Recomendação de Conhecimento Disponível em Sítios Q&A para Auxílio ao Desenvolvimento e Depuração de *Software*

Eduardo Cunha Campos

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

Uberlândia
2019

Eduardo Cunha Campos

Recomendação de Conhecimento Disponível em Sítios Q&A para Auxílio ao Desenvolvimento e Depuração de *Software*

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

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2019

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

C198r
2019

Campos, Eduardo Cunha, 1988-
Recomendação de conhecimento disponível em sítios Q&A para auxílio ao desenvolvimento e depuração de software [recurso eletrônico] / Eduardo Cunha Campos. - 2019.

Orientador: Marcelo de Almeida Maia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://dx.doi.org/10.14393/ufu.te.2019.924
Inclui bibliografia.

1. Computação. 2. Software - Fatores humanos. 3. Software - Desenvolvimento. 4. Interface de programação de aplicações. I. Maia, Marcelo de Almeida, 1963-, (Orient.) II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Gloria Aparecida - CRB-6/2047

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da defesa de TESE DE DOUTORADO junto ao Programa de Pós-graduação em Ciência da Computação da Faculdade de Computação da Universidade Federal de Uberlândia.

Defesa de Tese de Doutorado: PPGCO-04/2019

Data: 12 de abril de 2019 Hora de início: 13:30

Discente: Eduardo Cunha Campos Matrícula: 11513CCP008

Título do Trabalho: Recomendação de Conhecimento Disponível em Sítios Q&A para Auxílio ao Desenvolvimento e Depuração de *Software*

Área de concentração: Ciência da Computação

Linha de pesquisa: Engenharia de Software

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação assim composta: Professores doutores: Henrique Coelho Fernandes – FACOM/UFU, Rivalino Matias Júnior – FACOM/UFU, André Cavalcante Hora – DCC/UFMG, Fernando José Castor de Lima - CIN/UFPE e Marcelo de Almeida Maia - FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. André Cavalcante Hora participou da defesa por meio de videoconferência desde a cidade de Belo Horizonte – MG, o Prof. Dr. Fernando José Castor de Lima Filho da cidade de Recife – PE e o Prof. Dr. Henrique Coelho Fernandes da cidade de Saarbrücken, Alemanha. Os outros membros da banca e o aluno participaram *in loco*

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

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

Em face do resultado obtido, a Banca Examinadora considerou o candidato **aprovado**.

Esta defesa de Tese de Doutorado é parte dos requisitos necessários à obtenção do título de Doutor. O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, legislação e regulamentação interna da Universidade Federal de Uberlândia.

Nada mais havendo a tratar foram encerrados os trabalhos às 17 horas e 25 minutos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Participou por meio de vídeo conferência

Prof. Dr. André Cavalcante Hora
DCC/UFMG

Participou por meio de vídeo conferência

Prof. Dr. Fernando José Castor de Lima Filho
CIN/UFPE

Participou por meio de vídeo conferência

Prof. Dr. Henrique Coelho Fernandes
FACOM/UFU

Prof. Dr. Rivalino Matias Júnior
FACOM/UFU

Prof. Dr. Marcelo de Almeida Maia
FACOM/UFU (Orientador)

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Agradeço...

A Deus, por minha vida e pela minha saúde.

Aos meus pais Olavo Luiz de Sousa Campos e Rose Mary Machado Cunha pelo amor incondicional, carinho, apoio, dedicação e perseverança em todos os momentos. Obrigado mãe por ter sido a fortaleza para a gente ter chegado até aqui. Sem o seu apoio, esse sonho não seria possível.

Às minhas irmãs Tatiana Cunha Campos e Izabella Cunha Campos pelos conselhos, incentivos, carinho e amor durante esta etapa da minha vida.

À minha namorada Lycea Maria Maciel Nogueira por ser a minha maior incentivadora a encarar este desafio e pela compreensão durante este projeto. Você me deu muita força, tranquilidade, paz, alegria e amor.

A todos os professores da Faculdade de Computação (FACOM) que contribuíram no meu processo de graduação e pós-graduação na Universidade Federal de Uberlândia.

Aos meus amigos pelo incentivo e carinho:
Antônia Maria Maciel,
Ana Myriam Maciel Reis,
Fellipe Martins Lamoglia,
Flávio Marques,
Henrique De Villa Alves,
Lucas Batista Leite de Souza,
Luis Felipe Nogueira,
Raphael Ferreira Vieira,
Ricardo Luiz de Sousa Campos,
Roberto César Rodrigues.

A CAPES, FAPEMIG, CNPq pelo apoio financeiro.

Principalmente ao professor Marcelo de Almeida Maia pelo profissionalismo, comprometimento, paciência e orientação em todos os momentos da realização deste trabalho.

Groups are remarkably intelligent and are often better than the smartest person in them.
(James Surowiecki: The Wisdom of Crowds)

Resumo

O desenvolvimento moderno de programas é inseparável do uso das Interfaces de Programação de Aplicativos (APIs). No entanto, vários estudos mostraram que aprender e lembrar como usar as APIs é difícil para os desenvolvedores de *software* devido à documentação inadequada de algumas APIs.

Os últimos anos testemunharam o surgimento e a crescente popularidade de sítios de mídias sociais relacionados ao desenvolvimento de programas, como o *Stack Overflow*, o *DaniWeb* e o *Quora*. A informação disponível nesses sítios é uma tendência importante no suporte de atividades relacionadas ao desenvolvimento e depuração de programas.

