Discovering code smells in Javascript software using clustering techniques

Luan M. Sales[1], **Charles M. de Macedo**[1], **Karina V. Delgado**[1]

[1] Escola de Artes Ciências e Humanidades - Universidade de São Paulo (USP)
Av. Arlindo Béttio, 1000 – 03828-000 – São Paulo – SP – Brazil

luan.sales@usp.br, charlesmendes@usp.br, kvd@usp.br

Abstract. *The presence of code smells in software projects has negative consequences with respect to the code's cohesion and maintainability. Therefore, the analysis of techniques used to discover and detect code smells automatically is an increasingly explored topic. A semi-automatic tool which allows to discover bug patterns and eventual code smells in JavaScript code is the BugAID. The purpose of this work was to contribute with the BugAID tool in the task of discovering code smells which are common in the development of JavaScript software by improving word identification associated with refactored code in commit messages and implementing the BE++ module. The BE++ module proved effectiveness in identifying code smells which involve small code changes discovering 5 common code smells within the refactor group. These code smells are candidates for inclusion in code smells detection tools to prevent issues in JavaScript software development.*

Resumo. *A presença de code smells em projetos de software têm consequências negativas no que diz respeito a coesão e manutenibilidade do código. Assim sendo, a análise de técnicas usadas para descoberta e detecção de code smells de maneira automática é um tópico cada vez mais explorado. Uma ferramenta semi-automática que permite descobrir padrões de defeitos e eventuais code smells em código JavaScript é a BugAID. O objetivo deste trabalho foi contribuir com a ferramenta BugAID na tarefa de descoberta de code smells comuns no desenvolvimento de software JavaScript através da melhoria na identificação de palavras associadas a código refatorado nas mensagens dos commits e com a implementação do módulo BE++. O módulo BE++ mostrou-se eficaz na identificação de code smells que envolvem pequenas alterações no código, descobrindo 5 code smells comuns dentro do grupo de refatorações. Esses code smells são candidatos à inclusão em ferramentas de detecção de code smells para prevenção de problemas no desenvolvimento de software JavaScript.*

1. Introdução

A linguagem de programação JavaScript, interpretada e fundamentada na execução dinâmica de scripts, é a mais popular no que concerne ao desenvolvimento de aplicações Web 2.0. A popularização dessa linguagem por parte de empresas e desenvolvedores fomentou a construção de novas soluções, inclusive para o contexto de servidores com ambientes de tempo de execução altamente escaláveis como o Node.js[1].

[1] https://nodejs.org/.

A estrutura flexível e fracamente tipada da linguagem JavaScript é determinante nos problemas enfrentados por seus desenvolvedores. Entre as possíveis causas de problemas estão: (i) o caráter interpretado da linguagem que dificulta verificações em tempo de compilação; (ii) a complexa interação com elementos estruturais do DOM (*Document Object Model*) que dificulta a programação; e (iii) alguns componentes da linguagem como *prototypes*, funções de primeira classe e *closures* são utilizados de maneira errônea pelos programadores [FARD and MESBAH 2013].

Esses aspectos do JavaScript são os principais motivos de ”maus cheiros de código”, também conhecidos como *code smells*. *Code smells* são elementos estruturais do software que sinalizam possíveis problemas de modelagem ou no código [Fowler 1999], tornando a sua evolução e manutenibilidade difíceis, motivando ciclos de refatoração.

As técnicas que realizam de forma sistemática a descoberta de *code smells* são fundamentais, pois os resultados obtidos por essas técnicas auxiliam (i) na identificação de quais *code smells* que ocorrem de fato no ciclo de desenvolvimento de projetos de software; (ii) na complementação para ferramentas estáticas de detecção de *code smells*; e (iii) na conscientização dos desenvolvedores a respeito dos *code smells* mais comuns.

Uma ferramenta que implementa técnicas para descoberta de padrões de defeitos em software JavaScript de maneira semiautomática e sistemática é a BugAID [Hanam et al. 2016]. A abordagem da ferramenta BugAID pode ser sumarizada em três grandes etapas: (i) mineração dos *commits* presentes em repositórios de projetos baseados em JavaScript; (ii) extração das mudanças realizadas nos *commits* em abstrações da linguagem JavaScript; e (iii) agrupamento das abstrações utilizando o algoritmo DBSCAN.

