UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

CHARLES MENDES DE MACEDO

**Aplicação de algoritmos de agrupamento para descoberta de padrões de
defeito em software JavaScript**

São Paulo

2019

CHARLES MENDES DE MACEDO

Aplicação de algoritmos de agrupamento para descoberta de padrões de defeito em software JavaScript

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em 26 de outubro de 2018. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Orientador: Profa. Dra. Karina Valdivia Delgado

São Paulo

2019

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

CATALOGAÇÃO-NA-PUBLICAÇÃO

(Universidade de São Paulo. Escola de Artes, Ciências e Humanidades. Biblioteca)
CRB-8 4936

Macedo, Charles Mendes de

Aplicação de algoritmos de agrupamento para descoberta de padrões de defeito em software JavaScript / Charles Mendes de Macedo ; orientadora, Karina Valdivia Delgado. – 2019.

82 f. : il.

Dissertação (Mestrado em Ciências) - Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, em 2018.

Versão corrigida

1. Engenharia de software. 2. Aprendizado computacional. 3. Reconhecimento de padrões. 4. Qualidade de software. 5. Mineração de dados I. Delgado, Karina Valdivia, orient. II. Título.

CDD 22.ed.– 005.1

Dissertação de autoria de Charles Mendes de Macedo, sob o título **Aplicação de algoritmos de agrupamento para descoberta de padrões de defeito em software JavaScript**, apresentado à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovado em 26 de outubro de 2018 pela comissão examinadora constituída pelos doutores:

Profª. Dra. Karina Valdivia Delgado

Instituição: Universidade de São Paulo - EACH

Presidente

Prof. Dr. Marcelo de Souza Lauretto

Instituição: Universidade de São Paulo - EACH

Prof. Dr. Marcelo Medeiros Eler

Instituição: Universidade de São Paulo - EACH

Profª. Dra. Juliana Cristina Braga

Instituição: Universidade Federal do ABC - CMCC

À Edeneir; minha mãe, por ser meu exemplo de luta e determinação, pelo amor e apoio.

À Marciêne e Marcia; minhas irmãs, pelo apoio.

À Bruna; minha esposa, pelo amor, compreensão e apoio incondicional.

Agradecimentos

A realização deste mestrado contou com importantes apoios e incentivos, sem os quais não teria se tornado uma realidade e aos quais estarei eternamente grato.

À professora Karina Valdivia Delgado, pelos ensinamentos e orientações, por acreditar em mim, por me ouvir com respeito e pela sua dedicação. Sempre preocupada com meu desenvolvimento acadêmico e cobrando o melhor de mim.

À professora Elisamara de Oliveira, pelo incentivo a realização do mestrado, por acreditar em mim já na graduação com o projeto de Iniciação Científica e pelo incentivo ao estudo e pesquisa. Ao professor Igor Wiese, pelo apoio, pelo compartilhamento de experiências e sugestões.

Aos amigos e colegas Andrei Martins, Diego Neves, Fernando Costa e Jonas Targino do Programa de Pós-graduação em Sistemas de Informação (PPgSI) da EACH pelo companheirismo, compartilhando experiências e aprendizados.

À Edeneir Rosa e minhas irmãs, pelo apoio e compreensão. À Bruna Coimbra, por estar sempre ao meu lado, pelo amor e compreensão.

Aos participantes da banca de qualificação e dissertação pelas observações e direcionamentos que me auxiliaram a guiar esse trabalho e pela participação da banca examinadora de defesa.

E, finalmente, agradeço à Deus pela minha vida, pela paz nos momentos em que me encontrei incapaz de prosseguir e pela força que me fez superar obstáculos e dificuldades e concluir com êxito este mestrado.

“Por vezes sentimos que aquilo que fazemos não é senão uma gota de água no mar. Mas o mar seria menor se lhe faltasse uma gota...”
(Madre Teresa de Calcuta)

“Não deixe o barulho da opinião dos outros abafar sua voz interior. E mais importante, tenha a coragem de seguir seu coração e sua intuição. Eles de alguma forma já sabem o que você realmente quer se tornar. Tudo o mais é secundário.”
(Steve Jobs)

Resumo

Macedo, Charles Mendes de. **Aplicação de algoritmos de agrupamento para descoberta de padrões de defeito em software JavaScript**. 2019. 82 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2019.

As aplicações desenvolvidas com a linguagem JavaScript, vêm aumentando a cada dia, não somente aquelas na web (*client-side*), como também as aplicações executadas no servidor (*server-side*) e em dispositivos móveis (*mobile*). Neste contexto, a existência de ferramentas para identificação de defeitos e *code smells* é fundamental, para auxiliar desenvolvedores durante a evolução destas aplicações. A maioria dessas ferramentas utiliza uma lista de defeitos predefinidos que são descobertos a partir da observação das melhores práticas de programação e a intuição do desenvolvedor. Para melhorar essas ferramentas, a descoberta automática de defeitos e *code smells* é importante, pois permite identificar quais ocorrem realmente na prática e de forma frequente. Uma ferramenta que implementa uma estratégia semiautomática para descobrir padrões de defeitos através de agrupamentos das mudanças realizadas no decorrer do desenvolvimento do projeto é a ferramenta BugAID. O objetivo deste trabalho é contribuir nessa ferramenta estendendo-a com melhorias na abordagem da extração de características, as quais são usadas pelos algoritmos de clusterização. O módulo estendido encarregado da extração de características é chamado de BugAIDExtract+ (BE+). Além disso, neste trabalho é realizada uma avaliação de vários algoritmos de clusterização na descoberta dos padrões de defeitos em software JavaScript.

Palavras-chave: Engenharia de Software. Aprendizado computacional. Descoberta de defeitos. JavaScript. Reconhecimento de padrões. Qualidade de software. Mineração de dados.

Abstract

Macedo, Charles Mendes de. **Application of clustering algorithms for discovering bug patterns in JavaScript software**. 2019. 82 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2019.

Applications developed with JavaScript language are increasing every day, not only for client-side, but also for server-side and for mobile devices. In this context, the existence of tools to identify faults is fundamental in order to assist developers during the evolution of their applications. Most of these tools use a list of predefined faults that are discovered from the observation of the programming best practices and developer intuition. To improve these tools, the automatic discovery of faults and code smells is important because it allows to identify which ones actually occur in practice and frequently. A tool that implements a semiautomatic strategy for discovering bug patterns by grouping the changes made during the project development is the BugAID. The objective of this work is to contribute to the BugAID tool, extending this tool with improvements in the extraction of characteristics to be used by the clustering algorithm. The extended module that extracts the characteristics is called BE+. Additionally, an evaluation of the clustering algorithms used for discovering fault patterns in JavaScript software is performed.

Keywords: Software Engineering. Machine Learning. Discovery of bug. JavaScript. Pattern Recognition. Software quality. Data Mining.

Lista de figuras

Figura 1 – Exemplo do funcionamento do algoritmo K-Means. (a) Todos os dados sem <i>clusters</i> definidos; (b) iteração 1, em que os centróides (em destaque) são iniciados aleatoriamente, de forma que foram geradas 3 partições; (c) iteração 2, os centróides se movimentando ao centro dos grupos correspondentes; (d) iteração 3; e (e) resultado final.	23
Figura 2 – Exemplo hipotético para ilustra os termos utilizados pelo algoritmo DBScan.	24
Figura 3 – Exemplo de um conjunto de dados com densidade variável e agrupamentos encontrados pelo algoritmo Optics.	25
Figura 4 – Exemplo do funcionamento do algoritmo Optics em um conjunto de dados com densidade variável.	25
Figura 5 – Dendrograma ilustrando o resultado do HDBScan com $MinPts = 25$ para um conjunto de dados com 8.000 objetos. O algoritmo encontra 6 <i>clusters</i> (em destaque em vermelho). No eixo X estão os agrupamentos e no eixo Y os valores de <i>epsilon</i>	26
Figura 6 – Exemplo de commit e BCTs encontrados pelo módulo de extração de características BE: (i) mensagem de confirmação do commit; (ii) mudanças realizadas no <i>commit</i> em destaque em vermelho e verde; (iii) BCTs encontrados no <i>commit</i> ; (iv) vetor de características correspondente a esse <i>commit</i>	53
Figura 7 – Exemplo de commit e BCTs encontrados pelo módulo de extração de características BE: (i) mensagem de confirmação do commit; (ii) mudanças realizadas no <i>commit</i> em destaque em vermelho e verde; (iii) BCTs encontrados no <i>commit</i> ; (iv) vetor de características correspondente a esse <i>commit</i>	54
Figura 8 – Os três primeiros <i>clusters</i> obtidos pela ferramenta BugAID.	60
Figura 9 – Indicação do código com defeito através do recurso <i>light bulb</i> e com a sugestão da possível correção apresentada pela ferramenta JS Advice. .	65

Figura 10 – <i>Commit</i> com a inclusão do valor padrão para a variável <i>pkg.author</i> (linha 188). Note que antes era verificado se não tinha valor referenciado (linha 190), porém essa linha foi removida.	68
Figura 11 – <i>Commit</i> criando o método <i>isDivisibleBy</i> a partir da prototipação do objeto <i>Validator</i>	68
Figura 12 – <i>Commit</i> alterando o código para verificar o tipo de instância com o comando <i>instanceof</i> antes de adicionar um novo item no vetor <i>array</i> . .	69

Lista de algoritmos

Algoritmo 1 – Extração BCTs	50
---------------------------------------	----

Lista de tabelas

Tabela 1 – Questões de Pesquisa	35
Tabela 2 – Strings de busca	36
Tabela 3 – Estudos selecionados na RSL	38
Tabela 4 – Catálogo de defeitos.	39
Tabela 5 – Catálogo de <i>code smells</i>	40
Tabela 6 – Características do método proposto dos estudos selecionados na RSL .	45
Tabela 7 – Características dos experimentos realizados nos estudos selecionados na RSL	45
Tabela 8 – Conjunto de dados de exemplo com três <i>commits</i> . Cada <i>commit</i> tem o nome do projeto ao qual ele pertence, o identificador do <i>commit</i> , o número de instruções modificadas no <i>commit</i> e uma lista dos BCTs e o número de vezes que cada um deles acontece no <i>commit</i>	51
Tabela 9 – Resultados da extração de características utilizando os módulos BE e BE+	66
Tabela 10 – Resultados da extração de características da amostra A-25 utilizando os módulos BE e BE+	69
Tabela 11 – <i>Cluster</i> e quantidade de <i>commits</i> por <i>cluster</i> considerando a amostra A-25	70
Tabela 12 – Índice Rand: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy	71
Tabela 13 – Índice Jaccard: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy	71
Tabela 14 – Índice Rand Ajustado: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy	72
Tabela 15 – Os 5 melhores resultados obtidos pelo algoritmo DBScan	72
Tabela 16 – Os 5 melhores resultados obtidos pelo algoritmo Optics	73
Tabela 17 – Os 5 melhores resultados obtidos do algoritmo HDBScan	73
Tabela 18 – Resumo com os melhores resultados obtidos pelos algoritmos avaliados na amostra A-25	74

Sumário

1	Introdução	16
1.1	<i>Apresentação do problema</i>	17
1.2	<i>Objetivo</i>	18
1.3	<i>Método</i>	18
1.4	<i>Organização</i>	19
2	Clusterização de dados	20
2.1	<i>A tarefa de clusterização</i>	20
2.2	<i>Algoritmos de clusterização</i>	21
2.2.1	K-Means	22
2.2.2	C-Means Fuzzy	22
2.2.3	DBScan	23
2.2.4	Optics	24
2.2.5	HDBScan	26
2.3	<i>Pré-processamento dos dados</i>	27
2.4	<i>Validação de agrupamentos</i>	27
2.4.1	Índice Rand	28
2.4.2	Índice Rand Ajustado	29
2.4.3	Índice Jaccard	29
3	Conceitos básicos sobre defeitos e JavaScript	30
3.1	<i>Linguagem JavaScript e bibliotecas</i>	30
3.2	<i>Defeito, erro e falha</i>	31
3.3	<i>Tipos de defeitos e code smells</i>	32
3.4	<i>Análise estática e dinâmica</i>	33
4	Detecção automática de defeitos em software JavaScript: Uma revisão sistemática da literatura	34
4.1	<i>Método da revisão sistemática</i>	34
4.1.1	Planejamento	34
4.1.2	Condução	36

4.2	<i>Resultados da revisão sistemática</i>	37
4.2.1	Catálogo de defeitos e <i>code smells</i> em JavaScript	37
4.2.2	Métodos utilizados para detecção: RQ1	37
4.2.3	Características do método proposto: RQ2	44
4.2.4	Características dos experimentos: RQ3	46
4.3	<i>Conclusões da revisão sistemática</i>	46
5	Estratégia para descoberta de padrões de defeitos de Hanam et.al	48
5.1	<i>Etapas da abordagem de descoberta de padrões</i>	48
5.1.1	Mineração dos dados	48
5.1.2	Extração de mudanças básicas	49
5.1.3	Clusterização	52
5.2	<i>Resultados</i>	55
5.3	<i>A ferramenta BugAID</i>	58
6	Identificação de deficiências no BugAID e proposta de melhorias 61	
6.1	<i>Identificação das deficiências do módulo BE da ferramenta BugAID</i> .	61
6.2	<i>Abordagem proposta BE+</i>	63
6.3	<i>Ferramenta JS Advice</i>	64
6.4	<i>Deficiências na avaliação do algoritmo de agrupamento utilizado na ferramenta BugAID</i>	65
7	Resultados	66
7.1	<i>Avaliação da extração de características do BE+</i>	66
7.2	<i>Validação dos algoritmos de agrupamento</i>	69
7.2.1	Resultados dos algoritmos K-Means e C-Means Fuzzy	71
7.2.2	Resultado do algoritmo DBScan	71
7.2.3	Resultado do algoritmo Optics	72
7.2.4	Resultado do algoritmo HDBScan	73
7.2.5	Resumo da validação dos algoritmos de agrupamento	73
8	Conclusão e trabalhos futuros	75
8.1	<i>Contribuições</i>	75

8.2	<i>Limitações</i>	76
8.3	<i>Trabalhos futuros</i>	77
	Referências ¹	78

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

O software é um elemento essencial que aparece em vários setores da sociedade tais como educação, medicina, ciência, transporte e telecomunicações, sendo que o desenvolvimento de software de qualidade e de baixo custo é um dos principais objetivos da Engenharia de Software. Levar em consideração a qualidade no desenvolvimento de software traz muitos benefícios, tanto para a empresa quanto para o cliente final. A empresa tem: (i) maior produtividade; (ii) redução de custo de desenvolvimento e manutenção; (iii) aumento de confiabilidade do produto; (iv) maior índice de satisfação do cliente. O cliente tem um produto confiável, seguro e de boa qualidade, pois dependendo da área isso pode significar salvar vidas ou evitar desastres.

Uma das características de um software de qualidade é ter uma baixa taxa de defeitos. O foco desta dissertação de mestrado é a descoberta de defeitos em software escrito na linguagem JavaScript, pois essa linguagem é considerada atualmente uma das mais populares usadas por desenvolvedores (HANAM; BRITO; MESBAH, 2016). JavaScript, além de ser uma linguagem interpretada por todos os navegadores web com aplicações no lado do cliente (*client-side*), atualmente é executada no servidor (*server-side*)¹ e em dispositivos móveis (*mobile*)².

JavaScript é uma linguagem dinâmica, fracamente tipada e assíncrona. Essa linguagem suporta recursos intrincados como o encadeamento de funções, é prototípica, aceita funções de primeira classe e funções que possuem variáveis independentes dentro de funções superiores (*closures*). O código fonte pode ser desenvolvido no paradigma estrutural, funcional, orientado a eventos, orientado a objetos e outros. Também é importante destacar que a maioria dos projetos possuem um grande número de dependências com bibliotecas de terceiros (*frameworks* e *plug-ins*) e *APIs* externas (OCARIZA et al., 2013).

Além destas particularidades, surgem preocupações distintas que os desenvolvedores, arquitetos e engenheiros de software têm que levar em consideração sobre o ambiente em que será desenvolvida a aplicação ou o software. Por exemplo, em aplicações executadas no lado do cliente a versão do navegador pode interferir na execução do código JavaScript e dependendo da versão pode apresentar falta de compatibilidade para interpretar alguns

¹ É executada com o ambiente de tempo de execução do JavaScript, chamado Node.js (CANTELON et al., 2014)

² É executada utilizando bibliotecas como o PhoneGap (HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2012)

códigos (PARK et al., 2015). Por esse motivo, a tarefa de descobrir defeitos em software nos diversos ambientes e situações torna-se um grande desafio para linguagens multiplataforma como JavaScript.

1.1 Apresentação do problema

Diferentes ferramentas e abordagens para detecção de defeitos para JavaScript têm sido propostas ao longo dos anos (OCARIZA, 2016; BAE et al., 2014; FARD; MESBAH, 2013; JR; PATTABIRAMAN; MESBAH, 2012; LIANG et al., 2014). A maioria dessas ferramentas utilizam uma lista fixa de defeitos predefinidos que são descobertos a partir da leitura das melhores práticas de programação, da intuição do desenvolvedor e da observação (HANAM; BRITO; MESBAH, 2016), isto é, não são implementadas estratégias que realizem de forma sistemática a descoberta desses defeitos. Esse tipo de estratégia é importante para identificar quais defeitos e *code smells* (código que pode indicar algum problema em potencial) realmente ocorrem na prática, por exemplo, no caso de programas desenvolvidos em projetos grandes. O resultado dessas estratégias podem ser úteis para melhorar ferramentas de detecção de defeitos ou *code smells* em JavaScript, além de conscientizar os desenvolvedores sobre os defeitos e *code smells* que são comuns quando são desenvolvidos programas em Javascript (HANAM; BRITO; MESBAH, 2016).

Uma ferramenta que implementa uma técnica semi-automática para descobrir padrões de defeitos, é a ferramenta BugAID (HANAM; BRITO; MESBAH, 2016). Essa ferramenta identifica as mudanças realizadas nos projetos que podem corresponder a defeitos. O BugAID primeiro obtém as características das mudanças realizadas no repositório público de código fonte GitHub e depois aplica o algoritmo de aprendizado de máquina não supervisionado DBScan (ESTER et al., 1996) para agrupar os padrões de defeitos mais similares. Porém, essa abordagem apresentou deficiências na detecção e descoberta de defeitos associados às *APIs* externas (OCARIZA et al., 2013), manipulação de arquivos, manipulação de textos, funções de cálculo matemático, variáveis de contexto local, entre outras.

1.2 Objetivo

Esta dissertação de mestrado tem por objetivo descobrir padrões de defeito em software JavaScript resolvendo as deficiências encontradas na ferramenta BugAID (HANAM; BRITO; MESBAH, 2016) na detecção e descoberta de defeitos associados às *APIs* externas (OCARIZA et al., 2013), manipulação de arquivos, manipulação de textos, funções de cálculo matemático, variáveis de contexto local, entre outras. Com isso, espera-se detectar defeitos e *code smells* que são comuns no desenvolvimento de programa em JavaScript e não foram descobertos no estudo Hanam, Brito e Mesbah (2016).

1.3 Método

Este projeto de pesquisa teve como foco descobrir padrões de defeito em software JavaScript utilizando algoritmos de aprendizado de máquina não supervisionado. Para tal, primeiro foram pesquisados trabalhos relacionados à detecção automática de defeitos em software JavaScript e foi realizada uma revisão sistemática da literatura. Essa revisão sistemática permitiu identificar os métodos usados para realizar a detecção automática de defeitos, as características destes métodos e as características dos experimentos realizados.

Foi escolhida a ferramenta BugAID (HANAM; BRITO; MESBAH, 2016) que faz a mineração de código no repositório online GitHub e utiliza aprendizado de máquina não supervisionado para encontrar os padrões de defeitos. Uma vez que o objetivo do BugAID é descobrir os padrões de defeito mais usados, não existe uma rotulação dos defeitos para serem utilizados algoritmos de aprendizado de máquina supervisionado.