Para lidar com o problema da introdução de erros relacionados com o uso incorreto da API por parte do desenvolvedor, foi proposta uma abordagem que recomenda *posts* do *Stack Overflow* que podem conter a correção destes erros. Todavia, esta abordagem recebe como entrada um trecho de código suspeito de conter um erro. Para avaliar esta abordagem, foi construído um *benchmark* contendo 30 trechos de código com potenciais *API-usage-related bugs* escritos nas linguagens Java e JavaScript extraídos do *site Open Hub Code Search*. Os resultados de recomendação mostraram que foram encontrados nos top-10 *posts* recomendados a correção para 66,67% dos trechos de código Java e 40% dos trechos de código JavaScript presentes no *benchmark*. Além disso, esta abordagem superou os motores de busca *Google* e *FaCoY* na recomendação de correções para esta categoria de erros de *software*.

Para auxiliar os desenvolvedores durante alguma tarefa de programação com API, foi proposta uma outra abordagem denominada *Lucene+Score+How-to* que recomenda somente pares Q&A do *Stack Overflow* pertencentes à categoria *How-to* a partir de uma consulta feita em linguagem natural pelo usuário. Para avaliar esta abordagem, foram conduzidos experimentos envolvendo 35 tarefas de programação extraídas aleatoriamente de livros de receitas de 3 tópicos amplamente utilizados pela comunidade de desenvolvimento de *software*: *Swing* (Java), *Boost* (C++) e *LINQ* (C#). Os resultados de recomendação desta abordagem mostraram que, para 77,14% das tarefas de programação analisadas, pelo menos um par Q&A recomendado provou ser útil para a solução da tarefa alvo.

Palavras-chave: Sabedoria da multidão. Depuração com a multidão. Tarefas de uso da API.

Abstract

Modern-day software development is inseparable from the use of the Application Programming Interfaces (APIs). However, several studies have shown that learning and remembering how to use APIs is difficult for software developers due to inadequate documentation of some APIs.

Recent years have witnessed the emergence and growing popularity of social media sites related to software development, such as Stack Overflow, DaniWeb and Quora. The information available on these sites is one important trend in supporting activities related to software development and debugging.

In order to address the problem of introducing errors related to incorrect use of the API by the developer, an approach has been proposed which recommends posts from Stack Overflow that may contain the correction of these errors. However, this approach receives as input a code snippet suspected of containing an error. To evaluate this approach, a benchmark was constructed containing 30 code excerpts with potential API-usage-related bugs written in the Java and JavaScript programming languages extracted from the Open Hub Code Search site. The recommendation results showed that 66.67% of Java excerpts with potential API-usage-related bugs had their fixes found in the top-10 query results. Considering JavaScript excerpts, fixes were found in the top-10 results for 40% of them. Moreover, this approach outperformed the Google and FaCoY search engines in recommending fixes for this category of software errors.

We have proposed an approach called *Lucene+Score+How-to* to assist developers during some programming task with a given API. This approach recommends only Q&A pairs from Stack Overflow belonging to the *How-to* category based on a query (list of terms) made in natural language by the user. We conducted experiments to evaluate the recommendation strategy. The programming problems used in the experiments were extracted randomly from cookbooks for three topics widely used by the software development community: Swing (Java), Boost (C++) and LINQ (C#). The results have shown that for 27 of the 35 (77.14%) activities, at least one recommended pair proved to be useful to the target programming problem.

Keywords: Wisdom of the Crowd. Debugging with the crowd. API usage tasks.

Lista de ilustrações

Lista de tabelas

Tabela 44 – Resumo de técnicas e ferramentas de mineração de código. A coluna Granularidade especifica como os usuários especificam as consultas (Chamada de **A**PI, **C**onceito ou Caso de **T**este) e como os resultados da pesquisa são retornados para cada abordagem (Repositório da **A**plicação, **F**ragmento de Código ou **M**ódulo composto por um conjunto de classes da aplicação). A coluna Corpora especifica o escopo da pesquisa (**C**ódigo ou **D**ocumentos), seguido pela coluna Expansão que especifica se uma determinada abordagem utiliza a técnica de expansão de consulta para melhorar a precisão da busca. 113

Lista de siglas

API *Application Programming Interface*

ARFF *Attribute-Relation File Format*

DSL *Domain-Specific Language*

DT *Decision Tree*

IDE *Integrated Development Environment*

KAE *Known API Element*

KNN *K-Nearest Neighbors*

LINQ *Language Integrated Query*

LR *Logistic Regression*

MLP *Multi-Layer Perceptron*

NB *Naïve Bayes*

NDCG *Normalized Discounted Cumulative Gain*

NKAE *Not Known API Element*

Q&A *Question and Answer*

RF *Random Forest*

RSSE *Recommendation System for Software Engineering*

SO *Stack Overflow*

SVM *Support Vector Machine*

SVN *Apache Subversion*

Sumário

Capítulo 1

Introdução

O desenvolvimento moderno de programas é inseparável do uso de *Application Programming Interfaces* (APIs). Os desenvolvedores de *software* fazem uso constante de APIs como interfaces para bibliotecas de código ou *frameworks* para ajudar a acelerar o processo de desenvolvimento de *software* e melhorar a qualidade do mesmo (DUALA-EKOKO; ROBILLARD, 2012). Em outras palavras, as APIs são expostas aos desenvolvedores para permitir a reutilização de bibliotecas de *software* (MONPERRUS et al., 2012). Como qualquer outra ferramenta técnica sofisticada, as APIs devem ser devidamente aprendidas e operadas corretamente para serem efetivas. Para atingir esse objetivo, as APIs geralmente possuem documentação (UDDIN; ROBILLARD, 2015).