Em Macedo et. al (2019) foi proposto um novo módulo para extração de mudanças chamado de BugAIDExtract+ (BE+) com o objetivo de aprimorar, complementar e sanar alguns problemas encontrados na ferramenta BugAID. O módulo BE+ encontrou, além de padrões de defeitos, alguns *code smells* durante as avaliações realizadas em uma amostra. Contudo, não houveram análises a respeito da capacidade de descoberta de *code smells*.

O presente trabalho tem como objetivo descobrir *code smells* que são comuns no desenvolvimento de software JavaScript, para tal, é proposto neste trabalho o módulo denominado BE++, uma complementação do módulo BE+ para descoberta de *code smells* em código refatorado.

O trabalho está estruturado da seguinte forma: na Seção 2 é apresentada a ferramenta BugAID; na Seção 3 são descritas as melhorias desenvolvidas em Macedo et. al (2019) com o módulo BE+; na Seção 4 é proposta uma estratégia para a melhoria na identificação de *commits* de refatoração para a ferramenta BugAID; na Seção 5 é apresentado o módulo BE ++; na Seção 6 são apresentadas as avaliações do módulo BE++ e os resultados obtidos; e na Seção 7 são apresentadas as conclusões do presente trabalho.

2. A Ferramenta BugAID

A ferramenta BugAID [Hanam et al. 2016] implementa uma estratégia semiautomática para descoberta de padrões de defeitos comuns em software JavaScript. Essa ferramenta foi desenvolvida utilizando a linguagem de programação Java.

A abordagem utilizada pela ferramenta BugAID pode ser dividida em três etapas: (i) mineração dos *commits* de repositórios do GitHub, especificamente de projetos *server-*

side desenvolvidos em JavaScript ; (ii) extração das mudanças realizadas no código fonte dos *commits* minerados em abstrações denominadas BCTs (*Basic Change Types*); e (iii) agrupamento dos BCTs encontrados considerando o número de instruções modificadas.

2.1. Mineração dos *Commits*

A abordagem de Hanam et al. (2016) tem como objetivo identificar os padrões de defeitos mais recorrentes em 134 repositórios do GitHub de aplicações *server-side* desenvolvidas em JavaScript. Segundo Hanam et al. (2016) as mudanças que geralmente caracterizam correções de defeitos possuem entre 1 e 6 instruções de modificação (inserção, remoção ou atualização). A identificação da categoria dos *commits* na etapa de mineração do BugAID ocorre baseado em suas mensagens, ou seja, é fundamentada na percepção dos desenvolvedores. As categorias possíveis são *Bug Fix* (correção de defeitos), *Merge* (união de *commits* em *branchs*) e *Other* (outras mudanças).

2.2. Extração de BCTs

As mudanças capturadas na etapa de mineração são extraídas através da diferenciação do código fonte em árvores abstratas de sintaxe da linguagem JavaScript. A partir dessas árvores são construídas abstrações dos componentes estruturais da linguagem chamadas de BCTs (*Basic Change Types*) que identificam como as instruções foram modificadas. Os componentes representativos que formam os BCTs estão sumarizadas na Tabela 1.

Tabela 1. Propriedades dos BCTs

Identificador	**Definições**	**Abstrações**
Tipo de construção da linguagem	Tipo de artefato ou conceito sendo modificado, isto é, palavras reservadas, artefatos de API, tipos de instruções, etc.	Comportamento (B). Classe (CL), Campo (F), Método (M), Parâmetro (P), Reservado (RE), Variável (V), etc
Contexto de construção da linguagem	O contexto em que os elementos da linguagem aparecem. Por exemplo, a palavra reservada *this* pode aparecer em uma cláusula condicional ou como argumento de um método	Argumento (A), Declaração de Classe (CD), Condição (C), Chamada de Método (MC), Declaração de Método (MD), Instrução (S), etc
Tipo de Modificação	Como os elementos da linguagem foram modificados	Inserção (I), Remoção (R), Atualização (U)
Nome	O nome associado ao elemento da linguagem	*undefined, equal, return, var, callback, typeof, this, etc*

A Figura 1 representa o processo de extração de características de um *commit*. As linhas em vermelho no código correspondem as mudanças realizadas, no caso foram realizadas remoções nestes trechos de código (-). As estruturas destacadas em negrito no centro da Figura 1 correspondem aos BCTs encontrados. Esses BCTs indicam que ocorreram a remoção de atribuição de um módulo e de uma variável sem uso. Os BCTs representam as características das mudanças realizadas que são utilizadas na etapa de agrupamento. A quantidade de características corresponde ao total de BCTs encontrados

nos repositórios analisados, de forma que se forem descobertos k BCTs distintos entre si, o vetor de características terá k BCTs como atributos, no caso exemplificado na Figura 1 $k = 4$.