Neste trabalho de mestrado foram identificadas as deficiências na ferramenta BugAID para melhorá-la e estendê-la. Assim, foi proposto um novo módulo de extração de características para o BugAID chamado de BE+. Com o objetivo de avaliar e comparar os resultados foram selecionados seis algoritmos de clusterização (agrupamento) e as algumas métricas de validação. Esses algoritmos e métricas foram aplicadas em uma amostra com 25 *commits* selecionados aleatoriamente.

1.4 Organização

O restante deste texto está organizado da seguinte forma. No Capítulo 2 são apresentados os conceitos de clusterização de dados, os algoritmos de agrupamento e as estratégias de validação dos resultados da aplicação desses algoritmos. No Capítulo 3 são informados os conceitos e terminologias que serão utilizados nesta dissertação de mestrado e algumas informações sobre a linguagem JavaScript e suas bibliotecas. O Capítulo 4 contém a revisão sistemática realizada pelo autor sobre detecção automática de defeitos em software JavaScript (MACEDO; DELGADO; WIESE, 2017), em que são descritos os métodos, os resultados e a conclusão dessa revisão. Já no Capítulo 5 é apresentado o trabalho base utilizado nesta dissertação, que é o trabalho de Hanam, Brito e Mesbah (2016). Em seguida, no Capítulo 6 são apresentadas as principais contribuições deste trabalho de mestrado: o módulo estendido encarregado da extração de característica chamado de BE+ e a ferramenta JS Advice. Por último, no Capítulo 7 são apresentados os resultados da clusterização comparando os módulo BE da ferramenta BugAID e módulo BE+ proposto. Por fim, o Capítulo 8 apresenta as considerações finais.

2 Clusterização de dados

Neste capítulo são apresentados a tarefa de clusterização de dados, os algoritmos de clusterização que serão usados nos experimentos, a tarefa de pré-processamento dos dados e as estratégias de validação que serão utilizadas para avaliar a abordagem proposta.

2.1 A tarefa de clusterização

A tarefa de agrupamento de dados ou clusterização de dados tem sido estudada há muito tempo, e a partir do estudo de Hartigan (1975) vem ganhando destaque. Atualmente, a clusterização de dados é aplicada em diversos domínios e áreas de pesquisa, como na Biomedicina (ANDREOPOULOS et al., 2009), na descoberta de medicamentos (XU; WUNSCH, 2005), Psiquiatria (JAIN; DUBES, 1988) e outros.

O objetivo principal dos algoritmos de clusterização é detectar e agrupar os dados com características semelhantes, em outras palavras, gerar grupos de dados com informações parecidas. Desta forma, o resultado dos algoritmos de clusterização de dados são conjuntos de dados agrupados, denominados *clusters*.

A seguir, são apresentados alguns termos usados nesta seção:

- Conjunto de dados: $X = \{x_1, x_2, \dots, x_n\}$ representa o conjunto de dados com n elementos.
- Vector de características: $x_i = (x_{i1}, x_{i2}, \dots, x_{id})$ é o vetor de características que representa o dado x_i com d atributos.
- Matriz de características dos dados: os dados são armazenados em uma matriz de dados $n \times d$.
- Partição do conjunto de dados em k *clusters*: uma partição consiste em uma coleção de subconjuntos $C = \{C_1, C_2, \dots, C_k\}$ tal que $C_1 \cup C_2 \cup \dots \cup C_k = X$, $C_i \neq \emptyset$ e $C_i \cap C_j = \emptyset$.

A seguir, são descritas as classes de algoritmos de clusterização.

2.2 Algoritmos de clusterização

Os algoritmos de clusterização podem ser classificados da seguinte maneira (JAIN; DUBES, 1988):

- Algoritmos de clusterização particional: a abordagem destes algoritmos é formar agrupamentos dos dados, dividindo-os em grupos de k -partições e de forma iterativa, minimizando as distâncias entre os dados similares e maximizando a distância entre os grupos até chegar na melhor distribuição das partições. Os algoritmos K-Means (MACQUEEN et al., 1967) e C-Means Fuzzy (HATHAWAY; BEZDEK, 1988) utilizam essa abordagem.
- Algoritmos de clusterização baseados em grade: os algoritmos que utilizam essa estratégia definem uma estrutura de grade para o espaço de exemplos. O algoritmo CLIQUE (AGRAWAL et al., 1998) utiliza essa abordagem.
- Algoritmos de clusterização baseados em densidade: essa abordagem considera que os *clusters* estão agrupados por regiões com mais dados (densas) e separados por regiões com menos dados (vales ou ruídos). O agrupamento é realizado de forma que um *cluster* é composto por dados que tenham um número mínimo de vizinhos dentro de uma esfera com raio R , o qual é passado como parâmetro para o algoritmo. Os algoritmos DBScan (ESTER et al., 1996) e Optics (ANKERST et al., 1999) seguem essa abordagem.
- Algoritmos de clusterização com sobreposição: essa abordagem pode associar cada dado a vários *clusters*, desta forma cada dado recebe um grau de pertinência aos *clusters* encontrados. O algoritmo AUTOCLASS (CHEESEMAN et al., 1988) utiliza essa abordagem.
- Algoritmos de clusterização de redes auto-organizáveis (SOM - em inglês *Self-Organizing Map*): esses algoritmos utilizam redes neurais artificiais em que os neurônios estão organizados em um reticulado bidimensional. A cada dado apresentado, cada neurônio ajusta o seu próprio peso ativando uma região do reticulado. Dessa forma, encontra-se os neurônios de referência, de modo que os dados sejam associados ao neurônio de referência mais próximo. O algoritmo SomPack (KOHONEN, 1997) segue essa abordagem.

- Algoritmos de clusterização hierárquicos: esses algoritmos constroem uma hierarquia de *clusters*. São gerados *clusters* que contêm *subclusters*, e esses *subclusters* podem gerar outros *subclusters* e assim por diante, até esgotar o nível hierárquico mais granular por similaridade. O algoritmo HDBScan (CAMPELLO; MOULAVI; SANDER, 2013) utiliza esse conceito.

A seguir, são descritos alguns algoritmos de clusterização e que serão usados nos experimentos.

2.2.1 K-Means

O algoritmo K-Means (MACQUEEN et al., 1967) é um algoritmo de clusterização particional, que realiza o agrupamento dos dados, dividindo-os em k -partições. Esse algoritmo recebe como parâmetro de entrada a quantidade de k -partições que ele deve procurar. Desta forma, o algoritmo iniciará aleatoriamente o ponto central (centróide) de cada partição e de forma iterativa, minimizando assim a distância do centróide com os objetos (dados) que compõem a partição, até que todos os objetos tenham uma partição e não ocorram mudanças nos grupos com base nos centróides. Esses passos são ilustrados na Figura 1.

O algoritmo K-Means é sensível a dados que possuem valores atípicos, os chamados ruídos ou *outliers*. Além disso, esse algoritmo não lida bem com conjuntos de dados que tenham formatos arbitrários, não convexos e espiralados (ESTER et al., 1996).

2.2.2 C-Means Fuzzy

O algoritmo C-Means Fuzzy (HATHAWAY; BEZDEK, 1988) ou K-Means Fuzzy tem uma abordagem similar ao algoritmo K-Means. Entretanto, o C-Means Fuzzy leva em consideração que o mesmo objeto (dado) pode ser similar com mais de um grupo, tendo um grau de pertinência maior a um grupo do que aos outros.

Esse algoritmo recebe como parâmetros: (i) o número de *clusters* desejado (parâmetro c), (ii) a distância máxima entre os pontos e o centróide; (iii) um erro *epsilon*, que é um valor pequeno usado pelo critério de parada; e (iv) o grau de pertinência de cada objeto para com o seu centróide.

Figura 1 – Exemplo do funcionamento do algoritmo K-Means. (a) Todos os dados sem *clusters* definidos; (b) iteração 1, em que os centróides (em destaque) são iniciados aleatoriamente, de forma que foram geradas 3 partições; (c) iteração 2, os centróides se movimentando ao centro dos grupos correspondentes; (d) iteração 3; e (e) resultado final.

Fonte: JAIN, 2010

2.2.3 DBScan

O algoritmo DBScan (ESTER et al., 1996) é um algoritmo baseado em densidade. Diferentemente, dos algoritmos K-Means e C-Means Fuzzy, o DBScan não recebe a priori a quantidade de *clusters* que deve ser encontrada. Ao invés disso, o algoritmo precisa saber o número mínimo de pontos que um *clusters* pode ter (parâmetro *MinPts*) e o valor da distância máxima do raio (*epsilon*) entre um ponto (objeto) e outro para serem considerados pontos do mesmo *cluster*. O DBScan trabalha muito bem com *outliers*, de forma que estes são identificados pelo algoritmo e informados no resultado.

O algoritmo, inicia por qualquer ponto p do conjunto de dados, analisando todos os pontos, segundo os seguintes critérios (ESTER et al., 1996):

1. Se os pontos vizinhos estão dentro do raio (*epsilon*) e se a quantidade de vizinhos iguala ou supera o valor mínimo de pontos (*MinPts*) para que o grupo seja considerado um *cluster*; o ponto p é considerado o ponto central (ponto vermelho A da Figura 2) e esse grupo é definido como ϵ -vizinhança do ponto p (pontos em vermelho da Figura 2);
2. Se a ϵ -vizinhança de um ponto p tiver a quantidade de pontos menor do que *MinPts*, mas ainda assim tiver algum ponto central nessa vizinhança, o ponto p é chamado de ponto de borda (pontos amarelos B e C da Figura 2);
3. Se ϵ -vizinhanças tiverem pontos centrais respeitando a quantidade mínima de pontos (*MinPts*) e cada ponto central estiver no raio (*epsilon*) máximo, então essas ϵ -vizinhanças são considerados um *cluster* (pontos vermelhos e amarelos da Figura 2);
4. Se o ponto p não possui a quantidade mínima de vizinhos (*MinPts*) e não está em uma ϵ -vizinhança com um ponto central, então esse ponto p é considerado *outlier* (ponto azul N da Figura 2).

Desta forma, o algoritmo retorna todos os *clusters* encontrados e os *outliers* detectados.

Figura 2 – Exemplo hipotético para ilustra os termos utilizados pelo algoritmo DBScan.

Fonte: SCHUBERT et al., 2017

2.2.4 Optics

O algoritmo Optics (ANKERST et al., 1999), assim como o DBScan, é um algoritmo de clusterização baseado em densidade. A forma de realizar o agrupamento do Optics é

bem similar ao do DBScan. Contudo, o Optics consegue detectar *clusters* significativos em conjunto de dados com densidade variável. A Figura 3 mostra um exemplo em que não é possível identificar os cinco *clusters* (A, B, C1, C2 e C3) que existem nesse conjunto de dados, utilizando um único parâmetro de densidade global, como aquele que é utilizado pelo DBScan.

Figura 3 – Exemplo de um conjunto de dados com densidade variável e agrupamentos encontrados pelo algoritmo Optics.

Fonte: ANKERST et al., 1999

Para resolver esse problema, o algoritmo Optics ordena os objetos do conjunto com base na densidade dos dados, de forma que os pontos mais próximos por densidade se tornem vizinhos nesta ordenação.

Figura 4 – Exemplo do funcionamento do algoritmo Optics em um conjunto de dados com densidade variável.

Fonte: SHAH; BHENSADIA; GANATRA, 2012

A Figura 4 ilustra a forma de clusterização do Optics, gerando os agrupamentos com base nos vales gerados pela ordenação por densidade. O algoritmo Optics, têm os mesmos parâmetros do DBScan.

2.2.5 HDBScan

O HDBScan (CAMPELLO; MOULAVI; SANDER, 2013) utiliza o conceito do DBScan para detecção dos *clusters*, mas de forma hierárquica. Segundo Campello, Moulavi e Sander (2013), o HDBScan generaliza as melhores técnicas de agrupamento baseado em densidade existentes, de forma que esse algoritmo só recebe como parâmetro de entrada o valor da quantidade mínima de vizinhos (*MinPts*) e assim, o algoritmo busca uma hierarquia completa de todos os possíveis agrupamentos baseados em densidade.

O algoritmo produz uma árvore de agrupamento que contém todos os agrupamentos obtidos através do DBSCAN de uma maneira hierárquica e aninhada, incluindo nós que indicam quando um objeto que estava isolado se torna *outlier*. Com o conjunto de todos os agrupamentos obtidos pelo DBSCAN é gerada uma lista com o valor *MinPts* e todos os valores possíveis do raio *epsilon*. Assim, é gerada uma estrutura de dendrograma com os agrupamentos obtidos.

A Figura 5 ilustra a estrutura de dendrograma que corresponde ao resultado do HDBScan para um exemplo hipotético com o parâmetro *MinPts*=25.

Figura 5 – Dendrograma ilustrando o resultado do HDBScan com *MinPts* = 25 para um conjunto de dados com 8.000 objetos. O algoritmo encontra 6 *clusters* (em destaque em vermelho). No eixo X estão os agrupamentos e no eixo Y os valores de *epsilon*.

Fonte: CAMPELLO; MOULAVI; SANDER, 2013

2.3 Pré-processamento dos dados

A maioria dos dados extraídos das fontes originais não estão formatados adequadamente para que os algoritmos de clusterização consigam ser executados. Desta forma, é necessário realizar o processo de preparação e adequação dos dados para que os algoritmos trabalhem corretamente. Esse procedimento é chamado de pré-processamento de dados.

Nessa etapa de pré-processamento, os atributos dos dados podem ser transformados em tipos diferentes, como: (i) binário (0 ou 1); (ii) valores discretos ; e (iii) valores contínuos. Assim, pode ser necessário normalizar os dados em escalas mais similares ou até remover atributos que não estejam no domínio da solução do problema (JAIN; DUBES, 1988).

2.4 Validação de agrupamentos

Os algoritmos de aprendizado de máquina supervisionado possuem rótulos indicando os resultados que deveriam ser esperados pela classificação do algoritmo. Já no paradigma de aprendizado não supervisionado não temos esses rótulos e é necessário o uso de heurísticas e suposições para avaliar a qualidade dos *clusters* encontrados pelos algoritmos. Assim, para validar os agrupamentos, obtidos pelos algoritmos de clusterização, é necessário aplicar medidas de validação.

A validação é feita com base em índices que medem de maneira quantitativa a similaridade da estrutura encontrada com os *clusters* que deveriam ter sido detectados. A maneira pela qual um índice é aplicado é chamado de critério de validação. Os principais critérios de validação de agrupamentos são (JAIN; DUBES, 1988; HALKIDI; BATISTAKIS; VAZIRGIANNIS, 2001):

- Critérios de validação relativa: a ideia fundamental desta validação é como escolher a melhor solução de agrupamento que melhor representa os dados, de acordo com algum critério pré-especificado. Esses critérios normalmente são utilizados para identificar o melhor valor para um parâmetro utilizado pelo algoritmo ou comparar resultados obtidos entre algoritmos.
- Critérios de validação interna: esses critérios utilizam os dados originais ou matriz de similaridade para medir a qualidade do agrupamento. Em outras palavras, são

utilizados testes de hipótese para determinar se um agrupamento obtido é apropriado, se comparado com os dados originais ou com a matriz de similaridade.

- Critérios de validação externa: para medir a qualidade dos *clusters* é utilizada adicionalmente uma partição já conhecida do conjunto de dados, por exemplo uma partição dada por um especialista. Em outras palavras, a avaliação do grau de concordância ocorre entre duas soluções de agrupamento de dados: (i) partição obtida pelo algoritmo de clusterização, denotada por $C^{ob} = \{C_1^{ob}, \dots, C_k^{ob}\}$; e (ii) partição de referência, i.e. uma partição já conhecida, denotada por $C^{ref} = \{C_1^{ref}, \dots, C_q^{ref}\}$. A validação externa ocorre em muitas ocasiões por contagem de pares de objetos entre as duas partições. Assim, são definidas as seguintes variáveis que são usadas nos índices de validação externa:

- a_{11} quantidade de pares em um mesmo grupo em C^{ob} e C^{ref} .
- a_{10} quantidade de pares em um mesmo grupo em C^{ob} , mas em diferentes grupos em C^{ref} .
- a_{01} quantidade de pares em diferentes grupos em C^{ob} , mas em um mesmo grupo em C^{ref} .
- a_{00} quantidade de pares em diferentes grupos em C^{ob} e em diferentes grupos em C^{ref} .

A seguir, são apresentados três índices de validação externa: Rand, Rand Ajustado e Jaccard. Esses índices são utilizados como forma de avaliação nos experimentos.

2.4.1 Índice Rand

O índice Rand (RAND, 1971) é um índice de validação externa que retorna valores entre 0 e 1, sendo que os valores mais próximo de 1 indicam maior similaridade entre as duas partições analisadas. O índice Rand calcula a proporção de acertos de pares de objetos entre duas partições (que correspondem à soma de a_{11} e a_{00}) em relação ao total de possíveis pares de objetos, isto é:

$$IR(C^{obj}, C^{ref}) = \frac{a_{11} + a_{00}}{a_{11} + a_{10} + a_{01} + a_{00}}.$$

2.4.2 Índice Rand Ajustado

O Rand Ajustado (HUBERT; ARABIE, 1985) (ou Rand Ajustado para aleatoriedade) é um índice de validação externa que foi criado para suprir a dificuldade do índice Rand afim de identificar a qualidade do valor obtido na comparação entre as duas partições geradas aleatoriamente, pois tais partições podem ter valores inesperados muito altos para o índice Rand.

Esse índice retorna valores entre $-\infty$ e 1, sendo que valores positivos indicam que a similaridade entre as duas partições é maior do que o valor esperado da similaridade de duas partições geradas aleatoriamente. Considerando uma distribuição hipergeométrica, o índice Rand Ajustado é calculado por (HUBERT; ARABIE, 1985):

$$IRA(C^{obj}, C^{ref}) = \frac{a_{11} - \frac{(a_{11}+a_{01})(a_{11}+a_{10})}{a_{11}+a_{10}+a_{01}+a_{00}}}{\frac{(a_{11}+a_{01})+(a_{11}+a_{10})}{2} - \frac{(a_{11}+a_{01})(a_{11}+a_{10})}{a_{11}+a_{10}+a_{01}+a_{00}}}.$$

2.4.3 Índice Jaccard

O índice Jaccard (REAL; VARGAS, 1996) é um índice de validação externa, que calcula a proporção de pares de objetos em um mesmo grupo em C^{obj} e em C^{ref} com relação à quantidade de pares de objetos contidos em um mesmo grupo em C^{obj} ou em C^{ref} , isto é:

$$IJ(C^{obj}, C^{ref}) = \frac{a_{11}}{a_{11} + a_{01} + a_{10}}.$$

3 Conceitos básicos sobre defeitos e JavaScript

Neste capítulo são apresentadas algumas informações sobre a linguagem JavaScript e bibliotecas. Depois são descritos alguns conceitos e termos que são utilizados nos próximos capítulos, tais como: defeito, erro, falha, análise estática e análise dinâmica.

3.1 Linguagem JavaScript e bibliotecas

A linguagem JavaScript foi criada em 1995 por Brendan Eich na Netscape (EICH, 2005) e mesmo ultrapassando os vinte anos de existência, é considerada uma das linguagens mais utilizadas atualmente (HANAM; BRITO; MESBAH, 2016).

A linguagem JavaScript recebe atualizações periódicas pela padronizadora Ecma International com a norma Ecma-262. A versão mais recente é a 9.0 (lançada em junho de 2018), mas a que está mais estável é a 8.0¹ ou também conhecida como EcmaScript 8 (nome dado para a linguagem resultante da norma Ecma-262). Desde, a versão do EcmaScript 6, a linguagem JavaScript tem classes, constantes, variáveis de escopo, interpolação de *strings*, *promise* e outros recursos². A empresa Ecma deixa disponível no GitHub³ o repositório do EcmaScript para a comunidade de desenvolvedores, engenheiros e arquitetos de software contribuírem no crescimento da linguagem, de forma *open-source*.