Devido ao fato de que uma boa documentação pode ajudar os desenvolvedores a trabalharem de forma eficiente, ela pode até mesmo servir para promover a API. Em contrapartida, uma documentação que não atende às expectativas dos seus leitores pode levar à frustração, perda de tempo importante e até mesmo ao abandono da API (ROBILLARD; DELINE, 2011). Nos últimos anos, vários trabalhos na literatura mostraram que um grande desafio para os desenvolvedores de *software* é aprender as APIs e lembrar como usá-las (DUALA-EKOKO; ROBILLARD, 2012) (ROBILLARD; DELINE, 2011) (MANDELIN et al., 2005) (OKUR; DIG, 2012) (PICCIONI; FURIA; MEYER, 2013) (ROBILLARD, 2009). O problema de aprendizado de APIs se deve principalmente ao fato de que a documentação oficial das APIs frequentemente contém erros gramaticais (ZHONG; SU, 2013) e/ou informação escassa sobre o *design* da API, ausência de cenários de uso e exemplos de código que ensinam como os elementos da API se relacionam (ROBILLARD; DELINE, 2011).

A manutenção da documentação oficial de uma API é uma tarefa desafiadora e que consome bastante tempo; consequentemente, a documentação se torna frequentemente desatualizada (SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014). Alguns trabalhos recentes confirmaram a crença dos desenvolvedores de que escrever uma documentação e mantê-la atualizada é bastante difícil (DAGENAIS; ROBILLARD, 2010) (DAGENAIS; ROBILLARD, 2012). Consequentemente, os desenvolvedores muitas vezes ignoram a do-

cumentação que existe (RIGBY; ROBILLARD, 2013). A área de Documentação de API tem recebido atenção considerável por parte da comunidade científica (ex. (TREUDE; ROBILLARD, 2016), (SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014), (SOUZA; CAMPOS; MAIA, 2014a), (HENß; MONPERRUS; MEZINI, 2012)). A documentação inadequada de uma API acarreta sérios obstáculos e problemas para o desenvolvimento de *software*. Por exemplo, os desenvolvedores podem escrever código inconsistente com a documentação da API por não saberem como utilizar corretamente um determinado método (MORENO et al., 2015) e, assim, introduzir defeitos no código (i.e. *bugs* relacionados com o uso incorreto de API (MONPERRUS; MEZINI, 2013), (SUSHINE; HERBSLEB; ALDRICH, 2015), (AMANI et al., 2016), (FAHL et al., 2012), (EGELE et al., 2013), (NADI et al., 2016), (GEORGIEV et al., 2012), (CAMPOS; MONPERRUS; MAIA, 2016)). Um outro problema consiste no fato de que os desenvolvedores muitas vezes enfrentam obstáculos ao tentar executar uma determinada atividade de programação com uma dada API. (CAMPOS; SOUZA; MAIA, 2016) (PONZANELLI et al., 2016) (SOUZA; CAMPOS; MAIA, 2014b) (PONZANELLI et al., 2014) (PONZANELLI; BACCHELLI; LANZA, 2013c) (PONZANELLI; BACCHELLI; LANZA, 2013a). São exemplos de obstáculos enfrentados pelos desenvolvedores durante as tarefas de desenvolvimento:

- Dificuldade no aprendizado de APIs extensas ou de *frameworks* complexos (ROBILLARD; DELINE, 2011; DUALA-EKOKO; ROBILLARD, 2012);
- Ausência de trechos de código na documentação oficial da API que ensinam como usar devidamente os métodos da mesma (MORENO et al., 2015; SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014);
- A evolução e atualização rápida de algumas APIs. Por exemplo, a plataforma Android e suas APIs evoluem rapidamente. Em média, existem 115 atualizações das APIs da plataforma por mês (MCDONNELL; RAY; KIM, 2013). Essas atualizações podem introduzir novas APIs, suspender as antigas ou até mesmo modificar o comportamento da API (WEI; LIU; CHEUNG, 2016).

Uma alternativa recente para mitigar problemas envolvendo o uso de APIs tem sido a disseminação de conhecimento em sítios de mídia social para desenvolvedores. Os últimos anos testemunharam o surgimento e a crescente popularidade de sítios de mídia social (também conhecidos como sítios Q&A) relacionados ao desenvolvimento de *software*, como o *Stack Overflow* [1], o *DaniWeb* [2] e o *Quora* [3]. Tais sítios de perguntas e respostas (Q&A) estão mudando rapidamente as maneiras pelas quais os desenvolvedores colaboram, aprendem e compartilham conhecimento (VASILESCU et al., 2014).

[1] https://stackoverflow.com/ (verificado em 11/01/2018)
[2] https://www.daniweb.com/ (verificado em 11/01/2018)
[3] https://www.quora.com/ (verificado em 11/01/2018)

A Sabedoria das Multidões ("*The Wisdom of Crowds*"), primeiramente publicado em 2005, é uma obra literária escrita por *James Surowiecki* (SUROWIECKI, 2005) sobre agregação de informação em grupos, resultando em decisões coletivas que, como ele argumenta, são quase sempre melhores do que as que poderiam ser feitas por qualquer membro do grupo. O autor do livro apresenta numerosos casos e anedotas para ilustrar seus argumentos, recorrendo a diversas áreas do conhecimento, principalmente à economia e psicologia. A informação disponível nesses sítios Q&A também é conhecida como "sabedoria da multidão" e atualmente é uma tendência importante no suporte de atividades relacionadas ao desenvolvimento e depuração de *software*. Parnin *et al.* (PARNIN; TREUDE, 2011) descobriram que os sítios Q&A realizam um bom trabalho de cobertura de APIs. Por exemplo, eles reportaram que 87% das classes da API Android foram referenciadas em respostas de *posts* do *Stack Overflow*.