Figura 1. Processo de extração de características de um *commit*

2.3. Agrupamento de Padrões de Defeitos

A última etapa do BugAID corresponde ao agrupamento de BCTs e número de instruções modificadas semelhantes utilizando o algoritmo de agrupamento DBSCAN com o objetivo de descobrir padrões de defeitos recorrentes. Os principais motivos [Hanam et al. 2016] para a escolha do DBSCAN foram: (i) ser um algoritmo baseado em densidade, adequado ao vetor de características proposto; e (ii) não precisar especificar o número de agrupamentos como entrada.

3. A abordagem BE+

A abordagem BugAIDExtract+ (BE+), proposta em Macedo et. al (2019), corresponde à melhorias na ferramenta BugAID na tarefa de identificação de APIs e palavras reservadas presentes atualmente na linguagem JavaScript. Além disso, foram realizadas melhorias na categorização dos *commits* que indicam correções de defeitos e definida uma estratégia para identificação dos *commits* de refatoração de código. Em relação a refatoração, a abordagem BE+ adicionou a categoria *Refactoring* para *commits* minerados que contenham em suas mensagens as palavras *refactor*, *refactoring* ou *refactored*.

4. Estratégia de Identificação de Commits de Refatoração

A categoria de *commits* que interessam para o presente trabalho são os relacionados a refatoração. A decisão desta escolha fundamentou-se nas seguintes características em relação a *code smells* em projetos de *software*:

1. *Code Smells* são elementos estruturais específicos no código que podem causar problemas eventualmente [Fowler 1999].
2. *Code smells* são uma forma de débito técnico [MacCormack and Sturtevant 2016]. Os débitos consistem de más práticas de programação, rotinas de testes inadequadas, ausência de documentação e arquiteturas excessivamente interdependentes.
3. Os estudos a respeito de correção de *code smells* estão intrisicamente associados a ciclos de refatoração de código [Fowler 1999].

4.1. Descrição da Estratégia

Em Macedo et. al (2019) o número de commits que correspondem a categoria de refatorações são aproximadamente 1% do total minerado. Dado que o presente trabalho utiliza a população dessa categoria para avaliação foi considerada uma estratégia para melhorar a identificação de *commits* de refatoração. Essa estratégia, tem como objetivo identificar palavras similares as do conjunto C = {*refactor, refactoring, refactored*} através da realização de uma análise manual das palavras contidas nas mensagens dos *commits*. Porém, após a mineração preliminar nos 128 repositórios utilizados no presente trabalho foram capturadas 287.094 linhas de texto. Por tal motivo, foi estabelecida uma estratégia para a redução do espaço de análise textual, considerando a necessidade de identificação de palavras de interesse. Essa estratégia deve: (i) utilizar técnicas que capturem grupos de palavras fortemente associadas; (ii) lidar com similaridades semânticas e/ou sintáticas entre as palavras, e (iii) utilizar o contexto das mensagens que os desenvolvedores fornecem nos *commits*.

A estratégia desenvolvida neste trabalho foi baseada no algoritmo de aprendizado de máquina não supervisionado Propagação de Afinidade [Frey and Dueck 2007] com similaridades geradas a partir de métricas de casamento de padrões. Esse algoritmo é bastante utilizado para agrupamento de imagens faciais, detecção de genes e reconhecimento de palavras e sentenças representativas em textos. Além disso, o algoritmo não precisa receber o número de agrupamentos a priori.