A variedade de bibliotecas (*frameworks* e *plug-ins*) e *APIs* externas (OCARIZA et al., 2013) que facilitam o desenvolvimento de aplicações em JavaScript aumenta a cada dia e esse é um dos fatores da popularidade desta linguagem. Por exemplo, a biblioteca jQuery foi criada para auxiliar o desenvolvimento das aplicações web. Essa biblioteca veio com o propósito de mitigar os problemas de incompatibilidade que os navegadores possuem para interpretar algumas funcionalidades da linguagem, como a tecnologia Ajax (EERNISSE, 2006) e funções de manipulação de HTML.

JavaScript é uma linguagem multiplataforma, podendo ter softwares executando em sistemas operacionais em servidores (*server-side*) com o ambiente de tempo de execução do JavaScript chamado Node.js. Esse ambiente foi criado para facilitar o desenvolvimento de aplicações assíncronas e escaláveis para servidores. Além disso, Node.js é orientado a

¹ <https://www.ecma-international.org/ecma-262/8.0/>

² <http://es6-features.org/>

³ <https://github.com/tc39/ecma262>

eventos e foi desenvolvido em 2009 por Ryan Dahl sobre o interpretador de JavaScript V8⁴. Desde sua criação, vem ganhando adeptos e evoluindo, tanto que o repositório de pacotes do Node.js chamado **npm**⁵, atualmente ultrapassou os repositórios Maven Central⁶ e RubyGems⁷, com mais de 132.000 pacotes disponíveis e mais de 250 pacotes adicionados todos os dias (MADSEN; TIP; LHOTÁK, 2015).

Além do JavaScript ser executado no ambiente *server-side*, ele está sendo executado em dispositivos móveis com aplicativos que são desenvolvidos utilizando bibliotecas (*frameworks*) de aplicações móveis híbridas, como por exemplo o PhoneGap (HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2012). As bibliotecas de aplicações híbridas são executadas em diversos sistemas operacionais de dispositivos móveis (por exemplo, Android, iOS e Windows Phone) e atuam como interpretadoras do código JavaScript para manipulação dos recursos do dispositivo (por exemplo, câmera e GPS). Esses *frameworks* não possuem componentes visuais (botões, campos de texto, tabelas e outros) e por essa razão são utilizados o HTML e CSS com auxílio de bibliotecas JavaScript (AngularJS⁸, Bootstrap⁹, Ionic¹⁰ e outros) para geração destes componentes.

3.2 Defeito, erro e falha

Os termos defeito, erro e falha estão comumente relacionados a sistemas de software como aplicativos, *sites* e programas de computadores. A seguir as definições desses termos segundo Radatz, Geraci e Katki (1990):

- Defeito (*fault* ou *bug*): um passo, processo ou definição de dados incorretos num programa de computador. É uma ação inconsistente cometida por um humano, ao tentar utilizar um método ou ferramenta, ou resolver determinado problema. Em outras palavras, é uma ação inconsistente realizada por um humano na escrita do código fonte.
- Erro (*error*): é a manifestação concretizada do defeito, ou seja, é o impacto com que o defeito interferiu na execução do código, podendo fazer com que mais defeitos

⁴ <https://developers.google.com/v8/>

⁵ <https://www.npmjs.com/>

⁶ <https://search.maven.org/>

⁷ <https://rubygems.org>

⁸ <https://angularjs.org/>

⁹ <https://getbootstrap.com/>

¹⁰ <https://ionicframework.com/>

sejam propagados no fluxo das execuções. Erro é a diferença entre o valor obtido e o valor esperado.

- Falha (*failure*): é a incapacidade do sistema de executar as funções requeridas. É o resultado, como um todo, de um comportamento do software inesperado ou indesejado.

3.3 Tipos de defeitos e code smells

Os defeitos em códigos podem ser classificados de acordo com o impacto do defeito no compilador ou no interpretador e de forma lógica. Assim, os defeitos podem ser classificados em:

- Defeito sintático: são defeitos causados pela falta de consistência gramatical com a linguagem de programação compilada ou interpretada. Por exemplo, no JavaScript, Java, C, C# ou Pascal, se for escrito o comando *ese* ou *lse* no lugar de *else*, vai ocorrer um defeito sintático.
- Defeito semântico: são defeitos causados pela inconsistência semântica da linguagem de programação. Após realizada a análise gramatical da linguagem, o compilador ou o interpretador vai verificar se a estrutura das instruções que foram escritas estão coerentes com a estrutura esperada pela linguagem. Por exemplo, ao tentar realizar uma operação matemática com uma variável que nunca foi iniciada, ocorrerá um erro semântico;
- Defeito lógico: são defeitos causados pela falta de coerência no fluxo lógico de execução do código escrito. Esse tipo de defeito só se manifesta no momento que é realizado um teste da funcionalidade esperada. Assim, se a funcionalidade não ocorrer na forma esperada, ocorreu um defeito lógico. Por exemplo, se o esperado ao pressionar um botão fosse realizar uma soma, mas ocorre uma subtração, ocorreu um defeito lógico.

A seguir é definido o termo *code smell*. *Code smells* são códigos que não possuem qualquer tipo de defeito, erro ou falha, mas podem indicar um problema em potencial. Por exemplo, se for criada uma variável *X* no escopo global através de um arquivo JavaScript, existe um problema em potencial. Se um outro programador criar a mesma variável *X* no escopo global através de um outro arquivo JavaScript, ocorrerá um problema.

Para o concerto de defeitos é necessário realizar uma correção, já para o concerto dos *code smells* é necessário realizar uma refatoração no código. A refatoração é uma técnica para remover *code smells* ou melhorar a estrutura do código, sem afetar o comportamento esperado já realizado pelo código.

3.4 Análise estática e dinâmica

Existem muitas maneiras para analisar códigos a fim de identificar defeitos, erros, falhas ou *code smells*, porém as mais comuns são:

- Análise estática: analisa-se o código sem executá-lo buscando defeitos, erros, falhas ou *code smells*.
- Análise dinâmica: para esse tipo de análise, o código é executado. São passadas informações, depois o código é executado, são coletadas informações de tempo de execução e os resultados obtidos são analisados para identificar possíveis problemas.

Em ambas os tipos de análise, o objetivo principal é identificar possíveis problemas antes que eles impactem o cliente enquanto ele estiver utilizando o software.

Atualmente, existem muitas ferramentas que fazem análise estática do código, como por exemplo JsLint¹¹ ou Closure Compiler¹². Porém, muitos defeitos são semânticos ou lógicos. Para lidar com defeitos desta natureza, analisadores semânticos de código estão sendo construídos. Por exemplo, a ferramenta TAJs¹³ (JENSEN; MØLLER; THIEMANN, 2009), que verifica tipos, funções/propriedades inexistentes e códigos não utilizados. Um outro exemplo é a ferramenta de análise estática SAFE¹⁴. Estas ferramentas geralmente criam uma Árvore de Sintaxe Abstrata (em inglês Abstract Syntax Tree – AST) e geram um grafo de fluxo de controle. Além disso, existem ferramentas que fazem análise dinâmica, como DLint¹⁵ (GONG et al., 2015) e JSNOSE¹⁶ (FARD; MESBAH, 2013), as quais verificam regras de consistência baseadas em *code smells*.

¹¹ JsLint (JavaScript Lint) disponível em <http://www.jshint.com/>

¹² <https://developers.google.com/closure/compiler/>

¹³ TAJs (Type Analysis for JavaScript) disponível em <http://www.brics.dk/TAJS>

¹⁴ SAFE (Scalable Analysis Framework for ECMAScript) disponível em <https://github.com/sukyoung/safe>

¹⁵ DLint (Dynamically checking bad coding practices in JavaScript)

¹⁶ JSNOSE (Detecting JavaScript code smells)

4 Detecção automática de defeitos em software JavaScript: Uma revisão sistemática da literatura

Foi realizada uma revisão sistemática da literatura (RSL) (MACEDO; DELGADO; WIESE, 2017) pelo autor desta dissertação. O objetivo desta revisão foi identificar os métodos e ferramentas para detecção automática de defeitos e *code smells* em sistemas de software desenvolvidos na linguagem JavaScript (i) buscando catalogar os defeitos e *code smells* mais procurados pelos estudos, (ii) identificando os ambientes de desenvolvimento que são mais usados e (iii) descobrindo as lacunas existentes na área.

4.1 Método da revisão sistemática

A revisão sistemática da literatura é um estudo secundário que fornece um processo rigoroso e replicável para rever as evidências relevantes para uma determinada questão de pesquisa. Uma RSL tem três etapas principais (KITCHENHAM, 2012): planejamento, condução e relatório. Na etapa de planejamento é produzido um protocolo descrevendo os componentes da RSL, o que permite a replicabilidade do método. Na etapa de condução são selecionados os estudos seguindo o protocolo. Na etapa de relatório, os resultados obtidos são divulgados e analisados. A seguir, são descritas cada uma dessas etapas.

4.1.1 Planejamento

Questão de pesquisa

Na Tabela 1 são apresentadas as questões de pesquisa usadas na RSL. Na primeira questão (RQ1) é levantado um dos aspectos mais importantes, que é identificar quais os métodos utilizados para detecção automática de defeitos em JavaScript.

A questão RQ2 tem o objetivo de determinar algumas das características dos métodos propostos. Para isso, foram identificados: (i) os tipos de defeitos e *code smells* que podem ser encontradas pelos métodos de detecção automática (RQ2-1); (ii) se os métodos propostos na literatura têm alguma base de dados com os defeitos catalogados (RQ2-2); (iii) se as abordagens da literatura propostas para detecção de defeitos são para *client-side*, *server-side*, *mobile* ou para aplicativos (RQ2-3); e (iv) se a detecção é realizada em tempo de desenvolvimento ou pós desenvolvimento (RQ2-4).

A RQ3 tem por objetivo determinar algumas das características dos experimentos realizados. Para isso são identificados: (i) os sistemas de software que são utilizados nos experimentos (RQ3-1); (ii) o tipo do repositório de código fonte utilizado para realizar os testes dos métodos de detecção (RQ3-2); (iii) se são feitas comparações com outras técnicas (RQ3-3); (iv) o número de projetos ou aplicações utilizadas nos experimentos (RQ3-4); e (v) os principais resultados obtidos em termos de precisão, cobertura e tempo (RQ3-5).

Tabela 1 – Questões de Pesquisa

RQ#	Questões de Pesquisa
RQ1	Quais os métodos usados para realizar a detecção automática de defeitos?
RQ2	Quais as características dos métodos propostos?
RQ2-1	Que tipo de defeitos e <i>code smells</i> são detectados pelos métodos propostos?
RQ2-2	Os métodos propostos têm alguma base de dados predefinida de defeitos?
RQ2-3	Os métodos propostos são utilizados em JavaScript <i>client-side server-side</i> , <i>mobile</i> ou aplicativos?
RQ2-4	Os métodos de detecção de defeitos são executados durante o desenvolvimento do software ou após?
RQ3	Quais as características dos experimentos realizados?
RQ3-1	Quais os sistemas de software utilizados nos experimentos?
RQ3-2	Os testes são realizados com repositórios de código fonte públicos ou privados?
RQ3-3	Com quais outras técnicas foram comparados os métodos propostos?
RQ3-4	Quantos projetos ou aplicações foram utilizados nos experimentos?
RQ3-5	Quais foram os principais resultados dos experimentos em termos de precisão, cobertura e tempo?

Seleção de fontes e string de Busca

Foram utilizadas duas bases de dados na RSL, Scopus e Web of Science, pois sabe-se que o motor de pesquisa do Scopus indexa as bases de dados da IEEE Xplore, Science Direct, ACM Digital Library e Engineering Village. Nas *string* de busca foram incorporados os termos e sinônimos alternativos nos campos título, resumo e palavras-chave dos artigos (TITLE-ABS-KEY para a base Scopus e TS para a base Web of Science). Assim, foram geradas as *strings* para as duas bases de dados (Tabela 2), com base em alguns artigos de controle.

Tabela 2 – Strings de busca

Fonte	String
Scopus	<i>TITLE-ABS-KEY ((detection OR (code AND analysis) OR (static AND analysis)) AND (bug OR error OR fault)) AND TITLE-ABS-KEY (java*script)</i>
Web of Science	<i>(TS= ((detection OR (code AND analysis) OR (static AND analysis)) AND (bug OR error OR fault)) AND TS= (javascript)</i>

Crterios de incluso e excluso

Os critrios de incluso (CI) e excluso (CE) foram desenvolvidos para selecionar os estudos primrios, estabelecendo critrios mnimos de qualidade e dessa maneira, selecionar fontes acessveis para consultas posteriores. Os critrios foram:

CI-1 Os artigos devem estar disponveis integralmente online via a instituio dos autores da RSL;

CI-2 Os artigos devem estar inteiramente escritos em lngua inglesa.

CE-1 Trabalho de deteco de defeitos que no so automatizados.

CE-2 Artigos que no tenham como foco principal a deteco de defeitos em sistemas implementados em JavaScript.

CE-3 Artigos publicados em veculos que no exigem reviso por pares, por exemplo, captulos de livro.

CE-4 Estudos secundrios ou tercirios.

CE-5 Estudos no relacionados s reas de Cincias da Computao, Matemtica e Engenharia.

CE-6 Estudos incompletos ou que no relatam de maneira adequada os mtodos ou experimentos realizados.

CE-7 Artigos focados em testes e criao de testes automticos.

CE-8 Artigos com nfase em vulnerabilidade de segurana em software escrito em JavaScript.

4.1.2 Conduo

A pesquisa realizada no motor de busca Scopus em 8 de fevereiro de 2017 retornou 95 trabalhos e no motor Web of Science retornou 44 estudos. Do total de 139 artigos,

16 foram excluídos pelos critérios de exclusão CE-3, CE-4, CE-5, CE-7 por meio das ferramentas de filtros dos motores de busca, restando 123. Destes 123, foram identificados 42 artigos duplicados. Após isso, os critérios foram aplicados manualmente em cada um dos 81 estudos restantes analisando o título, resumo e palavras chave, e no caso de dúvida o estudo foi analisado por completo. No fim desta etapa foram selecionados 19 estudos para o desenvolvimento da RSL.

Para auxiliar na condução da RSL foi utilizado o software StArt (*State of the Art through Systematic Review*)¹.

4.2 Resultados da revisão sistemática

Na Tabela 3 são listados os estudos selecionados ordenados pelo ano de publicação. O identificador ID inclui o veículo de publicação (C–Congresso e J–Periódico) e um número de sequência. Note que, apesar de não termos restringido o ano de publicação nos motores de busca, todos os estudos selecionados foram publicados a partir de 2009.

4.2.1 Catálogo de defeitos e *code smells* em JavaScript

Um dos poucos trabalhos que fazem uma categorização de defeitos é (HANAM; BRITO; MESBAH, 2016). Porém, nesse trabalho é apresentada apenas uma lista pequena de defeitos mais comuns. Dessa forma, um dos objetivos da RSL é fornecer um catálogo de defeitos e *code smells* mais completo criado a partir dos estudos selecionados. Nas Tabelas 4 e 5 cada defeito e *code smell* apresenta (i) um identificador (ID) que começa com Bg e Cs, respectivamente; (ii) o nome e o tipo entre parênteses (no caso de ser defeito), a descrição e o ambiente em que pode ser identificado.

4.2.2 Métodos utilizados para detecção: RQ1

A seguir os estudos são agrupados de acordo com a ferramenta com a qual estão relacionados.

¹ http://lapes.dc.ufscar.br/tools/start_tool

Tabela 3 – Estudos selecionados na RSL

ID	Referência	Título	Ano
C1	(JENSEN; MØLLER; THIEMANN, 2009)	Type Analysis for JavaScript	2009
C2	(KO; WOBBROCK, 2010)	Cleanroom: Edit-time error detection with the uniqueness heuristic	2010
C3	(KO; ZHANG, 2011)	FeedLack detects missing feedback in web applications	2011
C4	(ZHENG; BAO; ZHANG, 2011)	Statically locating web application bugs caused by asynchronous calls	2011
C5	(JENSEN; MADSEN; MØLLER, 2011)	Modeling the HTML DOM and browser API in static analysis of JavaScript Web Applications	2011
C6	(JR; PATTABIRAMAN; MESBAH, 2012)	AutoFLox: An automatic fault localizer for client-side JavaScript	2012
C7	(FARD; MESBAH, 2013)	JSNOSE: Detecting javascript code smells	2013
C8	(MADSEN; LIVSHITS; FAN-NING, 2013)	Practical static analysis of JavaScript applications in the presence of frameworks and libraries	2013
C9	(PARK, 2014)	JavaScript API misuse detection by using typescript	2014
C10	(KASHYAP et al., 2013)	Type Refinement for Static Analysis of JavaScript	2014
C11	(HSIAO; CAFA-RELLA; NARAYANASAMY, 2014)	Using web corpus statistics for program analysis	2014
C12	(LIANG et al., 2014)	PatBugs: A Pattern-Based Bug Detector for Cross-Platform Mobile Applications	2014
C13	(JR; PATTABIRAMAN; MESBAH, 2014)	Vejovis: Suggesting fixes for JavaScript faults	2014
C14	(BAE et al., 2014)	<i>SAFE_{WAPI}</i> : Web API misuse detector for web applications	2014
C15	(ANDREASEN; MØLLER, 2014)	Determinacy in static analysis for jQuery	2014
C16	(PARK et al., 2015)	Static analysis of JavaScript web applications in the wild via practical DOM modeling	2015
C17	(YOUNANG; LU, 2015)	Improving Precision of JavaScript Program Analysis with an Extended Domain of Intervals	2015
C18	(MADSEN; TIP; LHOTÁK, 2015)	Static analysis of event-driven Node.js JavaScript applications	2015
J19	(OCARIZA et al., 2016)	Automatic fault localization for client-side JavaScript	2016

Estudos relacionados com a ferramenta TAJs

No trabalho **C1** foi criado o *plug-in* para o IDE Eclipse que analisa programas JavaScript chamado TAJs. Nesse trabalho é usada a técnica de *interpretação abstrata*

Tabela 4 – Catálogo de defeitos.