Os problemas de uso de API surgem quando as decisões de *design* feitas pelos desenvolvedores da API não são bem compreendidas pelos seus usuários (ROBILLARD; DELINE, 2011). Ao criar as APIs, os projetistas precisam traçar uma linha tênue entre uma exposição excessiva dos elementos internos das APIs (o que violaria o princípio de *Information hiding* (PARNAS, 1972)) e um *design* que torna o comportamento da API completamente impenetrável e difícil de aprender (ROBILLARD; DELINE, 2011). Para preencher esta lacuna de facilitar o fluxo de conhecimento entre os desenvolvedores da API e seus usuários, o *site Stack Overflow* (SO) pode ser usado como uma forma de facilitar a troca de conhecimento entre programadores conectados via *Internet* (TREUDE; BARZILAY; STOREY, 2011)(MAMYKINA et al., 2011).

A utilização correta das APIs existentes é um desafio importante na programação, uma vez que as bibliotecas e APIs estão aumentando em número e complexidade (GLASSMAN et al., 2018). Estudos anteriores investigaram os diferentes tipos de perguntas que os desenvolvedores fazem quando enfrentam dificuldades ao usar as APIs (DUALA-EKOKO; ROBILLARD, 2012)(GLASSMAN et al., 2018). Para obter soluções para estas dúvidas de programação envolvendo o uso de APIs, duas situações são frequentes em relação ao comportamento dos desenvolvedores:

- **Situação A:** Eles realizam perguntas no SO (ex. fornecem um cenário e perguntam sobre como implementá-lo em uma determinada tecnologia ou API) e recolhem as respostas fornecidas pelos membros da comunidade, geralmente especialistas na tecnologia ou API;
- **Situação B:** Eles fornecem um ou mais trechos de código contendo *bugs* com o intuito dos membros da comunidade identificarem o *bug* e fornecerem uma correção para o mesmo.

De acordo com Nasehi *et al.* (NASEHI et al., 2012a), as perguntas no SO podem ser classificadas considerando as principais preocupações dos questionadores e o que eles

querem resolver. Eles identificaram quatro categorias principais de perguntas nesta dimensão:

- *Debug-corrective*: Perguntas que lidam com problemas no código de desenvolvimento, como erros em tempo de execução, notificações ou comportamento imprevisível. O questionador normalmente procura por uma revisão no seu código;
- *How-to-do*: O questionador provê um cenário e uma pergunta sobre como implementá-lo (algumas vezes informando uma dada tecnologia ou API);
- *Need-to-know*: Perguntas conceituais em um tópico particular (ex. definição de conceitos, melhores práticas para uma dada tecnologia). O questionador está esperando por uma explicação de um assunto particular ou sobre um determinado comportamento;
- *Seeking-different-solution*: O questionador está procurando por alguma coisa mais objetiva (ex. um livro, um tutorial, um *framework*, uma biblioteca) ou mais subjetiva (ex. um conselho, uma opinião, uma sugestão, uma recomendação).

Este trabalho de doutorado concentra-se em duas categorias principais de perguntas realizadas no SO pelos desenvolvedores, a saber: *How-to-do* (**Situação A**) e *Debug-corrective* (**Situação B**). Dessa forma, as abordagens de recomendação propostas nos Capítulos 3 e 4 deste trabalho visam recomendar somente *posts* do SO pertencentes a estas duas categorias, uma vez que somente eles podem possuir o conteúdo adequado para auxiliar os desenvolvedores de *software* durante as atividades de desenvolvimento e depuração envolvendo APIs.

Um outro comportamento comum dos desenvolvedores é fazer uso de mecanismos de busca na *Web*, como o *Google*, para localizar recursos *on-line* para melhorar sua produtividade (XIA et al., 2017). Motores de busca como o *Google*[4], o *Bing*[5] e o *Baidu*[6] tornaram-se ferramentas populares e importantes para auxiliar os desenvolvedores em diferentes tipos de tarefas de Engenharia de *Software* (XIA et al., 2017) (ex. corrigir erros em um programa). A recuperação do código na *Web* é um problema importante no desenvolvimento de *software* moderno, onde componentes e exemplos de uso presentes em páginas da *Web* ou em projetos de código aberto são amplamente utilizados (SIM et al., 2011). Bao *et al.* (BAO et al., 2015) apontaram que os desenvolvedores emitem mais de 20 consultas relacionadas ao desenvolvimento de *software* diariamente. Em outras palavras, a pesquisa de código é uma parte fundamental da compreensão do programa no desenvolvimento de *software* (XIA et al., 2017). Diversos motores de busca para código têm sido propostos na literatura (ex. *FaCoY* (KIM et al., 2018), *Prompter* (PONZANELLI et al., 2016)).

[4] http://www.google.com
[5] http://www.bing.com
[6] http://www.baidu.com

Diferentemente dos motores de busca de propósito geral como o *Google*, estes motores de busca foram projetados para receber trechos de código como entrada da consulta ao invés de termos em linguagem natural.