O algoritmo de Propagação de Afinidade recebe como entrada de dados uma coleção de valores reais de similaridades entre os pontos de dados, de forma que a semelhança $s(i, k)$ indica o grau em que o ponto com índice k é adequado para ser o *representante* do ponto i. Os pontos de dados podem ser vistos como uma rede em que todos os pontos se comunicam enviando mensagens uns aos outros. O conteúdo dessas mensagens informam o grau de adequação de um ponto para se tornar representante em um agrupamento, sendo que os representantes são os pontos mais significativos em seus agrupamentos e explicam da melhor forma os outros pontos no mesmo agrupamento. As mensagens são trocadas iterativamente até a convergência, momento em que os representantes finais são escolhidos e os agrupamentos determinados. Além dos valores reais de similaridade, o algoritmo Propagação de Afinidade recebe os seguintes valores de entrada: (i) a iteração de convergência c determinada pelo número de iterações em que não houve alteração no número de agrupamentos estimados que interrompem a convergência; (ii) fator de amortecimento λ utilizado para a estabilização numérica e como taxa de aprendizado para convergência do algoritmo; e (iii) a preferência ρ que corresponde a uma matriz com os valores de propensão (pesos) dos pontos para se tornarem representantes.

Uma vez que o algoritmo Propagação de Afinidade recebe como entrada uma

matriz de similaridades pré-computadas, foram selecionados os seguintes algoritmos de casamento de padrões textual para a criação das matrizes numéricas das palavras:

1. *Hamming* [Pfeifer et al. 1996]: corresponde ao número mínimo de substituições necessárias para alterar uma cadeia de caracteres em outra.
2. *Levenshtein* [Christen 2006]: corresponde ao menor número de operações de inserções, remoções e substituições para transformar uma *String* em outra.
3. *Jaro-Winkler* [Christen 2006]: corresponde ao número s de carácteres comuns (concordância dos carácteres que estão dentro da metade do comprimento da *String* mais longa), número de transposições t para transformar uma *String* em outra e concordância dos caracteres iniciais.
4. *Word2Vec - Continuos Bag of Words* [Mikolov et al. 2013]: algoritmo de construção de modelos com redes neurais. A entrada é um grande *corpus* de texto e a saída um espaço vetorial, tipicamente de várias centenas de dimensões, com cada palavra única no *corpus* atribuída a um vetor numérico correspondente no espaço. Os vetores são posicionados no espaço vetorial de modo que as palavras que compartilham contextos comuns estejam localizadas próximas umas das outras no espaço. A arquitetura *Continuos Bag of Words* prevê a palavra atual a partir de uma janela de palavras de contexto adjacentes, contudo a ordem das palavras não influenciam na previsão.

4.2. Implementação da Estratégia

A primeira etapa recebe como entrada 128 dos 134 repositórios do GitHub utilizados em [Hanam et al. 2016], dado que os 6 repositórios restantes não estavam disponíveis publicamente. Para cada *commit* nos repositórios são capturados os *corpus* das mensagens. A partir desses *corpus* é criada uma tabela de frequências de todas as palavras que contenham sufixos comuns na língua inglesa que indiquem ação, estado ou qualidade [Carroll JB 1971], por exemplo o sufixo *"ing"* indica verbo (ação) no gerúndio. A saída dessa etapa são os *corpus* das mensagens e a tabela de frequências das palavras.

A segunda etapa da estratégia está descrita no Algoritmo 1. Nessa etapa são utilizados como entrada uma das métricas de similaridades e um algoritmo de agrupamento, além disso são fornecidos os *corpus* das mensagens dos *commits* e a tabela de frequência construídas na primeira etapa. No método FiltrarPalavras as palavras são pré-processadas e são mantidas aquelas que possuem frequência > 30. Se a métrica escolhida for o *Word2Vec - Continuos Bag of Words (CBOW)*, a matriz de similaridades utiliza como contexto para construção do modelo os *corpus* das mensagens dos *commits*, caso contrário será gerada uma matriz $n \times n$ das n palavras com a métrica de similaridade especificada. Vale ressaltar que, para o *Word2Vec* foram definidos os seguintes parâmetros para a construção do modelo: (i) dimensão do vetor numérico igual a 100; (ii) janela de palavras consideradas à direita e à esquerda igual a 5; e (iii) *threshold* inferior das palavras ignoradas igual a 30. Por fim, no método RealizarAgrupamento o algoritmo de agrupamento é executado utilizando a matriz construída nos passos anteriores.