ID	Nome da defeitos	Descrição	Ambiente
Bg1	Função inexistente (<i>SE</i>)	Chamada realizada para uma função inexistente.	Todos
Bg2	Variável inexistente (<i>SE</i>)	Acessar uma variável <i>null</i> ou <i>undefined</i> .	Todos
Bg3	Propriedade inexistente (<i>SE</i>)	Acessar propriedades inexistentes, retornando <i>null</i> ou <i>undefined</i> .	Todos
Bg4	Número de argumentos inválidos (<i>SE</i>)	Acionando funções com um número errado de argumentos.	Todos
Bg5	Tipo de argumento inesperado (<i>SE</i>)	Passar para a função, argumentos com tipos distintos do esperado.	Todos
Bg6	Exceção não tratada (<i>LO</i>)	Erros que geram exceção e que não são tratadas.	Todos
Bg7	Erro na manipulação do DOM (<i>SI</i>)	Acessar objeto DOM inexistente, página ou tipo do objeto inesperado.	<i>client-side</i>
Bg8	Funções de eventos inexistente ou inválidas (<i>SE</i>)	Erros gerados na interação do usuário com a página web, ocorridos normalmente em eventos que não possuem funções JavaScript ou inválidas.	<i>client-side</i>
Bg9	Uso errado do <i>instanceof</i> (<i>LO</i>)	Nenhum objeto usado nos lados do operador do <i>instanceof</i> .	Todos
Bg10	Condicional inoperante (<i>LO</i>)	Expressões condicionais com valores constantes, de forma a sempre retornar valor verdadeiro ou sempre falso.	Todos
Bg11	Erros em execução assíncrona (<i>SE</i>)	Erros que podem ocorrer em execução assíncrona com AJAX (EERNISSE, 2006), causando problemas de execução por ordem inesperada.	Todos
Bg12	Ausência de funções <i>callback</i> de erro (<i>SE</i>)	Caso ocorra erro durante a execução da função em API, sem função <i>callback</i> , não será possível saber o erro que ocorreu.	Todos
Bg13	Elemento DOM inexistente (<i>SE</i>)	Propriedade dentro de um elemento DOM inexistente.	<i>client-side</i>
Bg14	<i>Loop</i> infinito (<i>LO</i>)	Laço sem condição de parada.	Todos
Bg15	Erro de tipo numérico (<i>SI</i>)	Erros que podem ocorrer com relação ao tipo.	Todos
Bg16	Erro de cálculo numérico (<i>SI</i>)	Erros que podem ocorrer no cálculo matemático em JavaScript.	Todos
Bg17	Erro na chamada do método (<i>SE</i>)	Instrução de chamada de método que envolve Of, chamada (<i>MethodCall</i>).	<i>Mobile</i>
Bg18	Erro no retorno (<i>SE</i>)	Uma instrução de retorno que retorna Of.	<i>Mobile</i>
Bg19	<i>Assign</i> (<i>SE</i>)	Uma declaração de atribuição que envolve Of.	<i>Mobile</i>
Bg20	<i>NullEqualityIfCheck</i> (<i>SE</i>)	Uma instrução <i>if</i> que faz uma verificação de igualdade entre Of e <i>null</i> .	<i>Mobile</i>
Bg21	<i>DirectIfNotCheck</i> (<i>SE</i>)	Uma instrução <i>if</i> cuja condição Of é diretamente prefixada com o operador de negação, tais como " <i>if</i> (!Of)".	<i>Mobile</i>
Bg22	Converter <i>undefined</i> para número (<i>SI</i>)	Converter <i>undefined</i> para número, retornando tipo <i>NaN</i> (não é um número).	Todos
Bg23	Invocar um valor não funcional (<i>SI</i>)	Tentar invocar um objeto ou valor como função.	Todos
Bg24	Variáveis ou propriedades de objeto inacessíveis (<i>SE</i>)	Tentar escrever ou acessar variáveis ou propriedades de objeto inacessíveis.	Todos

Legenda: *SI*: Defeito sintático; *SE*: Defeito semântico; *LO*: Defeito lógico.

e é gerado um grafo do fluxo de dependência das funções JavaScript do arquivo (".js") através da ferramenta WALA².

Em **C5**, foi desenvolvida uma extensão do TAJIS (JENSEN; MØLLER; THIEMANN, 2009) que diferentemente da versão inicial, utiliza uma análise de controle. Também utiliza um modelo de *heap* baseado na abstração do local de alocação, que identifica anomalias em funções, por exemplo no jQuery que tem um comportamento diferente dependendo se é passado como parâmetro de entrada uma *string* HTML, uma função, uma *string* CSS ou um elemento DOM.

O estudo **C15** realiza uma análise estática, baseada na investigação sistemática de imprecisão da ferramenta TAJIS (desenvolvida em C1). Nessa ferramenta, foram incluídos o contexto seletivo e a sensibilidade do caminho, que são variações de ideias bem conhecidas, mas que nunca tinham sido utilizadas para analisar programas JavaScript com jQuery.

² WALA (T. J. Watson Libraries for Analysis) disponível em <http://wala.sourceforge.net/>

Tabela 5 – Catálogo de *code smells*.

ID	Nome da defeitos	Descrição	Ambiente
Cs1	Variável morta	Variável não está sendo utilizada mais em nenhum lugar.	Todos
Cs2	Argumento morto	Argumento que não está sendo utilizado em nenhum lugar da função.	Todos
Cs3	Código morto	Trecho de código não utilizado.	Todos
Cs4	Conversões em tempo de execução	Conversões de tipos primitivos ou não, durante a execução do software.	Todos
Cs5	Evento morto	Evento que não está sendo utilizado em nenhum lugar do software.	Todos
Cs6	Modificação do método ou propriedade em tempo de execução	Alterar o método ou a propriedade durante o momento de execução do software.	Todos
Cs7	<i>Loop</i> oneroso	Oneroso.	Todos
Cs8	Código duplicado no projeto	Código repetido dentro do mesmo projeto.	Todos
Cs9	Código duplicado em projetos de terceiros	Código repetido em projetos de terceiros, que são referenciados dentro do projeto.	Todos
Cs10	Retorno morto	Retorno da função não utilizado em nenhum lugar.	Todos
Cs11	Blocos de <i>catch</i> vazio	Utilizar <i>try/catch</i> sem capturar a lógica do erro.	Todos
Cs12	Objeto grande	Objeto com muitas responsabilidades.	Todos
Cs13	Objeto preguiçoso	Objeto com pouca funcionalidade.	Todos
Cs14	Função longas	Função com muitas responsabilidades.	Todos
Cs15	Lista de parâmetros longa	Função com muitos argumentos.	Todos
Cs16	Instruções <i>Switch</i>	Código duplicado e com alta complexidade.	Todos
Cs17	<i>Closure Smells</i>	Funções aninhadas com encadeamento de escopo.	Todos
Cs18	Acoplamento entre JavaScript, HTML e CSS	Código JavaScript acoplado com HTML e CSS.	<i>client-side</i>
Cs19	Excessivas variáveis globais	Muitas variáveis globais, podendo ter dados inconsistentes.	Todos
Cs20	Cadeia de mensagens longa	Trecho de código que é muito longo e aninhado. Por exemplo, o encadeamento de funções em jQuery.	Todos
Cs21	Retorno de chamada aninhado	Funções aninhadas, retornando para outra função, diretamente nos parâmetros de entrada.	Todos
Cs22	Recusa legado	Altera muitas propriedades da função herdada (função pai), a ponto de não fazer sentido existir a herança.	Todos
Cs23	Nomes únicos	Nomes de variáveis, funções ou propriedades, que aparecem apenas uma vez no projeto.	Todos

Estudos relacionados com a ferramenta SAFE

No artigo **C9** a ferramenta SAFE foi estendida. Foi utilizado o repositório do GitHub chamado DefinitelyTyped³, que fornece definições dos tipos (tipagem dos atributos e funções) para TypeScript⁴, das mais de 300 bibliotecas de JavaScript frequentemente utilizadas. Desta forma, criou-se uma ferramenta de análise escalável utilizando tipos extraídos da especificação correspondente das bibliotecas em TypeScript. Os arquivos HTML e JavaScript são analisados gerando árvores DOM a partir de arquivos HTML e ASTs para serem comparados com o catálogo de definições *DefinitelyTyped*.

No estudo **C14**, os autores estenderam a ferramenta SAFE propondo o *SAFE_{WAPI}*⁵. O foco principal é analisar o uso indevido das APIs Web, cometido pelos desenvolvedores. Utilizaram as especificações das APIs Web, que são disponibilizadas pelos fornecedores dos serviços e são frequentemente declaradas em linguagens descritivas de interfaces (IDLs). Assim, a detecção é realizada em cima destas descrições e está focada em aplicações *mobile*.

³ <https://github.com/DefinitelyTyped/DefinitelyTyped>

⁴ <https://www.typescriptlang.org/>

⁵ *SAFE_{WAPI}* (Web API misuse detector for web applications)

No estudo **C16** estendeu-se a ferramenta SAFE criando o *SAFE_{WApp}*⁶. Os autores utilizaram o Jericho⁷ para extrair o código JavaScript e seu local de origem do HTML, e em seguida utilizam o CyberNeko⁸ para construir uma árvore DOM. O resultado é passado para os módulos *DOMModeler* e *DOMBuilder*. O *DOMModeler* adiciona a um objeto protótipo do DOM, simulando um *browser*. O *DOMBuilder* traduz uma árvore DOM concreta para uma DOM abstrata. Depois de analisar um determinado programa, são analisados todos os manipuladores de eventos registrados pelo programa, para posteriormente combinar com segurança os resultados da análise.

Estudos que usam o *parser* de JavaScript da ferramenta Rhino

Em **C3**, foi realizada uma análise estática. Primeiro é feita a identificação do código JavaScript (arquivos *“.js”* e em *tags scripts* no HTML), utilizando o *parser* de JavaScript do Rhino⁹ que gera um conjunto de ASTs que posteriormente são convertidas em grafos de fluxo de controle. Finalmente são criados *clusters*, agrupando as possíveis saídas de cada função. É realizada uma iteração através de todos os caminhos no *cluster* para identificar quais levam a uma sequência crítica.

No estudo **C13**, o processo para detecção começa a partir dos métodos e propriedades do DOM API, tendo uma combinação de análise estática e dinâmica para identificar as linhas de código defeituosas. Primeiro é utilizado o Rhino para obter os códigos JavaScript que são executados no software através de CRAWLJAX (MESBAH; DEURSEN; LENSELINK, 2012) e depois, um algoritmo que procura por substituições viáveis entre os parâmetros válidos encontrados. As abordagens foram implementadas na ferramenta Vejovis¹⁰, que também fornece sugestões de correção para os defeitos encontrados.

Outros

No estudo **C2**, foi apresentada uma ferramenta simples chamada Cleanroom que é uma versão modificada do editor Bepin¹¹. A ferramenta destaca qualquer nome ou

⁶ *SAFE_{WApp}* (Static analysis framework for JavaScript web applications)

⁷ <http://jericho.htmlparser.net>

⁸ <http://nekohtml.sourceforge.net>

⁹ <http://www.mozilla.org/rhino/>

¹⁰ Vejovis (Suggesting fixes for JavaScript faults)

¹¹ <https://wiki.mozilla.org/Labs/Bepin>

par de nomes que aparece apenas uma vez no programa. São identificados os *tokens* (identificadores únicos de trechos de código) e aplicadas expressões regulares simples. Por fim, é utilizado o cálculo da distância de cordas de Levenshtein (LEVENSHTAIN, 1966) entre os *tokens* e a lista que contém todos os trechos de código encontrados no momento da análise.

Em **C4** foi proposto um analisador estático de código inter-procedural para verificar possíveis defeitos que podem ocorrer em requisições assíncronas Ajax (EERNISSE, 2006). Os autores utilizaram Datalog, que é uma linguagem de consulta com notação Prolog (LEVY; RAJARAMAN; ORDILLE, 1996), a fim de fornecer uma representação para a análise de programa, especialmente de fluxo. Através da extração de códigos JavaScript são verificadas as chamadas Ajax, realizando análise em cima das funções encontradas, exceto os corpos (código interno) das bibliotecas de terceiros, pois buscaram apenas abstrações para as funções de interface.

Nos estudos **C6** e **J19** foram utilizadas a análise dinâmica e a técnica *Dynamic Backward Slicing* (AGRAWAL et al., 1995) na ferramenta AutoFlox¹². Esta ferramenta pode rodar através do CRAWLJAX (MESBAH; DEURSEN; LENSELINK, 2012) e como *plug-in* do Eclipse. A abordagem realiza a separação das funções e executa os passos de cada uma delas. São utilizados os marcadores ASYNCCALL, ASYNC e FAILURE/ERROR. O marcador FAILURE/ERROR é adicionado quando um erro é detectado. O ASYNCCALL é colocado após uma chamada assíncrona para uma função e colocado no início da execução da função assíncrona. A função assíncrona contém o nome da função ASYNC bem como o seu identificador. Diferente de C6, o J19 pode localizar defeitos em funções *eval*, funções anônimas e código JavaScript minificado.

Em **C7**, foram utilizadas também ASTs aplicadas na ferramenta JSNOSE, que realiza uma análise estática e dinâmica do código. A ferramenta intercepta em alto nível o código JavaScript da aplicação web que vai analisar, com um proxy configurado entre o servidor e o navegador. Depois, é extraído o código JavaScript de todos os arquivos nos formatos *.js* e HTML, para logo em seguida, realizar a conversão do código em AST. Então, é feita uma análise em cada entidade, objeto, propriedade, função e bloco de código do programa JavaScript. Nesse processo, são coletados dados estáticos e dinâmicos para realizar a comparação com uma lista predefinida de códigos que podem indicar um problema potencial (*code smells*).

¹² AutoFlox (An automatic fault localizer for client-side JavaScript)

No estudo **C8** foi feita a combinação da tradicional *análise de ponteiro* (GUARNIERI; LIVSHITS, 2009) e uma nova análise chamada de *análise de uso*. Um dos principais objetivos deste estudo é conseguir analisar bibliotecas de terceiros dentro do software escrito em JavaScript. A análise proposta é declarativa e expressa como uma coleção de regras de inferência, permitindo fácil manutenção, portabilidade e modificação. Este trabalho faz análise iterativa parcial ou completa. A análise iterativa parcial depende da existência dos *stubs*, que são mecanismos de especificação parcial para resolver o problema da biblioteca. A análise iterativa completa, independe de quaisquer *stubs* ou especificações de biblioteca. Esta abordagem foi implementada na API WinRT, com uso em aplicativos de Windows 8.

Em **C10** foi proposta a técnica chamada *refinamento de tipo*, que trabalha com um domínio abstrato genérico criado com base em tipos. Essa técnica analisa o código e identifica o tipo (por exemplo de *string*, inteiro ou lógico) da variável ou propriedade. A chave deste trabalho é reconhecer que a semântica de JavaScript inclui muitas verificações condicionais implícitas nos tipos e que a execução do refinamento de tipo sobre essas verificações implícitas proporcionam um benefício significativo para a precisão da análise.

No estudo **C11**, foram utilizados (i) o modelo N-Gram de frequência de texto, técnica comumente utilizada nos estudos de processamento de linguagem natural; e (ii) tokenização, para extrair as características dos trechos de código a partir do código fonte bruto. As declarações que carregam termos e conjuntos de palavras relevantes da linguagem são decompostas e depois é criado um grafo. Com a estrutura convertida em grafo, é feita uma *busca em largura* para identificar os códigos duplicados. O método proposto pode ser usado para detecção de plágio e para encontrar defeitos por ter código repetido.

O estudo **C12** tem como foco a detecção automática de defeitos temporais através de padrões para aplicações móveis e multiplataforma durante o desenvolvimento. As verificações são realizadas para detectar defeitos provenientes do ambiente *mobile*. Utilizaram uma notação chamada de *Flexible Bug Pattern Specification Notation* que trabalha com autômatos de estados finito para especificar defeitos típicos ou cenários de uso incorreto de objetos com padrões de defeitos já especificados. Os autores implementaram essa abordagem na ferramenta chamada PatBugs¹³, que é usada em aplicativos *mobile* desenvolvidos em Apache Cordova¹⁴.

¹³ PatBugs (Pattern-Based Bug detector for cross-platform mobile applications)

¹⁴ <https://cordova.apache.org/>

No trabalho **C17**, o objetivo foi melhorar o desempenho da ferramenta de análise JSAI¹⁵ (KASHYAP et al., 2014) que não tinha uma análise de tipos numéricos. Focaram exclusivamente numa abordagem que busca identificar e analisar tipos numéricos chamada de *domínio estendido de intervalos numéricos*, a qual ajudou a fazer uma inferência mais precisa do tipo.

Em **C18**, os autores mostraram que os grafos de chamadas baseadas em eventos (clique, pressionar de teclada, etc.), podem ser usados como base para detectar vários tipos de defeitos e apresentar uma família de análises. As análises sensíveis ao evento e ao ouvinte (*listener*) calculam fatos dos fluxos de dados separados com base nos eventos que foram emitidos e no que está registrado nos *listeners*, respectivamente. A implementação desta abordagem foi realizada na ferramenta de análise estática, chamada RADAR, para ambientes de *server-side* com Node.js¹⁶.

4.2.3 Características do método proposto: RQ2

Os estudos selecionados foram organizados na Tabela 6 para responder as questões RQ2-1, RQ2-2, RQ2-3 e RQ2-4.

Entre os tipos de defeitos que são mais buscados pelos estudos estão Bg1, Bg2 e Bg3 que foram descritos na Tabela 4. Foi possível verificar que 52,6% dos estudos tentam detectar o defeito Bg2 e 31,6% dos estudos buscam detectar os defeitos Bg1 e Bg3.

Em relação ao ambiente para o qual o método foi proposto, 78,9% dos estudos selecionados estão focados em detectar defeitos ou *code smells* do ambiente *client-side*. Contudo, observa-se estudos, entre eles C8, C12, C14 e C18, que têm o objetivo de detectar defeitos ou *code smells* em aplicativos, *mobile* e *server-side*.

É importante destacar que os estudos C6 e J19 encontram defeitos provenientes de *exceções não tratadas* e sem possuir nenhuma base de defeitos predefinidos. Além disso, realizam as detecções durante e após o desenvolvimento do software. Desta forma, essa abordagem pode detectar vários tipos de defeitos que não foram previamente catalogados, como por exemplo, defeitos que ocorrem por causa de dados ou por interação do usuário com o software.

¹⁵ JSAI (Static analysis platform for JavaScript)

¹⁶ <https://nodejs.org/>

Tabela 6 – Características do método proposto dos estudos selecionados na RSL

ID	Tipo de defeitos	Base predefi- nida de defei- tos	<i>Client-side</i> , <i>server- side</i> , <i>mobile</i> , aplica- tivos	Durante ou após desenvol- vimento
C1	Bg2, Bg3, Bg4, Cs3, Cs4, Bg22, Bg23, Bg24	Sim	<i>client-side</i>	Durante
C2	Bg2, Cs23	Sim	<i>client-side</i>	Durante
C3	Bg8	Sim	<i>client-side</i>	Durante
C4	Bg11	Não	<i>client-side</i>	Durante
C5	Bg2, Bg3, Cs3, Bg24	Sim	<i>client-side</i>	Ambos
C6	Bg6	Não	<i>client-side</i>	Ambos
C7	Cs3, Cs11, Cs12, Cs13, Cs14, Cs15, Cs16, Cs3, Cs17, Cs18, Cs19, Cs20, Cs21, Cs22	Sim	<i>client-side</i>	Durante
C8	Bg2, Cs2, Cs3, Cs10	Não	Aplicativos dows 8	Win- Ambos
C9	Bg1, Bg2, Bg3, Bg4, Bg5,	Sim	<i>client-side</i>	Durante
C10	Bg1, Bg2, Cs4	Sim	<i>client-side</i>	Após
C11	Cs8, Cs9	Não	<i>client-side</i>	Durante
C12	Bg17, Bg18, Bg19, Bg20, Bg21	Sim	<i>mobile</i>	Durante
C13	Cs1, Bg4, Bg7	Sim	<i>client-side</i>	Ambos
C14	Bg1, Bg2, Bg3, Bg4, Bg5	Sim	<i>mobile</i>	Após
C15	Bg5, Cs3, Cs7, Bg14	Não	<i>client-side</i>	Após
C16	Bg2, Bg3, Bg9, Bg10, Bg22, Bg23	Não	<i>client-side</i>	Após
C17	Bg1, Bg2, Bg3, Bg15, Bg16	Sim	<i>client-side</i>	Após
C18	Bg1, Bg2, Bg3, Cs5	Sim	<i>server-side</i>	Durante
J19	Bg6	Não	<i>client-side</i>	Ambos

Tabela 7 – Características dos experimentos realizados nos estudos selecionados na RSL

ID	Características dos softwa- res	Reposi- tório	Comparação com	Nº de projetos aplicações	ou	Preci- são	Cober- tura	Tempo
C1	Projetos JavaScript	Público	Nenhum	7 projetos		NI	NI	<6min
C2	Aplicações web	Público	Nenhum	1 projeto		NI	NI	NI
C3	Aplicações web	Público	Nenhum	330 aplicações		18%	NI	NI
C4	Ajax nas aplicações web	Público	Nenhum	8 aplicações		NI	NI	<13s
C5	Aplicações web sem bibli- otecas	Público	Nenhum	53 aplicações web Ja- vaScript		90%	NI	NI
C6	Aplicações web	Ambos	Nenhum	9 projetos e 7 aplicações web		NI	NI	NI
C7	Aplicações web	Público	Nenhum	11 aplicações web		93%	98%	NI
C8	Aplicativos JavaScript do Windows 8	Privado	Ferramentas como Eclipse, IntelliJ , Visual Studio	25 projetos		81,5%	NI	<10s
C9	Projeto de TypeScript	Público	Nenhum	NI		NI	NI	NI
C10	Frameworks e software JS	Público	Nenhum	8 aplicações JavaScript		86%	NI	NI
C11	Aplicações de JavaScript	Público	Nenhum	100.000 projetos		23,9%	NI	NI
C12	Aplicações <i>Mobile</i>	Privado	Nenhum	3 projetos		84%	NI	<5,1s
C13	Aplicações web	Público	Nenhum	11 projetos		91%	73%	<91,1s
C14	Aplicações JavaSript de TV Inteligente e para An- droid	Privado	Nenhum	43 aplicações		NI	NI	NI
C15	Aplicações web com jQuery	Público	Nenhum	154 programas		53%	NI	<24s
C16	Aplicações web e bibliote- cas JavaScript	Público	TAJS com Gatekeeper	5 bibliotecas de JavaS- cript e 10 <i>sites</i>		75,5%	NI	<3h
C17	Sistemas de software Ja- vaScript	Público	Nenhum	19 programas JavaS- cript		NI	NI	NI
C18	Aplicações Node.js	Público	Nenhum	6 programas		NI	NI	NI
J19	Aplicações web	Público	Nenhum	3 projetos		NI	NI	<33,2s

4.2.4 Características dos experimentos: RQ3

As respostas para as questões RQ3-1, RQ3-2, RQ3-3, RQ3-4 e RQ3-5, de cada um dos estudos selecionados, são mostradas na Tabela 7 em que NI significa *Não Informado*.