Sistemas de Recomendação para Engenharia de *Software* (RSSEs) (ROBILLARD; WALKER; ZIMMERMANN, 2010a) representam uma solução possível para os vários obstáculos (dificuldade no aprendizado de APIs extensas ou de *frameworks* complexos, a ausência de trechos de código na documentação oficial de uma API, etc.) enfrentados pelos desenvolvedores durante as atividades de desenvolvimento e depuração. Um sistema de recomendação reúne e analisa os dados, identifica artefatos úteis e sugere-os para o desenvolvedor. Algumas ferramentas seminais como eROSE (ZIMMERMANN et al., 2004), HIPIKAT (ČUBRANIĆ, D. MURPHY, G., 2003) e DEEPINTELLISENSE (HOLMES; BEGEL, 2008a) sugerem artefatos de projeto no IDE com o objetivo de fornecer aos desenvolvedores informações adicionais em partes específicas do sistema. De acordo com Happel e Maalej (HAPPEL; MAALEJ, 2008), RSSEs fornecem informações de desenvolvimento (código-fonte, artefatos, medidas de qualidade e ferramentas) e informações de colaboração (pessoas, conscientização, *status* e prioridades). Depois de obter e avaliar as decisões relevantes, os RSSEs ajudam os desenvolvedores de *software* na tarefa de encontrar informações relevantes para seu problema (ROBILLARD; WALKER; ZIMMERMANN, 2010a).

1.1 A Proposta

Este trabalho de doutorado propõe duas abordagens de recomendação distintas que possuem em comum o fato de utilizarem a "sabedoria da multidão" disponível no SO como base de conhecimento. A primeira abordagem de recomendação visa fornecer uma solução para lidar com o problema de *bugs* no código-fonte associados a usos incorretos de API por parte do desenvolvedor. Esta estratégia de recomendação tem como objetivo recomendar *posts* do SO que contenham correções para esta classe particular de *bugs* (i.e. *API-usage-related bugs*). Para isso, a estratégia de recomendação foi projetada para receber como entrada trechos de código-fonte Java ou JavaScript contendo *API-usage-related bugs*. A suposição é de que essa classe de *bugs* possui caráter geral (i.e. ocorre em diferentes domínios de aplicação com o mesmo sintoma). Logo, a comunidade do SO já pode ter identificado como corrigi-los e o conhecimento de como corrigir um dado *API-usage-related bug* pode ser reutilizado pelo desenvolvedor, livrando-o de gastar várias horas para tentar entender o *bug* e corrigir o mesmo. A segunda abordagem de recomendação visa fornecer uma receita de como resolver um dado problema de programação que um desenvolvedor esteja enfrentando utilizando uma determinada API fornecida. Para isso, esta estratégia de recomendação tem como objetivo recomendar pares Q&A (um par Q&A é composto por uma pergunta e uma resposta para esta pergunta) da categoria *How-to-*

do provenientes do SO. Além disso, esta estratégia de recomendação foi concebida para receber como entrada as descrições de tarefas de programação em linguagem natural bem como a API de interesse.

Neste trabalho, utilizou-se a seguinte definição para Sistemas de Recomendação:

> *"Um sistema de recomendação para engenharia de software (RSSE) é uma aplicação de software que fornece itens de informação estimados como valiosos para uma tarefa de engenharia de software em um determinado contexto (ROBILLARD; WALKER; ZIMMERMANN, 2010a)."*

A noção de **valor** presente nesta definição captura simultaneamente dois aspectos distintos: (i) Novidade e surpresa, porque os RSSEs ajudam a descobrir novas informações; e (ii) Familiaridade e reforço, porque os RSSEs suportam a confirmação do conhecimento existente. Finalmente, a referência a **uma tarefa específica e contexto** distinguem os RSSEs de ferramentas de pesquisa genéricas, por exemplo, ferramentas para ajudar os desenvolvedores a encontrar exemplos de código (ROBILLARD et al., 2014).

Esta definição de RSSEs é, no entanto, ainda ampla e permite uma grande variedade de suporte de recomendação para desenvolvedores. Especificamente, um grande número de itens de informação diferentes podem ser recomendados, tais como: código-fonte dentro de um projeto, código-fonte reutilizável, exemplos de código, relatórios de problemas, ferramentas, comandos, operações e até mesmo, pessoas (ex. ajudar a encontrar a melhor pessoa para atribuir uma tarefa de desenvolvimento) (ROBILLARD et al., 2014).

Um dos grandes desafios na criação de abordagens de recomendação para auxiliar os desenvolvedores durante as suas atividades de desenvolvimento de *software* está associado ao entendimento das necessidades de informação dos desenvolvedores de *software* (ROBILLARD et al., 2017). Este trabalho de doutorado difere das abordagens de recomendação do estado da arte (i.e. *FaCoY* (KIM et al., 2018) e *Prompter*(PONZANELLI et al., 2016)(PONZANELLI et al., 2014)) pois classifica o conteúdo a ser recomendado para os desenvolvedores levando em consideração as necessidades de informação dos mesmos. Em outras palavras, somente uma categoria particular de pergunta realizada no serviço de mídia social SO é recomendado (i.e. *posts* da categoria *How-to-do* pois somente eles possuem a característica de explicar passo a passo como solucionar uma tarefa de programação com uma dada API). Para atingir esse objetivo, foi necessário construir um classificador baseado em Aprendizado de Máquina Supervisionado. Esta classificação é importante para atender as necessidades de informação dos desenvolvedores de *software* e é uma etapa que não está presente nas abordagens de recomendação do estado da arte como *FaCoY* e *Prompter*. Sem esta etapa de classificação dos *posts* do SO, muito ruído pode ser introduzido na recomendação (ex. *posts* conceituais e/ou teóricos podem ser recomendados ao invés de *posts* mais práticos contendo trechos de código-fonte que exemplificam como proceder com uma dada tarefa de programação).