4.3. Avaliação da Estratégia

A validação da estratégia ocorreu com uma amostra aleatória de tamanho 30 das 6483 palavras capturadas na tabela de frequências. As 30 palavras foram agrupadas manualmente baseadas nas suas relações sintáticas e semânticas dentro do contexto das mensagens dos

Algorithm 1: Agrupamento de Palavras Similares

Input: $\mathbb{F}$ (Tabela de frequência das palavras), $\mathbb{M}$ (Mensagens dos commits), *m* (Métrica de Similaridade), *a* (Algoritmo de Agrupamento)
Output: $\mathbb{A}$ (Agrupamentos resultantes)
$\mathbb{S} = \emptyset$ (Matriz de Similaridades);
$\mathbb{P}$ = FiltrarPalavras($\mathbb{F}$);
if *m* == *CBOW* **then**
  $\mathbb{S}$ = CriarModeloComWord2Vec($\mathbb{M}$);
else
  $\mathbb{S}$ = CalcularMatrizDeSimilaridades(*m*, $\mathbb{P}$);
$\mathbb{A}$ = RealizarAgrupamento($\mathbb{S}$, $\mathbb{P}$, *a*);

commits. A avaliação foi realizada utilizando índices de validação externa, tais índices comparam a qualidade dos agrupamentos formados a partir de um resultado conhecido externamente (conjunto de referência) [Aggarwal and Reddy 2013]. Os índices utilizados para validação da estratégia foram [Rosenberg and Hirschberg 2007]: (i) Homogeneidade - medida que quantifica se os agrupamentos contêm apenas pontos de dados que são membros de uma única classe; (ii) Completude - medida que quantifica se todos os pontos de dados que são membros de uma determinada classe são elementos do mesmo agrupamento; (iii) *V-Score* - a média harmônica da homogeneidade e da completude; e (iv) Rand Ajustado - medida de similaridade, ajustada para a aleatoriedade, de dois agrupamentos considerando todos os pares e as quantidades que estão nos mesmos agrupamentos ou em agrupamentos diferentes. Todos esses índices variam entre 0 e 1, com valores próximos a 0 representando baixo grau de concordância entre o conjunto de referência e os agrupamentos resultantes, em contrapartida, valores próximos a 1 representam alto grau de concordância.

O algoritmo foi executado com valores para o fator de amortecimento (λ) partindo de 0,5 até 0,9 com incrementos de 0,1 [Frey and Dueck 2007], iterações de convergência (c) partindo de 1 até 15 com incrementos de 1 e preferência (ρ) pré-definida como a mediana da matriz de entrada dos dados. Esse valor de preferência foi considerado assumindo desconhecimento a respeito das relações entre os pontos de dados. Os melhores resultados para o algoritmo Propagação de Afinidade foram obtidos com fator de amortecimento (λ) igual a 0,5 e iterações de convergência (c) igual a 3 para todas as métricas de construção de similaridades (Tabela 2).

Tabela 2. Melhores resultados do algoritmo Propagação de Afinidade para as métricas de similaridade com λ = 0,5 e c = 3

Métrica	**Total de Iterações**	**Total de *Clusters***	**Rand Ajustado**	**Homogeneidade**	**Completude**	**V−Score**
Hamming	7	6	0.1105	0.4829	0.5694	0.5226
Levenshtein	7	7	0.1441	0.5088	0.5676	0.5366
Jaro-Winkler	5	8	0.1194	0.5122	0.5606	0.5353
CBOW	**9**	**7**	**0.7305**	**0.8111**	**0.9327**	**0.8677**

Considerando os resultados da Tabela 2, a estratégia foi executada com a matriz de similaridades gerada a partir do algoritmo *Word2Vec* com *Continuos Bag of Words (CBOW)* utilizando o algoritmo de Propagação de Afinidade, com as entradas definidas pela validação descrita na presente seção, no conjunto completo de 6483 palavras. A inspeção manual dos 53 agrupamentos formados permitiu a escolha de palavras candidatas a partir do agrupamento de número 36. As palavras candidatas correspondem ao conjunto R = {*'refactor', 'refactoring', refactored', 'normalize', 'adjusted', 'reworked', 'organize', 'reworking'*}, o qual será utilizado pela abordagem BE++, descrita na próxima seção.

5. A Abordagem BE++

O presente estudo têm como objetivo descobrir *code smells* comuns em JavaScript utilizando a ferramenta BugAID. Para tal, foi proposto o módulo BE++ construído como uma complementação do módulo BE+ [de MACEDO 2019]. O módulo BE++ fundamenta-se nas alterações necessárias para melhorar a descoberta de *code smells* baseado na sua correlação com ciclos de refatoração. Tais alterações ocorreram tanto na etapa de mineração de dados quanto na etapa de agrupamento.