Os sistemas de software que mais foram utilizados nos experimentos, são as aplicações web obtidas em repositórios públicos de código fonte. Note que, a maioria dos estudos não realizam comparações com outras abordagens nos experimentos. A quantidade de projetos ou aplicações usadas nos experimentos variam entre 1 e 100.000. O estudo que obteve maior precisão e cobertura foi o C7, sendo que o menor tempo usado para encontrar os defeitos foi 5,1s do estudo C12.

4.3 Conclusões da revisão sistemática

A revisão sistemática da literatura em (MACEDO; DELGADO; WIESE, 2017) investigou e avaliou 19 estudos primários relacionados a detecção automática de defeitos e *code smells* em JavaScript, visando buscar respostas para as questões da pesquisa. Pode-se observar que o método mais utilizado entre os estudos para fazer as análises é criar ASTs e analisá-las. Contudo, alguns pesquisadores optam por estender ferramentas existentes, como ocorreu com os trabalhos que estenderam as ferramentas de análise SAFE e TAJIS.

A maioria dos estudos estão voltados para ambiente web (*client-side*), todavia poucas pesquisas focaram em detectar defeitos e *code smells* em bibliotecas de terceiros (*frameworks* e *plug-ins*). O que abre espaço para que pesquisas se desenvolvam nessa linha, pois muitos desenvolvedores buscam utilizar bibliotecas, como por exemplo o jQuery, para facilitar o desenvolvimento. O mesmo ocorre nos estudos para software de ambientes multiplataforma, principalmente em *server-side*, pois na revisão foram encontrados apenas quatro estudos focados na detecção de defeitos e *code smells* para esses ambientes.

As abordagens e ferramentas de descoberta e detecção de defeitos são voltadas para componentes ou ambientes específicos da linguagem como, por exemplo, estudos voltados apenas para bibliotecas JavaScript (Modernizr (JOSHI, 2012) e jQuery (ANDREASEN; MØLLER, 2014)), para aplicações que são executadas apenas do lado do servidor ou apenas para aplicativos em dispositivos móveis. Assim, as soluções propostas deixam de descobrir e detectar defeitos inerentes aos cenários de execução de código.

A grande maioria dos estudos utilizaram repositório de código fonte público, como o GitHub, sendo essa uma ótima fonte de dados para estudos na área.

5 Estratégia para descoberta de padrões de defeitos de Hanam et.al

Neste capítulo é descrito o trabalho base utilizado nesta dissertação, que é o trabalho de Hanam, Brito e Mesbah (2016) o qual apresenta uma estratégia semiautomática para descoberta dos padrões de defeitos mais comuns em sistemas escritos na linguagem JavaScript *server-side*. A ideia principal desse trabalho é que os defeitos causados pelo uso incorreto da linguagem podem ser agrupados de acordo com as mudanças no decorrer do desenvolvimento do projeto e podem ser encontrados nos *commits* de correções dos repositórios. A seguir são descritas as etapas dessa abordagem, os resultados obtidos por Hanam, Brito e Mesbah (2016) e a utilização da ferramenta BugAID em que foi implementada essa abordagem. No Capítulo 6, o módulo de extração de características do BugAID é estendido.

5.1 Etapas da abordagem de descoberta de padrões

As etapas da abordagem de Hanam, Brito e Mesbah (2016) são: (i) mineração dos códigos fonte realizada no repositório GitHub; (ii) extração das mudanças básicas dos resultados obtidos na mineração; e (iii) clusterização dos padrões de defeito. Essa abordagem foi implementada na ferramenta BugAID.

5.1.1 Mineração dos dados

Em (HANAM; BRITO; MESBAH, 2016) foi realizada a mineração de dados no repositório de código fonte público GitHub (THUNG et al., 2013). O trabalho focou apenas nos *commits* e desconsideraram as *issues* (relatórios de defeitos) pois, normalmente os desenvolvedores não criam *issues* ou associam *issues* a um *commit*, segundo os autores. Na busca realizada, os *commits* deveriam ter entre um e seis *instruções de modificação* que são: *inserido*, *atualizado*, *removido*, *movido*, *desconhecido* e *inalterado*.

5.1.2 Extração de mudanças básicas

Dado o grande número de *commits* obtidos do GitHub, o desafio é selecionar o melhor vector de características com as informações do código e um algoritmo de agrupamento de modo que um ser humano consiga inspecionar os *clusters* gerados. Com esse intuito, os autores optaram pelo vetor de característica que capture mudanças semânticas desconhecidas, ignorando informações não importantes, como por exemplo: nome de variáveis e pequenas mudanças realizadas no código.

Assim, são detectados de forma automática tipos de alterações, chamados de *Tipos de Mudança Básica* (em inglês *Basic Change Types* – BCTs). Os BCTs são estruturas compostas pelas seguintes propriedades:

- O *tipo de construção da linguagem* (representado por **TipoConstrucao**), que são os tipos de instruções, palavras reservadas, operadores, comportamento e outros. Esses tipos são: comportamento (B), método (M), classe (CL), erro (EX), evento (EV), campos (F), parâmetro (P), variáveis (V), constante (CO), conversão (CV), reservado (RE);
- O *contexto de construção da linguagem* (representado por **ContextoConstrucao**), que é o contexto em que os tipos de instruções aparecem. Por exemplo, a palavra reservada *this* poderia ser usada em muitos contextos diferentes, como dentro de uma condição ou como um argumento. Alguns desses contextos são: chamada de método (MC), condição (C), declaração de variável (VD), declaração de parâmetro (PD), entre outros;
- Os *tipos de modificações* (representado por **TipoModifica**), que representam a forma que a instrução (construção da linguagem) foi modificada no *commit*. Os tipos utilizados pelos autores, são: inserido (I), atualizado (U) e removido (R);
- O *nome* (representado por **NomeInstrucao**) atribuído a instrução (construção da linguagem) que foi modificada no *commit*. Esse nome pode ser:
 - **falsey**: que representa todas as variáveis que podem ser convertidas para booleano;
 - **typeof**: considerado pelos autores como sendo operadores de igualdade (`===`) ou desigualdade (`!==`) do JavaScript;
 - **callback**: representa as funções de retorno das chamadas;

- **error**: representa os erros que são capturados.

Com essa estrutura é possível representar tipos de mudanças mais complexas, como os tipos de alterações compostas de um ou mais BCTs.

Os BCTs são extraídos a partir de uma lista de subprogramas ($\mathbb{P}$)¹ obtida através da mineração de código (HANAM; BRITO; MESBAH, 2016). O algoritmo de extração de BCTs (Algoritmo 1) tem como saída um conjunto $\mathbb{D}$ de pares (*commit*, BCTs). Neste algoritmo, para cada *commit* (c) no histórico de *commits* ($\mathbb{C}$) de um projeto, é obtido o conjunto de todos os arquivos de código fonte modificados no *commit*. Esse conjunto é chamado de $\mathbb{F} := \{\langle f_{b1}, f_{r1} \rangle \dots \langle f_{bn}, f_{rn} \rangle\}$ e contém n pares (arquivo *defeituoso*, arquivo *reparado*) (Algoritmo 1, Linha 8). Para cada par em $\mathbb{F}$, são calculados os BCTs, criando o AST_b e AST_r , um para o arquivo defeituoso (f_b) e outro para o arquivo reparado (f_r), que incluem apenas as partes diferentes entre ambos os arquivos (Algoritmo 1, Linha 10), isto é feito conforme a abordagem de Falleri et al. (2014). A identificação das mudanças entre f_b e f_r com a diferenciação dentre as ASTs, é uma abordagem mais exata do que a diferença em nível de linha (HANAM; BRITO; MESBAH, 2016).

Algoritmo 1 Extração BCTs

```

1: Input:  $\mathbb{P}$  (lista de subprogramas)
2: Output:  $\mathbb{D}$  (commits e BCTs)
3:  $\mathbb{D} \leftarrow \emptyset$ 
4: for all  $p \in \mathbb{P}$  do
5:    $\mathbb{C} \leftarrow \text{Commits}(p)$ ;
6:   for all  $c \in \mathbb{C}$  do
7:      $\mathbb{T} \leftarrow \emptyset$ ;
8:      $\mathbb{F} \leftarrow \text{ArquivosModificados}(c)$ ;
9:     for all  $\langle f_b, f_r \rangle \in \mathbb{F}$  do
10:       $\langle AST_b, AST_r \rangle \leftarrow \text{ASTDasDiferencas}(f_b, f_r)$ ;
11:       $\mathbb{M} \leftarrow \text{FuncoesModificadas}(AST_b, AST_r)$ ;
12:      for all  $\langle m_b, m_r \rangle \in \mathbb{M}$  do
13:         $\mathbb{T} \leftarrow \mathbb{T} \cup \text{ExtrairBCTs}(m_b, m_r)$ ; ▷ BCTs
14:    $\mathbb{D} \leftarrow \mathbb{D} \cup \langle c, \mathbb{T} \rangle$ ;
```

Fonte: HANAM; BRITO; MESBAH (2016)

Para cada par $\langle AST_b, AST_r \rangle$, é extraído o conjunto de funções que ocorrem em AST_b e AST_r , omitindo funções que são inseridas ou removidas pois representam refatorações e não correções de defeitos. O resultado da extração é um conjunto $\mathbb{M} \leftarrow$

¹ Nesta seção, o termo *subprograma* é utilizado para referenciar arquivos que contêm códigos, de um ou mais projetos.

$\{\langle m_{b1}, m_{r1} \rangle \dots \langle m_{bk}, m_{rk} \rangle\}$ de k pares $\langle \text{função defeituosa}, \text{função reparada} \rangle$ (Algoritmo 1, Linha 11).

Para cada par em $\mathbb{M}$, são extraídos BCTs e incluídos no conjunto $\mathbb{T}$, cujos nós são marcados pela ferramenta de diferenciação com uma das *instruções de modificação* (inserido, atualizado ou removido) que foi indicada no *commit* (Algoritmo 1, Linha 13). Cada *commit* c tem um relacionamento um-para-muitos entre ele e os BCTs encontrados dentro dele (isto é, um *commit* c contém zero ou mais BCTs) e é guardado no conjunto $\mathbb{D}$ (Algoritmo 1, Linha 14).

Uma vez que o conjunto $\mathbb{D}$ é obtido, são filtrados os ruídos dos candidatos selecionados. Apenas são mantidos os *commits* candidatos que: (i) têm entre uma e seis instruções modificadas; (ii) não contêm o texto “*merge*” na mensagem de confirmação do *commit*; (iii) contêm pelo menos um BCT que não tenha o *contexto de construção de linguagem* do tipo *declaração*; e (iv) têm pelo menos um BCT que não tenha o *tipo de modificação* marcada como *atualizado*.

A Tabela 8 mostra um exemplo com três *commits* e algumas informações como o projeto ao qual pertence, o identificador do *commit*, o número de *instruções modificadas* e uma lista de BCTs (no formato [TipoConstrucao]_[ContextoConstrucao]_[TipoModificao]_[NomeInstrucao]) junto com o número de vezes que cada BCT ocorre no *commit* entre colchetes.

Tabela 8 – Conjunto de dados de exemplo com três *commits*. Cada *commit* tem o nome do projeto ao qual ele pertence, o identificador do *commit*, o número de instruções modificadas no *commit* e uma lista dos BCTs e o número de vezes que cada um deles acontece no *commit*

Projeto	<i>Id Commit</i>	Instruções Modificadas	BCTs
async	63b	1	B_C_I_falsey [1] B_C_R_falsey [0] M_MC_I_bind [0]
bower	2d1	4	B_C_I_falsey [1] B_C_R_falsey [2] M_MC_I_bind [0]
express	5c4	2	B_C_I_falsey [0] B_C_R_falsey [0] M_MC_I_bind [1]

Fonte: HANAM; BRITO; MESBAH (2016)

A matriz de características dos dados tem n linhas (uma linha para cada *commit*) e k colunas (uma coluna para a quantidade de vezes que cada BCT aparece no *commit*).

A Figura 6 ilustra um *commit* e os BCTs encontrados pelo módulo de extração de características. Na parte (i) da Figura 6 é mostrada a mensagem de confirmação do *commit* que inclui a palavra *fixed*. Na parte (ii) é mostrado o código com as mudanças feitas em destaque: na linha 765 foi incluída *key* na condição do comando *if*. Para essa linha em particular foi identificado o BCT *RESERVED.CONDITION.INSERTED.global.falsey*. Na parte (iii) da Figura 6 são mostradas as informações do BCT encontrado nesse *commit*. Na parte (iv) da Figura 6 é mostrado o vetor de características de tamanho $k = 3$ correspondente a esse *commit*.

A Figura 7 ilustra um outro exemplo de *commit* e os BCTs encontrados pelo módulo de extração de características. Na parte (i) da Figura 7 é mostrada a mensagem de confirmação do *commit* que não inclui uma palavra que represente defeito ou correção. Na parte (ii) é mostrado o código com as mudanças feitas em destaque, por exemplo, na linha 115 foi substituído o operador `==` pelo operador `===`. Para essa linha em particular foram identificados dois BCTs, um *RESERVED.CONDITION.REMOVED.global.eq* e um *RESERVED.CONDITION.INSERTED.global.sheq*. Na parte (iii) da Figura 7 são mostradas as informações de todos os BCTs encontrados nesse *commit*, que foram de três tipos. Na parte (iv) da Figura 7 é mostrado o vetor de características de tamanho $k = 5$ correspondente a esse *commit*.

Nesta dissertação o módulo da ferramenta BugAID encarregado da etapa de extração de mudanças básicas é chamado de BugAIDExtract (BE).

5.1.3 Clusterização

A partir da matriz de características obtida no passo da extração, é realizada a clusterização dos vetores de características utilizando o algoritmo DBScan (ESTER et al., 1996). Esse algoritmo foi escolhido por dois motivos: (i) ser um algoritmo que faz os agrupamentos baseado em densidade; e (ii) não requer o conhecimento a priori do número de *clusters* que devem ser encontrados (HANAM; BRITO; MESBAH, 2016). Nesse algoritmo foi utilizada a distância de Manhattan, pois essa métrica calcula distâncias mais curtas entre *commits* que têm os mesmos BCTs (HANAM; BRITO; MESBAH, 2016).

Com o resultado da clusterização é realizada a classificação dos defeitos encontrados manualmente, priorizando os *clusters* com mais *tipos de alterações*, que contenham cinco

Figura 6 – Exemplo de commit e BCTs encontrados pelo módulo de extração de características BE: (i) mensagem de confirmação do commit; (ii) mudanças realizadas no *commit* em destaque em vermelho e verde; (iii) BCTs encontrados no *commit*; (iv) vetor de características correspondente a esse *commit*.

Figura 7 – Exemplo de commit e BCTs encontrados pelo módulo de extração de características BE: (i) mensagem de confirmação do commit; (ii) mudanças realizadas no *commit* em destaque em vermelho e verde; (iii) BCTs encontrados no *commit*; (iv) vetor de características correspondente a esse *commit*.

ou mais *commits* e são representados por mais de um projeto. Foi utilizado um processo de inspeção sistemática para inferir padrões de defeitos de tipos de mudanças, com base nos defeitos (códigos defeituosos) e reparos (códigos reparados) que inclui:

1. Amostragem: a partir de cada *tipo de modificação*, é selecionado aleatoriamente um *commit* de cada projeto para fazer parte da amostra desse *tipo de modificação*.
2. Resumo: para ajudar os inspetores a entender a mudança, é visualizado o resumo do *commit* que está no GitHub, em um navegador web. Esses resumos contêm as alterações realizadas no *commit* e mensagens dessas alterações. Assim, os inspetores obtêm as descrições dos padrões.
3. Agrupamento: a partir dos resumos, os inspetores os agrupam em padrões de defeitos (códigos defeituosos). Depois, são realizadas comparações de cada *commit* desses grupos com cada grupo existente. Se o resumo do padrão de defeito (código defeituoso) do *commit* coincidir com algum dos grupos de padrões de defeitos existentes, então o *commit* é classificado nesse padrão de defeito. Caso contrário, cria-se um novo grupo de padrões com esse novo defeito.
4. Revisão: uma vez que todos os *commits* na amostra de *tipo de modificação* foram colocados em um grupo, são realizadas iterativamente revisões dos padrões, unindo ou dividindo-os de acordo com suas descrições.

5.2 Resultados

Nos resultados obtidos por Hanam, Brito e Mesbah (2016), quatro projetos não completaram o processo após 24 horas, pois o processo realizado pelo GumTree de análise da diferença entre os códigos defeituosos e códigos reparados não foi concluído. Esses projetos foram omitidos das estatísticas de tempo. Realizou-se buscas no repositório **npm** do Node.js, que resultou em 134 projetos que tinham seus códigos versionados no GitHub. O tempo total de análise foi de 4h32m, com um tempo médio de 6m10s e um tempo mediano de 4s. A ferramenta BugAID descobriu 1.031 BCTs nos 105.133 *commits* extraídos, que depois de filtrar com uma *query* no DataLog e agrupá-los, produziu 219 *clusters* que continham 4.179 *commits*.