Muitas organizações de desenvolvimento de *software* se esforçam para aumentar a produtividade de seus desenvolvedores (MEYER et al., 2017). Algumas pesquisas reportaram a natureza fragmentada do trabalho de desenvolvimento de *software*, especificamente o efeito de interrupções na fragmentação do mesmo (ex. (SOLINGEN; BERGHOUT; LATUM, 1998)(IQBAL; HORVITZ, 2007)(CZERWINSKI; HORVITZ; WILHITE, 2004)(PARNIN; RUGABER, 2011)). Minelli *et al.* (MINELLI; MOCCI; LANZA, 2015) analisaram como os desenvolvedores passam seu tempo dentro do IDE e relataram que os tempos inativos são frequentemente usados com a compreensão do programa. Em um trabalho anterior, LaToza *et al.* (LATOZA; VENOLIA; DELINE, 2006) revelaram que os desenvolvedores passam a maior parte do tempo no IDE para escrever e compreender o código e que eles devem ficar focados somente na tarefa atual sem grandes interrupções ou pertubações. Entretanto, os desenvolvedores muitas vezes precisam de conhecimento adicional para completar as tarefas de programação (PONZANELLI et al., 2016) e são forçados a deixar o IDE para realizar consultas em motores de busca de propósito geral, interrompendo assim o fluxo de programação e reduzindo o foco na tarefa atual (PONZANELLI; BACCHELLI; LANZA, 2013a). Os RSSEs representam uma solução possível para automatizar o processo de recomendação de informação disponível eletronicamente (PONZANELLI et al., 2016). Um exemplo proeminente é o SO, um serviço de mídia social popular entre os desenvolvedores como um local para compartilhamento de conhecimento de programação (PONZANELLI et al., 2016)(CAMPOS; SOUZA; MAIA, 2016).

O SO tem um grande número de exemplos de código com curadoria (Sadowski, Caitlin and Stolee, Kathryn T. and Elbaum, Sebastian, 2015; UMARJI; SIM; LOPES, 2008). A qualidade dos mesmos tem sido verificada pelos usuários da comunidade que marcam aqueles trechos de código que ajudaram na resolução de um problema específico (SUBRAMANIAN; HOLMES, 2013). O desenvolvimento rápido e o efetivo mecanismo de controle de qualidade do SO o tornam uma base de conhecimento bem aceita e confiável para os desenvolvedores (YE; XING; KAPRE, 2017). Uma diferença chave entre listas de discussão e o SO é o conceito de gamificação (i.e. utilizar elementos de *design* de jogos em contextos que não são jogos) (DETERDING, 2012) (DETERDING et al., 2011). Desta forma, os membros da comunidade do SO são estimulados a competir entre si para obter pontos de reputação e emblemas. Estes recursos motivacionais também podem ser vistos como uma medida da experiência por recrutadores potenciais (CAPILUPPI; SEREBRENIK; SINGER, 2013) e são conhecidos por motivar os usuários a contribuir mais (ANDERSON et al., 2012)(DETERDING, 2012)(CAVUSOGLU; LI; HUANG, 2015)(ANTIN; CHURCHILL, 2011).

Um resumo conciso dos pontos principais desta tese está descrito abaixo:

Declaração de tese. Existe um conhecimento significativo sobre tarefas de desenvolvimento e depuração disponível *on-line* em serviços Q&A. Esta tese propõe soluções

customizadas de busca que levam em consideração os aspectos intrínsecos do contéudo de desenvolvimento e depuração de *software*. Além disso, a presente tese demonstra que é possível obter alguns ganhos na busca deste conhecimento, sejam eles advindos de consultas baseadas em trechos de código ou de termos escritos em linguagem natural.

1.2 Contribuições

Esta seção apresenta as principais contribuições científicas desta tese:

1. A proposição de uma estratégia de recomendação para trechos de código com potenciais *API-usage-related bugs* para auxiliar o desenvolvedor na correção dos mesmos. Esta estratégia utiliza funções de pré-processamento de código inéditas que foram criadas e investigadas no presente trabalho de doutorado;

2. A construção e definição dos atributos de um classificador de Regressão Logística (PENG; LEE; INGERSOLL, 2002) que utiliza **Aprendizado de Máquina Supervisionado** para classificar automaticamente pares Q&A do SO em três categorias de preocupações do questionador ao realizar uma pergunta no *site*: *How-to-do*, *Conceptual* e *Seeking-something*. A existência deste classificador para fazer a filtragem do conteúdo a ser recomendado para um desenvolvedor (i.e. pares Q&A da categoria *How-to-do*) revela o ineditismo do trabalho em relação às abordagens de recomendação do estado da arte (i.e. *Prompter* (PONZANELLI et al., 2016) (PONZANELLI et al., 2014), *Seahawk* (PONZANELLI; BACCHELLI; LANZA, 2013c)). Este classificador é um primeiro passo na intenção de capturar as necessidades de informação dos desenvolvedores de *software* (ROBILLARD et al., 2017) através da categorização dos diferentes tipos de preocupações que os desenvolvedores possuem ao realizar uma pergunta no SO (CAMPOS; MAIA, 2014) (CAMPOS; SOUZA; MAIA, 2016);

3. A realização de uma comparação quantitativa e qualitativa da abordagem proposta (*Lucene+Score+How-to*) com o *Google* (atualmente é o motor de busca de propósito geral mais popular na *Web*). Essa análise permitiu a identificação dos pontos fortes e fracos nas recomendações de ambas abordagens para as tarefas de programação que envolvem uso de alguma API.