5.1. Complementação na Identificação dos Commits

O BugAID é executado a partir de argumentos de linha de comando, contudo na abordagem BE+ as palavras que identificam as categorias dos commits estão *hard coded*. Para utilizar as palavras do conjunto R obtidas na Subseção 4.3 foi adicionada a funcionalidade para fornecer as palavras da categoria *Refactoring* como uma opção em linha de comando.

5.2. Mudanças no Pré-processamento da Etapa de Agrupamento

No módulo BE+ [de MACEDO 2019], durante a etapa de pré-processamento, foram removidos *commits* minerados que possuem exclusivamente BCTs com contexto do tipo *STATEMENT* ou modificação do tipo *UPDATED*, porém as ocorrências destes tipos podem indicar *code smells*. Por exemplo, um *commit* que possua unicamente o BCT, *RESERVED-STATEMENT-REMOVED-global-var-1*, pode indicar a correção de um *code smell* conhecido como variável morta. Portanto, para os *commits* da categoria *Refactoring*, exclusivamente, não foram aplicados os filtros para a remoção desses BCTs nos cenários previamente descritos.

6. Resultados

A presente seção apresenta as avaliações e resultados obtidos com o módulo BE++ proposto.

6.1. Avaliação do Módulo BE++

A execução dos módulos BE+ e BE++ em 128 repositórios públicos do GitHub foi realizada com seus respectivos vetores de palavras. A quantidade de *commits* minerados e identificados como pertencentes à categoria de refatoração (Tabela 3) evidenciam um ganho de 30.9% com a utilização do módulo BE++.

Tabela 3. Resultado da mineração e identificação de *commits* utilizando os módulos BE+ e BE++

Vetor de Palavras	Número de Commits	Correções de Defeitos	Refatorações	Merge	Outras Mudanças
BE+	154374	31336	**2329**	20592	100117
BE++	154374	31336	**3049**	20592	99381

6.2. Avaliação dos Algoritmos de Agrupamento

Macedo et. al (2019) realizaram avaliações de algoritmos para validação do módulo de extração. Na presente seção foi utilizada a mesma abordagem para verificar se os *commits* da categoria *Refactoring* do módulo BE++ estão em concordância com os resultados obtidos por Macedo et. al (2019).

Foram selecionados aleatoriamente 30 dos 3049 *commits* da categoria *Refactoring* extraídos com o módulo BE++ para a avaliação de 5 algoritmos de agrupamento. Os 30 *commits* foram agrupados manualmente baseados em suas mudanças. No conjunto foram identificados 10 *outliers*, isto é, *commits* que não são correções de defeitos ou *code smells*. Os 20 *commits* restantes, com seus respectivos BCTs, foram agrupados manualmente em 7 grupos baseados em suas mudanças para a correção de defeitos ou *code smells*. Além disso, essa amostra possui um total de 13 BCTs distintos como características que foram utilizadas pelos algoritmos de agrupamentos.

Os algoritmos de agrupamentos utilizados foram [Aggarwal and Reddy 2013]: (i) DBSCAN - algoritmo de agrupamento baseado na distribuição de densidade dos pontos considerando pontos mínimos (*minPts*) e raio (*epsilon*), esse algoritmo é capaz de identificar ruídos e não exige que a quantidade de agrupamentos seja fornecida a priori; (ii) K-Means - algoritmo de particionamento que recebe como entrada a quantidade de partições K e de forma iterativa minimiza a distância dos pontos candidatos aos centróides, resultando em K agrupamentos; (iii) C-Means Fuzzy - algoritmo de particionamento similar ao K-Means, porém os pontos possuem grau de pertencimento a todos os agrupamentos resultantes; (iv) OPTICS - algoritmo de agrupamento baseado na distribuição de densidade dos pontos, similar ao DBSCAN, entretanto consegue detectar agrupamentos significativos em dados de densidade variável ordenando os pontos de forma que os mais próximos se tornem vizinhos na ordenação; e (v) HDBSCAN - transforma o algoritmo DBSCAN em um algoritmo de agrupamento hierárquico extraindo agrupamentos baseado em suas estabilidades.