Entre os padrões mais importantes e os correspondentes reparos listados por Hanam, Brito e Mesbah (2016) estão:

1. Valores não referenciados (Bg1, Bg2, Bg3 e Bg22): em JavaScript, os valores *null* e *undefined* representam valores não referenciados. Caso alguém tente utilizar uma variável com algum destes valores, pode ocorrer um erro de exceção. Por esse motivo, é uma boa prática verificar se a variável possui informações válidas antes de usá-las. A seguir algumas formas de reparar esse defeito:
 - a) Proteger checando a validade das informações: variáveis que possuem os valores *false*, 0, "", "", *null*, *NaN* e até variável não definida (*undefined*) são consideradas como falsas ao serem verificadas no comando condicional no JavaScript. Por isso, pode-se verificar se a variável é falsa, antes de tentar usá-las;
 - b) Proteger checando se não tem valor: os valores *null* e *undefined* são considerados equivalentes ao serem comparados com `==`. Por essa razão, pode se verificar se a variável tem valor igual a *null*;
 - c) Proteger checando o tipo: verificar, utilizando o comando *typeof*, se o tipo é igual ao tipo esperado.
2. Comparação incorreta: em JavaScript é possível comparar por tipo booleano (!), pelo valor apenas (`==` ou `!=`) ou pelo valor e tipo (`===` ou `!==`). Por essa razão, é necessário utilizar o operador correto. Os possíveis reparos são:
 - a) Comparar menos tipos: no momento de comparar valores em JavaScript é necessário verificar o tipo do valor comparado. Por exemplo, se for comparar `0 == "0"` o resultado vai ser verdadeiro, porque os operadores `==` e `!=` comparam apenas o valor e não o tipo. Nesse caso, o correto seria utilizar o operador que verifica valor e tipo, `===` ou `!==`;
 - b) Comparar mais tipos: em alguns casos quando não se sabe o valor ao certo que será comparado, convém utilizar a comparação de igualdade apenas pelo valor, isto é, `==` e `!=`.
3. Argumento ausente (Bg4): acionar uma função sem passar os argumentos que deveriam ter sido informados. Um possível reparo é passar *null* para os argumentos faltantes.
4. Configuração incorreta de API: a configuração das informações sobre ambiente, o software, banco e dados e outras ao utilizar APIs ou bibliotecas de terceiros não foi realizada de maneira correta ou não foi especificada.

5. Operador *this* não vinculado corretamente ao escopo esperado: a palavra reservada *this* é um operador que auxilia na obtenção do objeto ou função que o acionou. Por exemplo, pode ocorrer o acionamento do *this* em uma função filha, mas esperando obter a função pai. Os possíveis reparos são:
 - a) Criar uma propriedade que armazene o escopo esperado do *this*: criar uma variável e colocar o escopo esperado do *this*, assim o problema de acessar o *this* em um escopo diferente é reparado;
 - b) Construir *this* em objeto: em JavaScript existem algumas funções para construir de forma explícita *this* nos objetos e que podem ser utilizadas para reparo. Essas funções são: *call*, *bind* e *apply*.
6. Exceção não tratada (Bg6, Bg11 e Bg12): não utilizar os comandos *try-catch* em operações assíncronas ou síncronas, nas quais podem existir insegurança sobre o trecho de código que será executado, acarretando em um erro não esperado. Os possíveis reparos são:
 - a) Capturar exceções lançadas: quando a exceção ocorre, ela é lançada para a função que acionou. Assim, uma possível correção é envolver essa linha de código nos comandos *try-catch* e tratar a exceção lançada;
 - b) Exceção não propagada para a função que acionou *callback*: Pode ocorrer erros de exceção dentro das funções *callback* ou funções de promessa (*promises*). Por isso, é necessário que seja implementado o retorno nas funções para que os erros de exceções possam ser propagados para a função que originou o acionamento da função de retorno. Por exemplo, se uma função com o nome calculadora for acionar uma função *callback* chamada soma, a calculadora terá que receber a exceção que poderá ter ocorrido dentro da função soma, por isso é necessário retornar a exceção;
 - c) Verificar erro de exceção na função *callback*: é recomendado que retorne ou propague o erro de exceção ocorrido dentro da função *callback* ou da função *promise* e que seja verificado o retorno, tratando-o. Seguindo o exemplo anterior, a função calculadora deverá verificar sempre se ocorreu algum erro de exceção na função soma, caso ocorra, terá que tratá-lo.

Note que os padrões de defeitos 1, 3 e 6 correspondem aos defeitos Bg1, Bg2, Bg3, Bg4, Bg6, Bg11, Bg12 e Bg22 da Tabela 4, enquanto os defeitos 2, 4 e 5 não estão identificados e catalogados na RSL. Estes defeitos são identificados como Bg25 (defeito semântico), Bg26 (defeito lógico) e Bg27 (defeito semântico) respectivamente.

Hanam, Brito e Mesbah (2016) usaram uma amostra de 31 *commits* que foram selecionados de maneira aleatória para validar o algoritmo de agrupamento DBScan. Esses *commits* foram retirados dos projetos PM2² e MediacycenterJS³. As métricas utilizadas por Hanam, Brito e Mesbah (2016) para avaliar a ferramenta BugAID que usa o DBScan foram Precisão, Cobertura (*Recall*) e Medida F (*F1-Score*). Porém, essas métricas não são usadas comumente para avaliar algoritmos de aprendizado de máquina não supervisionado.

5.3 A ferramenta BugAID

Hanam, Brito e Mesbah (2016) implementaram essa técnica em uma ferramenta chamada BugAID, desenvolvida em Java e disponível online⁴. Os históricos dos projetos no GitHub são explorados pela ferramenta JGit⁵, um cliente do Git⁶ para Java. As ASTs do JavaScript, são criadas e exploradas usando uma versão da ferramenta Mozilla Rhino. As ASTs das diferenças entre o arquivo defeituoso (f_b) e o arquivo reparado (f_r), são geradas usando uma versão da ferramenta GumTree (FALLERI et al., 2014), que realiza essa diferença em granularidade baixa e que foi modificada em (HANAM; BRITO; MESBAH, 2016). Para fazer a clusterização foi utilizada a ferramenta Weka (HALL et al., 2009). Para auxiliar na inspeção manual dos tipos de alterações, foi implementado um *script* que seleciona aleatoriamente os *commits* de um tipo de alteração e exibe o resumo do GitHub no navegador web.

O primeiro passo para executar o BugAID é realizar o *download* das aplicações e extrair as características dos *commits*. Esse passo é realizado através do projeto BugAID, executando o comando do Maven⁷ a seguir:

² PM2 (Process Manager 2) é um gerenciador de processos para aplicações Node.js.

³ MediacycenterJS é um aplicativo de mídia que é executado no navegador.

⁴ <http://salt.ece.ubc.ca/software/bugaid/>

⁵ <https://eclipse.org/jgit/>

⁶ <https://git-scm.com/>

⁷ Maven é uma ferramenta de automação e gerenciamento de projetos Java disponível em <https://maven.apache.org/>

```
mvn exec:java -Dexec.mainClass="ca.ubc.ece.salt.pangor.js.learn.LearningAnalysisMain" -Dexec.args=
"--repositories ./input/javascript_repositories.txt --regex \"fix\" --dataset ./output/dataset.csv"
```

Nesse comando, a classe `LearningAnalysisMain` é acionada passando três parâmetros de entrada, que são:

1. **repositories**: o arquivo texto que contém a lista de URLs das aplicações que a ferramenta BugAID deve baixar do GitHub, neste exemplo foi informado o arquivo `javascript_repositories.txt`.
2. **regex**: a expressão regular utilizada na busca das mensagens nos *commits*, neste exemplo foram procuradas mensagens que continham a palavra `fix`;
3. **dataset**: o arquivo da base de dados em que serão gravadas as informações dos *commits* obtidos em cada repositório informado no primeiro parâmetro, isto é, o *dataset* é o arquivo em que será gravada a matriz de características dos dados. Neste exemplo, as informações foram gravadas no arquivo *csv* chamado `dataset`.

Depois de obter a matriz de características dos dados, o próximo passo é executar o algoritmo de clusterização para encontrar os *clusters* dos padrões existentes. Para isso, é executado no Maven o comando:

```
mvn exec:java -Dexec.mainClass="ca.ubc.ece.salt.pangor.learn.LearningDataSetMain" -Dexec.args=
"--dataset ./output/dataset.csv --complexity 6 --epsilon 0.3 --complexityWeight 0.2 --minClusterSize 5"
```

Esse comando aciona a classe `LearningDataSetMain` passando cinco parâmetros de entrada, sendo eles:

1. **dataset**: o arquivo que contém a matriz de características dos dados que foram obtidos a partir de cada repositório. Neste exemplo, foi passado o arquivo `dataset.csv`.
2. **complexity**: o número máximo de instruções modificadas. Neste exemplo, foi informado o número 6.
3. **epsilon**: o tamanho do raio que delimita a busca dos objetos vizinhos dentro do mesmo *cluster*, esse parâmetro é usado pelo algoritmo DBScan. Segundo Hanam, Brito e Mesbah (2016), o melhor valor para esse parâmetro é 0,3.
4. **complexityWeight**: peso da complexidade das características dentro dos *clusters*.
5. **minClusterSize**: o número mínimo de pontos necessários para formar uma região densa, esse parâmetro é usado pelo algoritmo DBScan.

Após executar essa última etapa, o BugAID informa o resultado do agrupamento realizado pelo DBScan, conforme visto na Figura 8. Na primeira linha é possível identificar que o *cluster* identificado como 16 está formado por 750 *commits* que incluem o BCT (RE_C_I_falsey) e esses 750 *commits* estão dentro de 71 projetos.

Figura 8 – Os três primeiros *clusters* obtidos pela ferramenta BugAID.

```
1 {RESERVED:CONDITION:INSERTED:global:falsey:1.0}: ClusterID = 16, Instances = 750, Complexity = 1, BasicChanges  
= 1, ProjectCount = 71  
2 {RESERVED:CONDITION:INSERTED:global:falsey:2.0}: ClusterID = 40, Instances = 188, Complexity = 2, BasicChanges  
= 1, ProjectCount = 47  
3 {RESERVED:CONDITION:REMOVED:global:falsey:1.0}: ClusterID = 18, Instances = 177, Complexity = 2, BasicChanges  
= 1, ProjectCount = 47
```

6 Identificação de deficiências no BugAID e proposta de melhorias

A ferramenta BugAID (HANAM; BRITO; MESBAH, 2016) apresentou deficiências na extração de mudanças para algumas funcionalidades básicas da linguagem JavaScript, como por exemplo, mudanças relacionadas a algumas APIs não foram identificadas. Neste capítulo são apresentados: (i) as deficiências na extração de mudanças e a forma em que foram identificadas, (ii) o módulo de extração de características da ferramenta BugAID melhorado, que é chamado de BugAIDExtract+ (BE+), (iii) a ferramenta JS Advise desenvolvida no mestrado que inclui alguns dos defeitos e *code smells* identificados pelo BugAID; e (iv) as deficiências encontradas na avaliação realizada em (HANAM; BRITO; MESBAH, 2016).

6.1 Identificação das deficiências do módulo BE da ferramenta BugAID

Para identificar as deficiências que o módulo de extração BE do BugAID possui, foi realizada uma análise dos resultados obtidos da extração de mudanças básicas (BCTs) dos projetos PM2 e MediacenterJS¹, totalizando 6.065 *commits*.

Com os resultados da execução do BugAID, primeiro foi realizada a seleção de uma amostra aleatória de 150 *commits*. Depois, foram analisados os BCTs detectados e descobertos pelo BugAID, comparando-os com as mudanças que foram realizadas no *commit*. Desta forma, foi possível identificar que mudanças relacionadas a algumas APIs não foram identificadas pela ferramenta. Essas APIs são:

1. Fetch²: essa API é uma biblioteca JavaScript que é popular para manipulação de requisições ao servidor de forma fácil e assíncrona.
2. Manipulação de *String*³: funções da API de manipulação de texto da linguagem, como por exemplo o *trimEnd*, *trimLeft*, *trimRight* e *toLocaleUpperCase*.
3. Manipulação de Arquivos⁴: funções da API de manipulação de arquivos, como por exemplo o *copyFileSync*, *chown* e *constantscopyFile*;

¹ Ambos projetos foram utilizados por Hanam, Brito e Mesbah (2016) nos seus experimentos.

² <https://developers.google.com/web/ilt/pwa/working-with-the-fetch-api>

³ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

⁴ <https://nodejs.org/api/fs.html>

4. Funções de Matemática⁵: API com funções matemáticas, que é básica em todas as linguagens;
5. Manipulação do DOM⁶: API com funções de manipulação de DOM, para auxiliar na interação com imagens SVG, HTML e XML;
6. Manipulação de Mídia⁷: essa API é para manipulação de vídeos.

Além das APIs listadas acima, na amostra utilizada, palavras reservadas básicas da linguagem não tinham sido detectadas nas mudanças dos *commits*, entre elas:

1. *module*⁸: forma de declarar a importação de outras bibliotecas ao projeto;
2. *let*⁹: declaração de variáveis de contexto local. Por exemplo, limitando o escopo da cláusula de decisão;
3. *await*¹⁰: operador para informar o ponto de espera para acionar a função de forma assíncrona.

Desta forma, foi identificado que o BugAID deixou de detectar aproximadamente 11% de BCTs. Após essa análise dos BCTs, foi realizado um ajuste no BugAID para retornar as mensagens que os usuários descrevem no momento que enviam as alterações do código ao repositório. Essa mensagem é pequena, com aproximadamente 50 caracteres.

O objetivo de analisar essa descrição por *commit*, foi verificar se o BugAID estava conseguindo identificar corretamente a palavra *fix* passada como parâmetro de entrada (descrito na Seção 5.3) para ser buscada nas mensagens. Foram selecionados os mesmos 150 *commits* da amostra utilizada para analisar os BCTs.

Desta forma, foi possível identificar que em apenas 9% dos *commits*, o BugAID havia identificado corretamente a palavra *fix* e rotulado eles como *commits* passíveis de correção.

Segundo Hanam, Brito e Mesbah (2016), no pré-processamento dos *commits* era realizada a omissão de funções que foram inseridas ou removidas, pois acreditavam que estariam representando refatorações e não correções de defeitos. Mas, com a análise das mensagens, foi identificado que mudanças que eram feitas para melhorar o código

⁵ https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Math

⁶ https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

⁷ https://developer.mozilla.org/en-US/docs/Web/API/Media_Session_API

⁸ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>

⁹ <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/let>

¹⁰ <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/await>

(chamadas de refatoração), eram identificadas como outros tipos de mudanças e mesmo assim foram consideradas mudanças realizadas como sendo de correção.

6.2 Abordagem proposta BE+

Com as deficiências identificadas no processo de extração de característica do BugAID, o próximo passo foi realizar as melhorias e ajustes. Uma vez que o código fonte da ferramenta BugAID não estava disponível, foi realizada uma engenharia reversa dos arquivos .class disponíveis no GitHub utilizando a ferramenta JDec¹¹.

A empresa Ecma International através da norma ECMA-262¹² e a documentação do Node.js¹³ fornecem informação de forma periódica das atualizações referente às APIs existentes e às palavras reservadas da linguagem JavaScript. Baseados nesses documentos, foram realizadas as alterações no BugAID para identificar as informações referentes às palavras reservadas e as APIs que a linguagem tem de atual no momento. Entre as APIs completamente novas incluídas estão: FSWatcher, Process, Server, Socket, Net, DNS e Assert. Entre as APIs já existentes no BugAID estão Stats, WriteStream e ReadStream, FileHandle, FS e Path, porém foram incluídas as seguintes palavras reservadas:

- API *Stats*: *dev*, *ino*, *mode*, *nlink*, *uid*, *gid*, *rdev*, *size*, *blksize*, *blocks*, *atimeMs*, *mtimeMs*, *ctimeMs* e *birthtimeMs*;
- APIS *WriteStream* e *ReadStream*: *path*, *close* e *ready*;
- API *FileHandle*: *appendFile*, *chmod*, *chown*, *close*, *datasync*, *fd*, *read*, *readFile*, *stat*, *sync*, *truncate*, *utimes*, *write* e *writeFile*;
- API *FS*: *constantscopyFile*, *copyFileSync*, *fchown*, *fchownSync*, *fdatasync*, *fdatasyncSync*, *mkdtemp* e *mkdtempSync*;
- API *Path*: *toNamespacedPath*.

Foram incluídas também 548 palavras reservadas da linguagem (funções, propriedades, constantes, excessões, extensões e eventos), entre elas estão estão: *await*, *const*, *let*, *module*, *AudioListener*, *cancelAnimationFrame*, *cancelIdleCallback*, *DataView*, *Date*, *DOMRect*, *File*, *FileList*, *FileReader*, *HTMLDocument*, *HTMLInputElement*, *MediaCapabilities*, *MediaDeviceInfo*, *MessageEvent*, *openDatabase*, *PerformanceEntry*, *RangeError*,

¹¹ <http://jdec.herokuapp.com/>

¹² <https://www.ecma-international.org/publications/standards/Ecma-262.htm>

¹³ <https://nodejs.org/api/>

ScriptProcessorNode, *scroll*, *scrollBy*, *search*, *ServiceWorker*, *SourceBuffer*, *SVGElement*, *SyncManager*, *TouchEvent*, *TransformStream*, *USBDevice*, *WebGLBuffer*, *webkitMediaStream*, *WebSocket*, *Worker*, *WebAssembly*, *onmouseout* e *onprogress*.

Além dessa alteração, foi melhorada a forma de verificar as palavras chaves na mensagem do *commit*, para identificar se o *commit* tem um possível defeito ou não. Assim, foram incluídas as palavras *fix*, *fixed*, *fixes*, *bug*, *error*, *hotfix*, *failure*, *fault*, *defect* e *miss* na lista de palavras a serem encontradas nas mensagens. Também, foram acrescentadas as palavras *refactoring*, *refactored*, *refactor*, a fim de identificar os *commits* de refatoração.

6.3 Ferramenta JS Advice

A ferramenta JS Advice realiza a análise estática do código JavaScript detectando alguns defeitos ou *code smells* durante o processo de escrita e sugerindo possíveis correções. JS Advice¹⁴ é uma extensão da IDE Microsoft Visual Studio 2015, utilizando .Net Framework v.4.6, na linguagem C#.

JS Advice compara partes do código escrito (em texto) pelo desenvolvedor, com uma lista predefinida de defeitos e *code smells* que incluem trechos de código que representam padrões de defeitos e *code smells* e possíveis correções. Essa lista predefinida dos padrões de defeitos e *code smells* é extraída de forma manual dos resultados obtidos pela ferramenta BugAID, isto é, não existem nenhuma integração direta entre as duas ferramentas. Atualmente a ferramenta Js Advice inclui uma lista pequena de defeitos e *code smells*, entre eles *Variável inexistente* (Bg2), *Variável morta* (Cs1) e *Argumento morto* (Cs2) catalogados na Tabela 4, além do defeito *Comparação de tipo e valor*, descoberto pela ferramenta BugAID.

O JS Advice utiliza o recurso *Light Bulb* (lâmpada ilustrativa de ideia) para indicar ao desenvolvedor onde está o possível defeito dentro do arquivo JavaScript (*.js*), com a sugestão da possível correção para o problema (ver exemplo na Figura 9). Este recurso é comum nas IDEs, pois é uma forma amigável e intuitiva para indicar uma melhoria no código escrito pelo desenvolvedor. Na Figura 9, o defeito indicado foi a *Comparação de tipo e valor*, sendo que nesse caso em particular, a possível correção é substituir a comparação feita apenas por valores (representada no JavaScript por `==`), pela comparação realizada por tipos e valores (representada por `===`).

¹⁴ Disponível em <https://github.com/MackMendes/JsAdvice-Beta>

Figura 9 – Indicação do código com defeito através do recurso *light bulb* e com a sugestão da possível correção apresentada pela ferramenta JS Advice.

Estão sendo incluídos na ferramenta os padrões mais comuns de defeitos descobertos nesta dissertação de mestrado usando o BugAID com o módulo BE+, bem como suas possíveis correções. Apesar da ferramenta JS Advise não ter sido avaliada, essa ferramenta permite mostrar de forma prática a utilização dos resultados obtidos da aplicação dos algoritmos de agrupamento na descoberta dos padrões de defeitos.

6.4 Deficiências na avaliação do algoritmo de agrupamento utilizado na ferramenta BugAID

Com relação aos experimentos realizados em (HANAM; BRITO; MESBAH, 2016), primeiro, não foram realizadas comparações entre algoritmos de agrupamento a fim de identificar qual seria o melhor para a tarefa de agrupamento dos padrões de defeitos e *code smells*. Segundo, a avaliação dos resultados obtidos do agrupamento dos padrões de defeito foi realizada com métricas comumente utilizadas para avaliar algoritmos de aprendizado de máquina supervisionado.