Capítulo 2

Fundamentação Teórica

A estratégia de recomendação de pares Q&A (um par Q&A é composto por uma pergunta e uma resposta para esta pergunta) deste trabalho visa recomendar uma categoria específica de pares Q&A, i.e. somente pares da categoria *How-to-do*. Como dito anteriormente, os pares desta categoria particular possuem o formato de uma receita de um *Cookbook* (título da receita, cenário da tarefa e solução para a tarefa). Por exemplo, considere a seguinte tarefa de programação: "gerar números aleatórios usando a API *Boost* do C++". Caso esta receita estivesse presente em um *Cookbook* da API *Boost*, a solução para a tarefa seria um passo-a-passo de como resolver a tarefa nesta API. Os pares Q&A da categoria *How-to-do* também possuem a característica de mostrar passo-a-passo como resolver a tarefa (muitos desses pares possuem trechos de código que podem ser reusados com o objetivo de solucionar a tarefa). Portanto, estes pares possuem o potencial de auxiliar os desenvolvedores durante as tarefas de programação com uma determinada API. Porém, como classificar automaticamente milhares de pares Q&A do SO relacionados a uma determinada API?

Para identificar automaticamente os pares Q&A da categoria *How-to-do* para uma determinada API, é necessário identificar padrões textuais (ex. presença de determinadas palavras-chave) que possibilitam classificar um par Q&A do SO dentro de um número de categorias ou classes. Para isso, existem vários algoritmos de classificação amplamente conhecidos na área de Aprendizado de Máquina que viabilizam este tipo de classificação. Esses algoritmos são treinados e posteriormente testados em relação a um conjunto de dados previamente rotulados, ou seja, dado um par Q&A, é sabido a sua categoria (ex. *How-to-do*, *Conceptual* ou *Seeking-something*). Além disso, é necessário definir os atributos do classificador bem como definir quais são as categorias reconhecidas por ele.

A Seção 2.1 apresenta quais são os métodos de Aprendizado de Máquina para classificação considerados neste trabalho. A Seção 2.2 apresenta alguns conceitos para Aplicação de Métodos de Aprendizado de Máquina (ex. redução de atributos do classificador, como será realizado o treinamento e teste do classificador). Finalmente, a Subseção 2.3.1 apresenta o *software WEKA*, que foi utilizado no trabalho para estimar a taxa de acerto dos

classificadores.

2.1 Métodos de Aprendizado de Máquina para Classificação

Esta seção apresenta os algoritmos de classificação amplamente conhecidos na área de Aprendizado de Máquina que foram considerados neste trabalho.

2.1.1 *Logistic Regression*

Logistic Regression (LR) é um método estatístico multivariado para análise de dados com variáveis de resultados categóricos. O objetivo da LR é encontrar o modelo mais econômico e que melhor descreve a relação entre o resultado (variável resposta) e um conjunto de variáveis explicativas. Este método é relativamente robusto e flexível. Na LR, não são feitas suposições sobre a distribuição das variáveis explicativas (POHAR; BLAS; TURK, 2004).

A LR é amplamente usada em ciências médicas e sociais, e tem outras denominações, como modelo logístico, modelo logit, e classificador de máxima entropia. O êxito da LR assenta sobretudo nas numerosas ferramentas que permitem interpretar de modo aprofundado os resultados obtidos (POHAR; BLAS; TURK, 2004).

Enquanto método de predição para variáveis categóricas, a LR é comparável às técnicas supervisionadas propostas em aprendizagem automática (árvores de decisão, redes neurais, etc.), ou ainda a análise discriminante preditiva em estatística exploratória. É possível de as colocar em concorrência para escolha do modelo mais adaptado para um certo problema preditivo a resolver (POHAR; BLAS; TURK, 2004).

Trata-se de um modelo de regressão para variáveis dependentes ou de resposta binomialmente distribuídas. É útil para modelar a probabilidade de um evento ocorrer como função de outros fatores. É um modelo linear generalizado que usa como função de ligação a função **logit** (POHAR; BLAS; TURK, 2004).

Para melhorar as estimativas dos parâmetros do modelo e diminuir os erros cometidos em futuras previsões, pode-se combinar estimadores de cume (*ridge estimators*) com regressão logística (CESSIE; HOUWELINGEN, 1992).

2.1.2 *Naïve Bayes*

Os classificadores *Naïve Bayes* (NB) assumem que todos os atributos são independentes e que cada um contribui igualmente para a categorização. A categoria é atribuída a um projeto, combinando a contribuição de cada característica. Esta combinação é atingida estimando as probabilidades *a posteriori* de cada categoria utilizando o Teo-

rema de *Bayes*. As probabilidades *a priori* são estimadas com os dados de treinamento (LINARES-VáSQUEZ et al., 2014).