Os índices de validação externa utilizados para avaliação dos algoritmos foram, respectivamente: (i) Rand Ajustado (descrito na Subseção 4.3); e (ii) Jaccard - mede a similaridade entre conjuntos finitos através da divisão do tamanho da interseção pelo tamanho da união dos conjuntos. Além disso, foi utilizado o índice de validação interna Coeficiente de Silhueta. Os índices de validação interna são utilizados para medir a qualidade dos agrupamentos formados sem a utilização de informações externas [Aggarwal and Reddy 2013]. O Coeficiente de Silhueta mede o quanto um ponto é similar ao agrupamento que o contém em relação aos outros agrupamentos formados. Esse coeficiente varia entre -1 e +1, de forma que um alto valor indica que os pontos estão em

concordância com seus agrupamentos. A distância Euclidiana foi utilizada para o cálculo do Coeficiente de Silhueta.

Os algoritmos K-Means e C-Means Fuzzy foram executados com valores de K partindo de 2 até 13 com incrementos de 1, utilizando a distância Euclidiana para o cálculo das distâncias entre os pontos. Dado que esses algoritmos utilizam inicialmente centróides aleatórios foram realizadas 10 execuções e calculados média, mediana e desvio padrão para todos os valores de K.

Os algoritmos DBSCAN e OPTICS foram executados com valores de *epsilon* partindo de 0,1 até 6,3 com incrementos de 0,2 e número mínimo de pontos (*minPts*) partindo de 2 até 20 com incrementos de 1. O algoritmo HDBSCAN foi executado também com número mínimo de pontos (*minPts*) partindo de 2 até 20 com incrementos de 1.

Tabela 4. Resumo dos melhores resultados obtidos pelos algoritmos avaliados na amostra de tamanho 30

Algoritmo	**Total de *clusters***	**Ruídos**	**Jaccard**	**Rand Ajustado**	**Coeficiente de Silhueta**
K-Means	12	0	0.0707	0.0992	0.0243
C-Means Fuzzy	7	0	0.0728	0.0832	0.0492
DBSCAN	**5**	**12**	**0.7961**	**0.8563**	**0.4301**
OPTICS	**5**	**12**	**0.7961**	**0.8563**	**0.4301**
HDBSCAN	6	4	0.4086	0.4842	0.2576

Os melhores resultados para os algoritmos estão descritos na Tabela 4. Os algoritmos DBSCAN e OPTICS, assim como no estudo de Macedo et. al (2019), obtiveram os melhores valores para os três índices avaliados. O melhor valor de *epsilon* para os algoritmos DBSCAN e OPTICS foi 0,3 com número mínimo de pontos (*minPts*) igual a 2. Os resultados dos algoritmos K-Means e C-Means Fuzzy correspondem as médias das 10 execuções realizadas para cada um deles, esses algoritmos apresentaram os piores resultados para todos os índices de validação com valores próximos a 0. Uma justificativa para os resultados dos algoritmos K-Means e C-Means Fuzzy foi a variabilidade no número de instruções modificadas dessa amostra em particular.

6.3. Code Smells Comuns Encontrados

A etapa de agrupamento foi executada utilizando os 3049 *commits* da categoria de refatoração minerados pelo módulo BE++. Os parâmetros de entrada para o DBSCAN foram baseados no estudo original [Hanam et al. 2016] e nos resultados obtidos na seção 6.2: (i) foram considerados *commits* que possuem entre 1 e 6 instruções modificadas; (ii) *epsilon* igual a 0,3; e (iii) número mínimo de pontos (*minPts*) igual a 2. A execução resultou em 30 agrupamentos que foram analisados manualmente. Essa análise permitiu identificar 5 *code smells* comuns em 8 dos 30 agrupamentos formados. Os 22 agrupamentos restantes correspondem a outras refatorações (alteração de nome de variáveis, adição de comentários, etc) e correções de defeitos (comparação em condicionais incorretas, *this* em escopo incorreto, etc). A seguir são descritos os *code smells* descobertos e o número de agrupamentos que os representam.

Variável Morta (4 *Clusters*) A variável não está sendo utilizada em nenhum momento do fluxo do código. Um exemplo da correção desse *code smell* está representado a seguir:

```
1: - var View = exports = function View(view, options) {
2: + function View(view, options) {
```

Retorno Morto (1 *Cluster*) O retorno da função não indica mais um caminho válido no fluxo do código. Um exemplo da correção desse *code smell* está representado a seguir:

```
1: - return value;
2: +
```