Com isso, este estudo vem sanar essas deficiências identificadas: (i) aplicando cinco algoritmos de agrupamento nos resultados obtidos do uso do módulo de extração original BE e aplicando os mesmos algoritmos usando o novo módulo proposto BE+; e (ii) avaliando os resultados utilizando as métricas mais utilizadas para avaliar algoritmos de aprendizado não supervisionado. Os resultados dessa avaliação são mostrados no próximo capítulo.

7 Resultados

Neste capítulo são apresentados os resultados obtidos usando as melhorias realizadas pelo módulo BE+ na extração das características dos *commits*, bem como os padrões de defeitos/*code smells* mais comuns encontrados pelo módulo BE+. Também são apresentados os resultados da execução dos algoritmos de agrupamento K-Means, C-Means Fuzzy, DBScan, Optics e HDBScan utilizando a matriz de características encontrada pelo módulo BE e utilizando a nova matriz de características encontrada pelo módulo BE+ em uma amostra do conjunto completo de dados.

7.1 Avaliação da extração de características do BE+

Primeiramente, foi executada a ferramenta BugAID (que inclui o módulo BE) disponível no GitHub¹. Neste repositório, além da ferramenta BugAID, está disponível o arquivo de configuração contendo os endereços dos 134 projetos do GitHub (70 bibliotecas de 64 aplicações) utilizados pelos autores para determinar a quantidade de informações que a ferramenta BugAID conseguiu extrair dos projetos. Todavia, dentro destes 134 projetos, 2 pacotes (body-parser.git e nodemon.git) não estão mais acessíveis para realizar a mineração de *commits* através do JGit²; além disso, 1 projeto (tinderplusplus.git) e 1 pacote (newwww.git) não possuem mais repositórios no GitHub. Por isso, foram analisados 130 projetos com 142.735 *commits*. A ferramenta BugAID com BE+ também foi executada nos mesmos projetos e *commits*. A Tabela 9 mostra os resultados obtidos referentes à extração de características dos *commits*. A segunda coluna apresenta a quantidade total de *commits*, a terceira, quarta, quinta e sexta colunas apresentam a quantidade de *commits* em que foram identificadas as palavras chave D/C/F/E (Defeito/Correção/ Falha/Erro), Merge, Refatoração e outras, respectivamente. Finalmente, a última coluna apresenta a quantidade de BCTs diferentes encontrados.

¹ <https://github.com/saltlab/BugAID>

² <https://www.eclipse.org/jgit/>

Tabela 9 – Resultados da extração de características utilizando os módulos BE e BE+

Módulo	<i>Commits</i>	D/C/F/E	Merge	Refatoração	Outras Mudanças	BCTs
BE	142.735	4	19.761	0	122.970	668
BE+	142.735	28.722	19.761	1.939	92.313	901

O BE+ descobriu 25,8% a mais de BCTs diferentes que o BE nos mesmos *commits* (ver Tabela 9). Isso destaca de forma positiva as implementações e melhorias realizadas no processo de extração de características das mudanças nos *commits*, beneficiando assim o agrupamento mais preciso dos padrões de defeitos. Note que aproximadamente 20% das mensagens dos *commits* possuem palavras relacionadas a Defeito, Correção, Falha e Erro na mensagem, indicando que são possíveis *commits* de correções. Foram encontrados 1.939 *commits* que apresentam a palavra Refatoração, isso indica que poucas modificações são voltadas para melhoria do código existente nos projetos.

A ferramenta BugAID com BE encontrou 549 *clusters* enquanto com BE+ encontrou 553 *clusters*. Para ambos casos, foi usado o algoritmo de agrupamento DBSCAN, mas não houve avaliação dos resultados através de métricas nesse conjunto de dados, pois não temos uma partição de referência C^{ref} para esse conjunto. Na próxima seção, será usada uma amostra para fazer a validação dos resultados.

Os seis padrões de defeitos e *code smells* mais comuns encontrados pelo módulo BE+ são:

1. Valores não referenciados: é o mesmo padrão de defeito identificado por Hanam, Brito e Mesbah (2016), descrito na Seção 5.2. Porém, além dos três reparos (*proteger checando a validade das informações*, *proteger checando se não tem valor* e *proteger checando o tipo*) foi identificado o seguinte reparo que é o mais utilizado pelo desenvolvedores:
 - a) Proteger iniciando as variáveis com valor padrão (BCT: *RESERVED:CONDITION:REMOVED:global:falsey:1.0*): caso uma variável que não tenha valor referenciado (*null* ou *undefined*) seja acessada, um possível reparo é iniciar as variáveis com um valor padrão. Um exemplo de commit com esse reparo é mostrado na Figura 10;
2. Configuração incorreta de API: é o mesmo padrão de defeito encontrado por Hanam, Brito e Mesbah (2016), descrito na Seção 5.2;
3. Modificação de método em tempo de execução (*code smell* BCT: *FIELD:ASSIGNMENT_LHS:INSERTED:global:prototype:1.0*): em JavaScript existem casos em que é necessário criar um método dentro de um objeto ou simplesmente criar uma função imutável. Note que, ao criar uma função dentro de um objeto, alguém mal-intencionado pode sobrescrever a função, se ela não for imutável. Assim, uma boa

Figura 10 – *Commit* com a inclusão do valor padrão para a variável *pkg.author* (linha 188). Note que antes era verificado se não tinha valor referenciado (linha 190), porém essa linha foi removida.

```

185 185          if (prop in props) { pkg[prop] = props[prop]; }
186 186      });
187 187      // Author.
188 188  +      pkg.author = {};
188 189      ['name', 'email', 'url'].forEach(function(prop) {
189 190          if (props['author_' + prop]) {
190 190  -      if (!pkg.author) { pkg.author = {}; }
191 191          pkg.author[prop] = props['author_' + prop];
192 192      }
193 193  });

```

prática é estender o objeto prototipando e criando a função ou método. Na Figura 11 é mostrado um exemplo;

Figura 11 – *Commit* criando o método *isDivisibleBy* a partir da prototipação do objeto *Validator*.

```

693      return this;
694  }
695
696  +      Validator.prototype.isDivisibleBy = function(n) {
697  +          return (parseFloat(this.str) % parseInt(n, 10)) === 0;
698  +      }
699  +
700      Validator.prototype.isFloat = function() {
701          return this.isDecimal();
702      }

```

4. Operador *this* não vinculado corretamente ao escopo esperado: é o mesmo padrão de defeito encontrado em (HANAM; BRITO; MESBAH, 2016), descrito na Seção 5.2;
5. Exceção não tratada: é o mesmo padrão de defeito encontrado por Hanam, Brito e Mesbah (2016), descrito na Seção 5.2;
6. Inconsistência de dados no vetor: em JavaScript, o vetor pode conter valores de vários tipos (inteiro, texto, data, objeto, etc). Porém, essa flexibilidade deve ser aplicada em momentos específicos e de maneira controlada, pois dependendo do caso, pode ser criada uma lista com valores inconsistentes. Um possível reparo é:
 - a) Verificar o tipo da instância (BCT: *METHOD:METHOD_CALL:INSERTED:global:push:1.0*): ao incluir um novo item no vetor, verificar se a instância é do tipo esperado utilizando o comando *instanceof*. Na Figura 12 é mostrado um exemplo. Nas linhas 270-273, o tipo do item é verificado antes de incluir o item *val* no *array*.

Figura 12 – *Commit* alterando o código para verificar o tipo de instância com o comando *instanceof* antes de adicionar um novo item no vetor *array*.

```

268     toArray: function() {
269         return this.reduce([], function(array, val){
270             - array.push(val)
271             + if (val instanceof Collection)
272               + array.push(val.toArray())
273             + else
274               + array.push(val)
275         })
276     },

```

7.2 Validação dos algoritmos de agrupamento

Nesta seção são avaliados os algoritmos de agrupamento K-Means, C-Means Fuzzy, DBScan, Optics e HDBScan, utilizando a matriz de características encontrada pelo módulo BE e utilizando a nova matriz de característica encontrada pelo módulo BE+, para uma amostra do conjunto completo de *commits*.

As métricas utilizadas por (HANAM; BRITO; MESBAH, 2016) para avaliar a ferramenta BugAID que usa o DBScan foram Precisão, Cobertura (*Recall*) e Medida F (*F1-Score*). Porém, elas não são usadas comumente para avaliar algoritmos de aprendizado de máquina não supervisionado. Por essa razão, nesta seção não são usadas essas métricas e sim três índices de validação externa, que são o índice Rand, o índice Jaccard e o índice Rand Ajustado.

Foram selecionados aleatoriamente 25 *commits* para validar os algoritmos de agrupamento. Esses *commits* foram retirados dos 130 projetos. A Tabela 10 mostra o resumo dos resultados da extração de características dessa amostra, que é chamada de A-25.

Tabela 10 – Resultados da extração de características da amostra A-25 utilizando os módulos BE e BE+

Módulo	<i>Commits</i>	D/C/F/E	Merge	Refatoração	Outras Mudanças	BCTs
BE	25	0	0	0	25	5
BE+	25	10	0	0	15	8

A amostra A-25 foi inspecionada manualmente a fim de agrupar os 25 *commits* em *clusters* e validar os resultados da execução dos algoritmos. A Tabela 11 mostra os oito *clusters* identificados, a quantidade de *commits* por *cluster*, os BCTs encontrados, os defeitos/code smell e correções/refatorações associadas. *Code smells* são códigos que podem indicar um problema potencial, porém não são defeitos propriamente ditos.

Note que foram identificados três *clusters* que tem *code smells*. Além disso, foram identificados 4 *commits outliers*, isto é, 16% dos *commits* extraídos não apresentaram defeitos nem *code smells*. Exemplos de *outliers* encontrados são: remover o comando para logar informações do console do navegador (*console.log*) e adicionar um comentário.

Tabela 11 – *Cluster* e quantidade de *commits* por *cluster* considerando a amostra A-25

<i>Cluster</i>	<i>Total de Commits</i>	<i>BCTs</i>	Defeito (D) ou <i>code smell</i> (CS)	Correções ou Refatoração (CS)
P_1	3	<i>FIELD:ASSIGNMENT_LHS:INSERTED: global:self</i>	(CS) Função em escopo local, evitando o reuso	Expor função no escopo global através da variável <i>self</i>
P_2	3	<i>FIELD:CONDITION:INSERTED: global:self</i> <i>RESERVED:CONDITION:INSERTED: global:falsey</i>	(CS e D) Função em escopo local, evitando o reuso e condição com comportamento indesejável.	Expor uma variável no escopo global através da variável <i>self</i> e incluir variável desejável na condição <i>if</i> .
P_3	3	<i>RESERVED:CONDITION:INSERTED: global:falsey</i>	(D) Condição com comportamento indesejável	Incluir variável desejável na condição <i>if</i> .
P_4	3	<i>METHOD:METHOD_CALL:INSERTED: global:close</i>	(D) Não acionar o método para fechar arquivos ou de conexão com o banco de dados, assim bloqueando a utilização do arquivo ou o banco de dados.	Acionar o método <i>close</i> para fechar o arquivo ou o banco de dados.
P_5	3	<i>RESERVED:ARGUMENT:INSERTED: global:error</i>	(D) Não informar a função de erro (<i>console.error</i>) o erro gerado.	Passar como parâmetro de entrada (argumento) o erro gerado para a função de erro (<i>console.error</i>).
P_6	3	<i>RESERVED:ARGUMENT:INSERTED: global:true</i>	(D) Função não recebe parâmetro booleano	Passar para a função o parâmetro booleano
P_7	2	<i>METHOD:METHOD_CALL:INSERTED: global:test</i>	(D) Falta de teste unitário de funcionalidade.	Realizar o teste unitário e acionar a função de teste (<i>test</i>)
P_8	1	<i>RESERVED:ARGUMENT:REMOVED: global:arguments</i>	(CS) Parâmetro (argumento) não utilizado na função	Remover o parâmetro (argumento) da função
<i>outliers</i>	4	N/A	N/A	N/A

Foram utilizados os seguintes pacotes da linguagem R³ que implementam os algoritmos de agrupamento avaliados nesta seção: (i) o pacote K-Means Clustering; (ii) o pacote e1071 com a função Fuzzy C-Means Clustering; e (iii) o pacote DBSCAN, que contém os três algoritmos de densidade DBScan, Optics e HDBScan.

Uma vez que os algoritmos K-Means e C-means Fuzzy usam inicialmente centróides que são gerados aleatoriamente, para cada quantidade de centróides, foram realizadas 10 execuções e calculados a média, mediana e desvio padrão destas 10 execuções.

Já os algoritmos DBScan, Optics e HDBScan foram executados apenas uma vez para cada uma das configurações dos parâmetros, pois eles são algoritmos baseados em densidade e não envolvem nenhum elemento aleatório, isto é, eles devolvem sempre os mesmos resultados para uma configuração determinada.

³ <https://www.r-project.org/>

7.2.1 Resultados dos algoritmos K-Means e C-Means Fuzzy

Os algoritmos K-Means e C-Means Fuzzy foram executados com $K=C=5$ para o módulo BE e com $K=C=8$ para BE+. Os módulos obtiveram os melhores resultados com esses valores de parâmetros, os quais correspondem ao número de BCTs diferentes encontrados por cada módulo (ver a última coluna da Tabela 10). Além disso, o algoritmo C-Means Fuzzy foi executado usando duas distâncias: a distância Euclidiana e a distância de Manhattan.

Nas Tabelas 12, 13 e 14 são apresentados os resultados usando os índices Rand, Jaccard e Rand Ajustado, respectivamente para a amostra A-25. O algoritmo C-Means Fuzzy com a distância Euclidiana se destaca positivamente, tanto usando o módulo BE, quanto usando o módulo BE+. Além disso, o BE+ foi superior ao BE, considerando os três índices.

Tabela 12 – Índice Rand: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy

Algoritmo	Distância	K=C BE	K=C BE+	Mediana BE	Mediana BE+	Média BE	Média BE+	Desvio Padrão BE	Desvio Padrão BE+
K-Means	Euclidiano	5	8	0,8133	0,9233	0,8420	0,9367	0,0423	0,0194
C-Means Fuzzy	Manhattan	5	8	0,8133	0,9333	0,8323	0,9387	0,0406	0,0178
C-Means Fuzzy	Euclidiano	5	8	0,8517	0,9333	0,8517	0,9387	0,0404	0,0178

Tabela 13 – Índice Jaccard: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy

Algoritmo	Distância	K=C BE	K=C BE+	Mediana BE	Mediana BE+	Média BE	Média BE+	Desvio Padrão BE	Desvio Padrão BE+
K-Means	Euclidiano	5	8	0,2632	0,4651	0,3160	0,5291	0,0744	0,1000
C-Means Fuzzy	Manhattan	5	8	0,2632	0,5178	0,3004	0,5426	0,0692	0,0921
C-Means Fuzzy	Euclidiano	5	8	0,3316	0,5178	0,3316	0,5426	0,0721	0,0921

7.2.2 Resultado do algoritmo DBScan

Foram utilizados os seguintes parâmetros para o DBScan: (i) epsilon, iniciando com 0,1 e terminando em 6,3 com incrementos de 0,2; e (ii) *MinPts* (Mínimo de BCT por grupo) iniciando com 2 e terminando em 20 com incrementos de 1. Por essa razão, o

Tabela 14 – Índice Rand Ajustado: mediana, média e desvio padrão obtidos pelos algoritmos K-Means e C-Means Fuzzy

Algoritmo	Distância	K ou C <i>clusters</i> BE	K ou C <i>clusters</i> BE+	Mediana BE	Mediana BE+	Média BE	Média BE+	Desvio Padrão BE	Desvio Padrão BE+
K-Means	Euclidiano	5	8	0,3347	0,5941	0,4046	0,6534	0,0980	0,0902
C-Means	Manhattan	5	8	0,3347	0,6469	0,3834	0,6668	0,0930	0,0821
Fuzzy									
C-Means	Euclidiano	5	8	0,4259	0,6469	0,4259	0,6668	0,0961	0,0821
Fuzzy									

número total de execuções do algoritmo DBScan foi 608. Na Tabela 7 são apresentados os 5 melhores resultados obtidos das 608 execuções realizadas pelo algoritmo DBScan para a amostra A-25. Diferente dos resultados obtidos por (HANAM; BRITO; MESBAH, 2016), em que foi encontrado que o melhor valor para epsilon foi 0,3, os experimentos nesta seção mostram que os melhores valores para epsilon são 0,1; 0,3; 0,5; 0,7 ou 0,9 e $MinPts=2$. A quantidade de *clusters* encontrada pelo algoritmo DBScan com o módulo BE foi 5, enquanto com o módulo BE+ foi 8. Dos 4 *outliers* existentes na amostra, apenas 1 foi encontrado tanto pelo BE quanto pelo BE+. Novamente, os resultados obtidos pelo DBScan com o módulo BE+ foram melhores que o BE, considerando os três índices.

Tabela 15 – Os 5 melhores resultados obtidos pelo algoritmo DBScan

	Epsilon	MinPts	Total <i>clusters</i> BE	Total <i>clusters</i> BE+	Total Outliers BE	Total Outliers BE+	Rand BE	Rand BE+	Jaccard BE	Jaccard BE+	Rand Ajustado BE	Rand Ajustado BE+
1	0,1	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
2	0,3	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
3	0,5	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
4	0,7	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
5	0,9	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643

7.2.3 Resultado do algoritmo Optics

Uma vez que o algoritmo Optics é similar ao DBScan, foram usados os mesmos valores para os parâmetros Epsilon e $MinPts$. Na Tabela 8 são apresentados os 5 melhores resultados obtidos pelo algoritmo Optics para a amostra A-25. Os resultados obtidos foram idênticos aos obtidos pelo algoritmo DBScan.

Tabela 16 – Os 5 melhores resultados obtidos pelo algoritmo Optics

	Epsilon	MinPts	Total <i>clusters</i> BE	Total <i>clusters</i> BE+	Total Outliers BE	Total Outliers BE+	Rand BE	Rand BE+	Jaccard BE	Jaccard BE+	Rand Ajustado BE	Rand Ajustado BE+
1	0,1	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
2	0,3	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
3	0,5	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
4	0,7	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
5	0,9	2	5	8	1	1	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643

7.2.4 Resultado do algoritmo HDBScan

Como comentado anteriormente, o algoritmo HDBScan é uma implementação do algoritmo DBScan para realizar agrupamentos dos dados de forma hierárquica. Diferentemente dos algoritmos DBScan e Optics, o algoritmo HDBScan não precisa do parâmetro epsilon. Assim, o algoritmo HDBScan foi executado apenas com diferentes valores de *MinPts*, iniciando com 2 *MinPts* e terminando com 20 com incrementos de 1. Na Tabela 9 são apresentados os 5 melhores resultados obtidos pelo algoritmo HDBScan para a amostra A-25. Note que o valor do índice Rand Ajustado está próximo de 0, tanto usando o módulo BE, quanto usando BE+, o que indica que as concordâncias encontradas entre as duas partições foram ao acaso.

Tabela 17 – Os 5 melhores resultados obtidos do algoritmo HDBScan

	<i>MinPts</i>	Total <i>clusters</i> BE	Total <i>clusters</i> BE+	Total Outliers BE	Total Outliers BE+	Rand BE	Rand BE+	Jaccard BE	Jaccard BE+	Rand Ajustado BE	Rand Ajustado BE+
1	4	2	2	0	6	0,4167	0,6300	0,125	0,1527	0,0870	0,1462
2	3	2	2	0	2	0,4167	0,4367	0,125	0,1289	0,0870	0,0946
3	2	2	2	0	0	0,4167	0,2367	0,125	0,0984	0,0870	0,0324
4	5	2	0	0	25	0,4167	0,0833	0,125	0,0833	0,0870	0
5	6	0	0	25	25	0,0833	0,0833	0,125	0,0833	0	0

7.2.5 Resumo da validação dos algoritmos de agrupamento

A Tabela 10 apresenta um resumo com os melhores resultados gerados pelos algoritmos avaliados na amostra A-25. Os algoritmos DBScan e Optics se destacaram positivamente, obtendo os maiores valores para os três índices, tanto com o módulo BE, quanto com o BE+, quando comparados com os outros algoritmos avaliados. Em particular, o Rand Ajustado, usando DBScan e Optics com o módulo BE+, foi superior ao K-Means e ao C-Means Fuzzy (0,7643 contra 0,6534 e 0,6668, respectivamente). Note que na amostra

A-25 existem 4 commits *outliers*, isto é, 16% dos *commits* dessa amostra (Tabela 11) e apenas 1 desses 4 *outliers* foi identificado como *outlier* pelo DBScan e Optics (Tabelas 15 e 16).

