Evaluation and Assessment in Software Engineering 2018**. Christchurch, New Zealand: ACM, 2018. (EASE'18), p. 101–111. ISBN 9781450364034. Disponível em: <https://doi.org/10.1145/3210459.3210469>.

XU, J.; LI, H. AdaRank: A Boosting Algorithm for Information Retrieval. In: **30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '07**. Amsterdam, The Netherlands: ACM, 2007. p. 391. ISBN 9781595935977. ISSN 1595935975 (ISBN); 9781595935977 (ISBN). Disponível em: <https://doi.org/10.1145/1277741.1277809>.

XUAN, J.; MONPERRUS, M. Learning to combine multiple ranking metrics for fault localization. In: **30th International Conference on Software Maintenance and Evolution (ICSME 2014)**. Victoria, BC, Canada: [s.n.], 2014. p. 191–200. ISBN 9780769553030. ISSN 1063-6773. Disponível em: <https://doi.org/10.1109/ICSME.2014.41>.

YANG, G.; MIN, K.; LEE, B. Applying Deep Learning Algorithm to Automatic Bug Localization and Repair. In: **35th Annual ACM Symposium on Applied Computing**. Brno Czech Republic: ACM, 2020. p. 1634–1641. ISBN 9781450368667. Disponível em: <https://doi.org/10.1145/3341105.3374005>.

YANG, Z. et al. IncBL: Incremental Bug Localization. In: **36th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. Melbourne, Australia: IEEE, 2021. p. 1223–1226. Disponível em: <https://doi.org/10.1109/ASE51524.2021.9678546>.

YE, X.; BUNESCU, R.; LIU, C. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In: **22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. Hong Kong, China: ACM, 2014. (FSE 2014, April), p. 689–699. ISBN 978-1-4503-3056-5. ISSN 9781450330565. Disponível em: <http://doi.acm.org/10.1145/2635868.2635874>.

______. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. **IEEE Transactions on Software Engineering**, v. 42, n. 4, p. 379 – 402, 2016. ISSN 00985589. Disponível em: <https://doi.org/10.1109/TSE.2015.2479232>.

YE, X. et al. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In: **38th International Conference on Software Engineering (ICSE)**. Austin, TX, USA: ACM, 2016. (ICSE '16), p. 404–415. ISBN 978-1-4503-3900-1. Disponível em: <https://doi.org/10.1145/2884781.2884862>.

YOUM, K. C. et al. Bug Localization Based on Code Change Histories and Bug Reports. **2015 Asia-Pacific Software Engineering Conference (APSEC)**, p. 190–197, 2015. Disponível em: <https://doi.org/10.1109/APSEC.2015.23>.

YUAN, W. et al. DependLoc: A Dependency-based Framework For Bug Localization. In: **27th Asia-Pacific Software Engineering Conference (APSEC)**. Singapore, Singapore: IEEE, 2020. p. 61–70. Disponível em: <https://doi.org/10.1109/APSEC51365.2020.00014>.

ZHANG, J. et al. Exploiting Code Knowledge Graph for Bug Localization via Bi-Directional Attention. In: **28th International Conference on Program Comprehension**. Seoul, Republic of Korea: ACM, 2020. p. 219–229. ISBN 9781450379588. Disponível em: <https://doi.org/10.1145/3387904.3389281>.

ZHANG, M. et al. Boosting spectrum-based fault localization using PageRank. In: **26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017**. Santa Barbara, CA, USA: ACM, 2017. p. 261–272. ISBN 9781450350761. Disponível em: <https://doi.org/10.1145/3092703.3092731>.

ZHANG, T. et al. A Commit Messages-Based Bug Localization for Android Applications. **International Journal of Software Engineering and Knowledge Engineering**, v. 29, n. 4, p. 457–487, 2019. ISSN 02181940. Disponível em: <https://doi.org/10.1142/S0218194019500207>.

ZHANG, Z. et al. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In: **2019 IEEE 26th International Conference**

on Software Analysis, Evolution and Reengineering (SANER). Hangzhou, China: IEEE, 2019. p. 445–455. Disponível em: <https://doi.org/10.1109/SANER.2019.8668002>.

ZHAO, F. et al. Is learning-to-rank cost-effective in recommending relevant files for bug localization? In: **International Conference on Software Quality, Reliability and Security (QRS)**. Vancouver, BC, Canada: IEEE, 2015. p. 298–303. ISBN 9781467379892. Disponível em: <https://doi.org/10.1109/QRS.2015.49>.

ZHENG, W. et al. Fault Localization Analysis Based on Deep Neural Network. **Mathematical Problems in Engineering**, v. 2016, p. 11, 2016. Disponível em: <https://doi.org/10.1155/2016/1820454>.

ZHOU, J. et al. Where Should the Bugs Be Fixed? More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In: **34th International Conference on Software Engineering**. Zurich, Switzerland: IEEE, 2012. (ICSE '12, 11), p. 14–24. ISBN 9781467310673. ISSN 02705257. Disponível em: <https://doi.org/10.1109/ICSE.2012.6227210>.