Atribuição em Condicionais (1 *Cluster*) Os desenvolvedores utilizam atribuições em cláusulas condicionais para escrever menos código [Saboury et al. 2017]. Um exemplo da correção desse *code smell* está representado a seguir:

```
1: - var charset;
2: - if (charset = mime.charsets.lookup(type)) {
3: + var charset = mime.charsets.lookup(type);
4: + if (charset) {
```

Código Complexo (1 *Cluster*) A complexidade ciclomática de um código é o número de caminhos linearmente independentes através do código [Saboury et al. 2017]. No exemplo a seguir ocorreu a diminuição da complexidade ciclomática removendo uma condicional encadeada para o escopo mais externo:

```
1: + clone = jQuery.support.html5Clone || !rnoshimcache.test( ≪"+ elem.nodeName ) ?
   elem.cloneNode( true ) :  shimCloneNode( elem );
2: if ((!jQuery.support.noCloneEvent || !jQuery.support.noCloneChecked)) {
3:      - if ( rnoshimcache.test( ≪"+ elem.nodeName ) ) clone = shimCloneNode( elem );
```

Código Duplicado (1 *Cluster*) Trecho de Código que ocorre sem necessidade no mesmo projeto mais de uma vez. No exemplo a seguir, as funções *increaseSubtitleOffset* e *decreaseSubtitleOffset* possuiam o mesmo fluxo exceto pelo valor de *offset* de modo que foi extraído um método comum *adjustSubtitleOffset* que recebe os parâmetros adequados:

```
1: - _this.increaseSubtitleOffset();
2: - _this.decreaseSubtitleOffset();
3: + _this.adjustSubtitleOffset(0.1);
4: + _this.adjustSubtitleOffset(-0.1);
```

7. Conclusão

O estudo proposto foi uma complementação da abordagem BE+ desenvolvida em Macedo et. al (2019). O conjunto de palavras do módulo BE+ conseguiu identificar 2329 *commits* da categoria refatoração, em contrapartida, o conjunto de palavras do módulo BE++, proposto no presente trabalho, identificou 3049 *commits*. A estratégia de identificação dos *commits* da categoria *Refactoring* obteve um ganho de 30.9%.

A avaliação dos algoritmos K-Means, C-Means Fuzzy, DBSCAN, OPTICS e HDBSCAN com o módulo BE++ demonstrou concordância com os resultados obtidos em Macedo et. al (2019). Os algoritmos DBSCAN e OPTICS, com *epsilon* igual a 0,3 e pontos mínimos (*minPts*) igual a 2, apresentaram os melhores resultados para os três

índices utilizados. Os algoritmos K-Means e C-Means Fuzzy obtiveram os piores resultados nas avaliações, demonstrando inadequação na tarefa de agrupamento de BCTs.

A ferramenta BugAID com o módulo BE++ se mostrou efetiva na identificação de *code smells* que envolvem pequenas alterações no código, descobrindo 5 *code smells* comuns dentro do grupo de refatorações, evidenciando o potencial do BugAID nesta tarefa. Esses *code smells* podem ser incluídos em ferramentas de detecção estática de *code smells* para prevenção de problemas no desenvolvimento de software JavaScript.

Referências

Aggarwal, C. C. and Reddy, C. K. (2013). *Data Clustering: Algorithms and Applications*. Chapman & Hall/CRC, 1st edition.

Carroll JB, D. P. . R. B. e. (1971). The american heritage word frequency book. *New York: American, Heritage Publishing Co.*

Christen, P. (2006). A comparison of personal name matching: Techniques and practical issues. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pages 290–294.

de MACEDO, C. M. (2019). Aplicação de algoritmos de agrupamento para descoberta de padrões de defeito em software javascript. Master's thesis, Dissertação (Mestrado em Sistemas de Informação) - Escola de Artes, Ciências e Humanidades, University of São Paulo, São Paulo, 2018.

FARD, A. M. and MESBAH, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Frey, B. J. and Dueck, D. (2007). Clustering by passing messages between data points. *Science*, 315(5814):972–976.

Hanam, Q., Brito, F. S. d. M., and Mesbah, A. (2016). Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 144–156, New York, NY, USA. ACM.

MacCormack, A. and Sturtevant, D. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, 120.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

Pfeifer, U., Poersch, T., and Fuhr, N. (1996). Retrieval effectiveness of proper name search methods. *Inf. Process. Manage.*, 32(6):667–679.

Rosenberg, A. and Hirschberg, J. (2007). V-measure: A conditional entropy-based external cluster evaluation measure. pages 410–420.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305.