Já o HDBScan se destacou negativamente, obtendo valores próximos de zero para o índice Rand Ajustado.

Tabela 18 – Resumo com os melhores resultados obtidos pelos algoritmos avaliados na amostra A-25

Algoritmo	<i>Clusters</i> BE	<i>Clusters</i> BE+	Rand BE	Rand BE+	Jaccard BE	Jaccard BE+	Rand Ajustado BE	Rand Ajustado BE+
K-Means	5	8	0,8420	0,9367	0,3160	0,5291	0,4046	0,6534
C-Means	5	8	0,8517	0,9387	0,3316	0,5426	0,4259	0,6668
DBScan	5	8	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
Optics	5	8	0,8933	0,9633	0,3846	0,6452	0,5013	0,7643
HDBScan	2	2	0,4167	0,6300	0,125	0,1527	0,0870	0,1462

8 Conclusão e trabalhos futuros

Uma ferramenta que implementa uma técnica semi-automática para descobrir padrões de defeitos a partir das mudanças realizadas no repositório GitHub, é a ferramenta BugAID (HANAM; BRITO; MESBAH, 2016). Porém, essa abordagem apresentou deficiências na detecção e descoberta de defeitos associados às *APIs* externas (OCARIZA et al., 2013), manipulação de arquivos, manipulação de textos, funções de cálculo matemático, variáveis de contexto local, entre outras. Para resolver essas deficiências foram realizadas alterações no BugAID para identificar informações referentes às palavras reservadas e às *APIs* que a linguagem tem de atual no momento. Essas melhorias foram implementadas no módulo BE+.

O módulo BE+ descobriu 34,88% a mais de BCTs que o BE. O novo módulo identificou aproximadamente 20% de *commits* que tinham mensagens contendo palavras indicando possíveis correções de defeitos, enquanto o BE identificou menos de 0,003%. Além disso, o módulo BE+ identificou 1.939 *commits* que possuem a palavra *Refatoração* nas suas mensagens, enquanto o BE não encontrou nenhum.

Foi realizado um conjunto de testes utilizando os algoritmos de agrupamento K-Means, C-means Fuzzy, DBScan, Optics e HDBScan nas matrizes de características obtidas pelo BE+ e pelo BE na amostra A-25. O BE+ encontrou três BCTs a mais que o BE. Os algoritmos DBScan e Optics com BE+ apresentaram os melhores resultados para os três índices, enquanto que o HDBScan com BE e com BE+ apresentou o pior resultado.

Assim, o objetivo de resolver as deficiências encontradas na ferramenta BugAID foi alcançado. Dentre os seis padrões de defeitos e *code smells* mais comuns encontrados pelo módulo BE+ e que não foram identificados pelo módulo BE estão o *code smell modificação de método em tempo de execução* e o defeito *inconsistência de dados no vetor*. Esses padrões identificados pelo BE+ e que realmente ocorrem na prática, podem ser incluídos em ferramentas de detecção de defeitos ou *code smells*, o que resultaria na melhoria da qualidade de software escrito na linguagem JavaScript .

8.1 Contribuições

Entre as principais contribuições desta dissertação de mestrado, destacam-se:

- A elaboração de uma revisão sistemática da literatura sobre detecção automática de defeitos em software JavaScript, a qual foi publicada no XIII Simpósio Brasileiro de Sistemas de Informação (SBSI) do ano de 2017;
- O novo módulo de extração de mudanças básicas BE+ que melhora o modelo BE da ferramenta BugAID, sendo que BE+ descobre mais padrões de defeitos que o módulo BE.
- A aplicação de cinco algoritmos de agrupamento nos resultados obtidos a partir da utilização dos módulos BE e BE+;
- A avaliação dos algoritmos de agrupamento K-Means, C-means Fuzzy, DBScan, Optics e HDBScan, na tarefa de descoberta de padrões de defeitos em duas amostras, sendo que foram utilizados índices mais aderentes ao cenário de agrupamento de dados.
- A criação da ferramenta Js Advice¹, que realiza a análise estática do código JavaScript detectando alguns defeitos durante o processo de escrita e sugerindo as possíveis correções. Essa ferramenta (i) utiliza uma lista extraída manualmente dos padrões de defeitos descobertos nesta dissertação de mestrado e suas respectivas correções; e (ii) realizada uma comparação de igualdade do código escrito com algum padrão de defeito.

8.2 Limitações

Dentre as limitações da ferramenta BugAID estão:

- Os módulos BE e BE+ utilizam uma lista genérica de termos e palavras reservadas. Essa lista precisa ser atualizada toda vez que uma bibliotecas ou API nova é criada.
- Para gerar os ASTs, o BugAID utiliza uma versão antiga da ferramenta Mozilla Rhino², a versão 1.7R4 de 2012.
- Nem todos os *commits outliers* encontrados pelos módulos BE e BE+ foram identificados pelos algoritmos de agrupamento como sendo *outliers*. Na amostra A-25 foram 75% que não foram identificados como *outliers* pelos algoritmo DBScan e Optics.
- Nem todos os *clusters* encontrados pelos módulos BE e BE+ correspondem a padrões de defeitos ou *code smells*. Foi selecionada uma amostra de 65 *clusters* dos 549

¹ <https://github.com/MackMendes/JsAdvice-Beta>

² https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Download_Rhino

clusters encontrados pelo módulo BE+ utilizando o DBSCAN. Dos 65 *clusters*, 9,23% não são nem padrões de defeitos nem *code smells*, esses clusters correspondem apenas a alterações pontuais nos códigos.

8.3 *Trabalhos futuros*

A estratégia para descoberta de padrões de defeitos de Hanam, Brito e Mesbah (2016) tem um grande potencial de ser usada para outras linguagens e bibliotecas, uma vez que a descoberta é feita de maneira automática a partir das informações dos códigos enviados aos repositórios dos projetos. Para isso, é necessário implementar um lista genérica de termos e palavras reservadas, a qual é usada durante a etapa de mineração dessa estratégia.

Um outro trabalho futuro é a atualização da versão da ferramenta Mozilla Rhino utilizada pelo BugAID. Com isso será possível obter informação adicional e mais detalhada da estrutura do código dentro dos *commits*, enriquecendo assim as ASTs para geração dos BCTs.

Uma vez que ainda não existe uma integração direta entre a ferramenta BugAID e o Js Advise, um possível trabalho futuro é fazer essa integração. A ferramenta Js Advise pode ser aprimorada com a utilizando de alguma técnica de similaridade para realizar a comparação do código escrito pelo desenvolvedor com os padrões de defeitos e as possíveis correções. Por exemplo, pode ser utilizado o cálculo de similaridade do cosseno (STEINBACH et al., 2000) no código escrito com a matriz de características dos *commits*, os padrões dos defeitos e as possíveis correções. Desta forma, a ferramenta Js Advice, realizaria a detecção de defeitos de forma semi-automática e com capacidade de evoluir com o tempo, pois sempre estaria observando os padrões de defeitos dos projetos do GitHub através do BugAID.

Referências³

- AGRAWAL, H. et al. Fault localization using execution slices and dataflow tests. In: IEEE. *Sixth International Symposium on Software Reliability Engineering*. [S.l.], 1995. p. 143–151. Citado na página 42.
- AGRAWAL, R. et al. *Automatic subspace clustering of high dimensional data for data mining applications*. [S.l.]: ACM, 1998. v. 27. Citado na página 21.
- ANDREASEN, E.; MØLLER, A. Determinacy in static analysis for jQuery. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2014. v. 49, n. 10, p. 17–31. Citado 2 vezes nas páginas 38 e 46.
- ANDREOPOULOS, B. et al. A roadmap of clustering algorithms: finding a match for a biomedical application. *Briefings in Bioinformatics*, Oxford University Press, v. 10, n. 3, p. 297–314, 2009. Citado na página 20.
- ANKERST, M. et al. Optics: ordering points to identify the clustering structure. In: ACM. *ACM Sigmod record*. [S.l.], 1999. v. 28, n. 2, p. 49–60. Citado 3 vezes nas páginas 21, 24 e 25.
- BAE, S. et al. SAFEWAPI: Web API misuse detector for web applications. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 507–517. Citado 2 vezes nas páginas 17 e 38.
- CAMPELLO, R. J.; MOULAVI, D.; SANDER, J. Density-based clustering based on hierarchical density estimates. In: SPRINGER. *Pacific-Asia conference on knowledge discovery and data mining*. [S.l.], 2013. p. 160–172. Citado 2 vezes nas páginas 22 e 26.
- CANTELON, M. et al. *Node.js in Action*. [S.l.]: Manning, 2014. Citado na página 16.
- CHEESEMAN, P. et al. Autoclass: A bayesian classification system. In: *Machine Learning Proceedings 1988*. [S.l.]: Elsevier, 1988. p. 54–64. Citado na página 21.
- EERNISSE, M. *Build Your Own AJAX Web Applications*. [S.l.]: SitePoint Pty Ltd, 2006. Citado 3 vezes nas páginas 30, 39 e 42.
- EICH, B. Javascript at ten years. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2005. v. 40, n. 9, p. 129–129. Citado na página 30.
- ESTER, M. et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In: *Kdd*. [S.l.: s.n.], 1996. v. 96, n. 34, p. 226–231. Citado 5 vezes nas páginas 17, 21, 22, 23 e 52.
- FALLERI, J.-R. et al. Fine-grained and accurate source code differencing. In: ACM. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. [S.l.], 2014. p. 313–324. Citado 2 vezes nas páginas 50 e 58.
- FARD, A. M.; MESBAH, A. JSNOSE: Detecting JavaScript code smells. In: *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM-2013)*. [S.l.: s.n.], 2013. p. 116–125. Citado 3 vezes nas páginas 17, 33 e 38.

³ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

- GONG, L. et al. DLint: Dynamically checking bad coding practices in JavaScript. In: ACM. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. [S.l.], 2015. p. 94–105. Citado na página 33.
- GUARNIERI, S.; LIVSHITS, V. B. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In: *USENIX Security Symposium*. [S.l.: s.n.], 2009. v. 10, p. 78–85. Citado na página 43.
- HALKIDI, M.; BATISTAKIS, Y.; VAZIRGIANNIS, M. On clustering validation techniques. *Journal of intelligent information systems*, Springer, v. 17, n. 2-3, p. 107–145, 2001. Citado na página 27.
- HALL, M. et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, ACM, v. 11, n. 1, p. 10–18, 2009. Citado na página 58.
- HANAM, Q.; BRITO, F. S. d. M.; MESBAH, A. Discovering bug patterns in JavaScript. In: ACM. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2016. p. 144–156. Citado 22 vezes nas páginas 16, 17, 18, 19, 30, 37, 48, 50, 51, 52, 55, 58, 59, 61, 62, 65, 67, 68, 69, 72, 75 e 77.
- HARTIGAN, J. A. Clustering algorithms. Wiley, 1975. Citado na página 20.
- HATHAWAY, R. J.; BEZDEK, J. C. Recent convergence results for the fuzzy c-means clustering algorithms. *Journal of Classification*, Springer, v. 5, n. 2, p. 237–247, 1988. Citado 2 vezes nas páginas 21 e 22.
- HEITKÖTTER, H.; HANSCHKE, S.; MAJCHRZAK, T. A. Evaluating cross-platform development approaches for mobile applications. In: SPRINGER. *International Conference on Web Information Systems and Technologies*. [S.l.], 2012. p. 120–138. Citado 2 vezes nas páginas 16 e 31.
- HSIAO, C.-H.; CAFARELLA, M.; NARAYANASAMY, S. Using web corpus statistics for program analysis. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2014. v. 49, n. 10, p. 49–65. Citado na página 38.
- HUBERT, L.; ARABIE, P. Comparing partitions. *Journal of classification*, Springer, v. 2, n. 1, p. 193–218, 1985. Citado na página 29.
- JAIN, A. K. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, Elsevier, v. 31, n. 8, p. 651–666, 2010. Citado na página 23.
- JAIN, A. K.; DUBES, R. C. Algorithms for clustering data. Prentice-Hall, Inc., 1988. Citado 3 vezes nas páginas 20, 21 e 27.
- JENSEN, S. H.; MADSEN, M.; MØLLER, A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: ACM. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.], 2011. p. 59–69. Citado na página 38.
- JENSEN, S. H.; MØLLER, A.; THIEMANN, P. Type analysis for JavaScript. In: SPRINGER. *International Static Analysis Symposium*. [S.l.], 2009. p. 238–255. Citado 3 vezes nas páginas 33, 38 e 39.

- JOSHI, B. Overview of html5 and asp. net 4.5. In: *HTML5 Programming for ASP. NET Developers*. [S.l.]: Springer, 2012. p. 1–34. Citado na página 46.
- JR, F. S. O.; PATTABIRAMAN, K.; MESBAH, A. AutofloX: An automatic fault localizer for client-side JavaScript. In: IEEE. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST-2012)*. [S.l.], 2012. p. 31–40. Citado 2 vezes nas páginas 17 e 38.
- JR, F. S. O.; PATTABIRAMAN, K.; MESBAH, A. Vejovis: Suggesting fixes for JavaScript faults. In: ACM. *Proceedings of the 36th International Conference on Software Engineering*. [S.l.], 2014. p. 837–847. Citado na página 38.
- KASHYAP, V. et al. JSAI: A static analysis platform for JavaScript. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. p. 121–132. Citado na página 44.
- KASHYAP, V. et al. Type refinement for static analysis of JavaScript. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2013. v. 49, n. 2, p. 17–26. Citado na página 38.
- KITCHENHAM, B. A. Systematic review in software engineering: Where we are and where we should be going. In: *Proceedings of the 2Nd International Workshop on Evidential Assessment of Software Technologies*. [S.l.]: ACM, 2012. p. 1–2. Citado na página 34.
- KO, A. J.; WOBBROCK, J. O. Cleanroom: Edit-time error detection with the uniqueness heuristic. In: IEEE. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.], 2010. p. 7–14. Citado na página 38.
- KO, A. J.; ZHANG, X. Feedback detects missing feedback in web applications. In: ACM. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [S.l.], 2011. p. 2177–2186. Citado na página 38.
- KOHONEN, T. Exploration of very large databases by self-organizing maps. In: IEEE. *Neural Networks, 1997., International Conference on*. [S.l.], 1997. v. 1, p. PL1–PL6. Citado na página 21.
- LEVENSHTAIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet physics doklady*. [S.l.: s.n.], 1966. v. 10, n. 8, p. 707–710. Citado na página 42.
- LEVY, A. Y.; RAJARAMAN, A.; ORDILLE, J. J. Query-answering algorithms for information agents. In: AAAI. *Thirteenth National Conference on Artificial Intelligence*. [S.l.], 1996. p. 40–47. Citado na página 42.
- LIANG, G. et al. Patbugs: A pattern-based bug detector for cross-platform mobile applications. In: IEEE. *IEEE International Conference on Mobile Services (MS-2014)*. [S.l.], 2014. p. 84–91. Citado 2 vezes nas páginas 17 e 38.
- MACEDO, C. M. D.; DELGADO, K. V.; WIESE, I. S. Detecção automática de falhas em software javascript: Uma revisão sistemática da literatura. In: SBC. *Anais Trilha Principal SBSI 2017*. [S.l.], 2017. p. 412–419. Citado 3 vezes nas páginas 19, 34 e 46.

MACQUEEN, J. et al. Some methods for classification and analysis of multivariate observations. In: OAKLAND, CA, USA. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. [S.l.], 1967. v. 1, n. 14, p. 281–297. Citado 2 vezes nas páginas 21 e 22.

MADSEN, M.; LIVSHITS, B.; FANNING, M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: ACM. *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2013. p. 499–509. Citado na página 38.

MADSEN, M.; TIP, F.; LHOTÁK, O. Static analysis of event-driven node.js JavaScript applications. In: *ACM SIGPLAN Notices*. [S.l.: s.n.], 2015. v. 50, n. 10, p. 505–519. Citado 2 vezes nas páginas 31 e 38.

MESBAH, A.; DEURSEN, A. V.; LENSELINK, S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, ACM, v. 6, n. 1, p. 3, 2012. Citado 2 vezes nas páginas 41 e 42.

OCARIZA, F. et al. An empirical study of client-side JavaScript bugs. In: IEEE. *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2013. p. 55–64. Citado 5 vezes nas páginas 16, 17, 18, 30 e 75.

OCARIZA, F. S. *On the detection, localization and repair of client-side JavaScript faults*. Tese (Doutorado) — University of British Columbia, 2016. Citado na página 17.

OCARIZA, F. S. et al. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 26, n. 1, p. 69–88, 2016. Citado na página 38.

PARK, C. et al. Static analysis of JavaScript web applications in the wild via practical DOM modeling (t). In: IEEE. *30th IEEE/ACM International Conference on Automated Software Engineering (ASE-2015)*. [S.l.], 2015. p. 552–562. Citado 2 vezes nas páginas 17 e 38.

PARK, J. JavaScript API misuse detection by using typescript. In: ACM. *Proceedings of the companion publication of the 13th international conference on Modularity*. [S.l.], 2014. p. 11–12. Citado na página 38.

RADATZ, J.; GERACI, A.; KATKI, F. IEEE standard glossary of software engineering terminology. *IEEE Std*, v. 610121990, n. 121990, p. 3, 1990. Citado na página 31.

RAND, W. M. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, Taylor & Francis Group, v. 66, n. 336, p. 846–850, 1971. Citado na página 28.

REAL, R.; VARGAS, J. M. The probabilistic basis of jaccard's index of similarity. *Systematic biology*, JSTOR, v. 45, n. 3, p. 380–385, 1996. Citado na página 29.

SCHUBERT, E. et al. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, ACM, v. 42, n. 3, p. 19, 2017. Citado na página 24.

- SHAH, G. H.; BHENSADIA, C.; GANATRA, A. P. An empirical evaluation of density-based clustering techniques. *International Journal of Soft Computing and Engineering (IJSCE) ISSN*, Citeseer, v. 22312307, p. 216–223, 2012. Citado na página 25.
- STEINBACH, M. et al. A comparison of document clustering techniques. In: BOSTON. *KDD workshop on text mining*. [S.l.], 2000. v. 400, n. 1, p. 525–526. Citado na página 77.
- THUNG, F. et al. Network structure of social coding in github. In: IEEE. *Software maintenance and reengineering (csmr), 2013 17th european conference on*. [S.l.], 2013. p. 323–326. Citado na página 48.
- XU, R.; WUNSCH, D. Survey of clustering algorithms. *IEEE Transactions on neural networks*, Ieee, v. 16, n. 3, p. 645–678, 2005. Citado na página 20.
- YOUNANG, A.; LU, L. Improving precision of Java Script program analysis with an extended domain of intervals. In: IEEE. *IEEE 39th Annual Computer Software and Applications Conference (COMPSAC-2015)*. [S.l.], 2015. v. 3, p. 441–446. Citado na página 38.
- ZHENG, Y.; BAO, T.; ZHANG, X. Statically locating web application bugs caused by asynchronous calls. In: ACM. *Proceedings of the 20th International Conference on World Wide Web*. [S.l.], 2011. p. 805–814. Citado na página 38.