Este tipo de classificador é capaz de lidar com entradas categóricas e problemas de multi-classe. Portanto, em um problema de categorização, as entradas para o classificador são os atributos e a saída é a distribuição de probabilidade do projeto nas categorias (LINARES-VáSQUEZ et al., 2014).

2.1.3 *Multi-Layer Perceptron*

Uma rede neural *Multi-Layer Perceptron* (MLP) consiste em um conjunto de unidades sensoriais, que constituem a camada de entrada, uma ou mais camada(s) oculta(s) e uma camada de saída. O sinal de entrada propaga-se através da rede para a frente em uma base de camada-a-camada. Uma MLP representa uma generalização da rede neural *Single-Layer Perceptron* (HAYKIN, 1998).

MLPs têm sido aplicadas com sucesso para resolver alguns problemas difíceis e diversos, treinando-as de forma supervisionada com um algoritmo muito popular conhecido como algoritmo de retropropagação do erro (*back-propagation*) (HAYKIN, 1998).

2.1.4 *Support Vector Machine*

Support Vector Machine (SVM) divide o espaço do problema em dois conjuntos possíveis por encontrar um hiperplano que maximiza a distância com o ponto mais próximo de cada subconjunto. SVMs são classificadores binários, mas podem ser utilizados para classificação multi-classe (LINARES-VáSQUEZ et al., 2014).

A função que divide o hiperplano é conhecida como função *Kernel*. Se os dados são linearmente separáveis, uma função linear *Kernel* é usada com o SVM. Nos outros casos, funções não lineares tais como polinômios, funções de base radial e sigmóides devem ser utilizadas (LINARES-VáSQUEZ et al., 2014).

2.1.5 *Decision Tree*

Decision Trees (DTs) são algoritmos que utilizam a estratégia *divide and conquer* para dividir o espaço do problema em subconjuntos. Uma DT é modelada de forma que a raiz e os nós são as perguntas, e os arcos entre os nós são possíveis respostas para as perguntas. As folhas de uma DT representam as categorias do problema. DTs são capazes de lidar com entradas categóricas e problemas multi-classe (LINARES-VáSQUEZ et al., 2014).

J4.8 é um algoritmo existente para construção de uma DT que permite a manipulação tanto de atributos discretos quanto contínuos. Além disso, permite a manipulação de dados de treinamento com valores de atributos ausentes (SEHGAL; MOHAN; SANDHU, 2012).

2.1.6 *Random Forests*

Random Forests (RFs) são uma combinação de preditores de árvores de tal modo que cada árvore depende dos valores de um vetor aleatório amostrado de forma independente e com a mesma distribuição para todas as árvores da floresta (BREIMAN et al., 1984).

RFs são treinadas de forma supervisionada. O treinamento envolve a construção de árvores, bem como a atribuição a cada nó-folha da informação sobre as amostras de treinamento que atingem este nó-folha (ex. a distribuição da classe no caso de tarefas de classificação). Em tempo de execução, uma amostra de teste é transmitida a todas as árvores da floresta, e a saída é calculada pela média das distribuições registradas nos nós-folha alcançados (BREIMAN, 2001).

2.1.7 *k-Nearest Neighbors*

O algoritmo *k-Nearest Neighbors* (KNN) é um classificador preguiçoso porque não induz um modelo de categorização de dados de treinamento. O processo de categorização é conseguido através da comparação da nova instância com todas as instâncias do conjunto de dados. Assim, a categoria para a nova instância é selecionada a partir das categorias dos k exemplos mais similares. Em um problema de categorização, as entradas são as características e a saída é uma categoria (LINARES-VáSQUEZ et al., 2014).

2.2 Conceitos para Aplicação de Métodos de Aprendizado de Máquina

Esta seção apresenta alguns métodos que podem ser aplicados no processo de Aprendizado de Máquina bem como a definição do conceito de *Overfitting* (ver Subseção 2.2.1).

2.2.1 *Overfitting*

Overfitting é um problema reconhecido na área de Aprendizado de Máquina no qual o modelo se degenera e especializa no conjunto de treinamento. Dessa forma, o modelo não tem a capacidade de generalizar outros dados, e portanto, não deve ser utilizado. Assim, pequenas alterações nos dados de treinamento podem ter uma influência significativa sobre o resultado do exercício de aprendizagem e o modelo terá alta variância (CUNNINGHAM, 2000). A probabilidade de ocorrer *Overfitting* aumenta à medida que a dimensão do espaço de características aumenta (AZHAGUSUNDARI; THANAMANI, 2013).

2.2.2 *Cross-validation*

Cross-validation é um método estatístico utilizado para estimar a acurácia dos classificadores em um ambiente de aprendizado supervisionado. A forma básica do *Cross-validation* é a *k-fold cross-validation* (KOHAVI, 1995).

Na *k-fold cross-validation*, os dados são primeiramente particionados em k *folds* de mesmo tamanho. Subsequentemente, k iterações de treinamento e validação são realizadas de forma que dentro de cada iteração, um *fold* diferente é escolhido para validação enquanto os k - 1 *folds* restantes são utilizados para treinamento (KOHAVI, 1995).

O método *Cross-validation* é adequado para comparar o desempenho de dois ou mais algoritmos de classificação diferentes e encontrar o melhor algoritmo para os dados disponíveis. Além disso, este método estatístico é indicado quando a quantidade de dados rotulados é relativamente pequena (KOHAVI, 1995).

A principal vantagem de se utilizar a forma *k-fold cross-validation* é a sua estimativa precisa de desempenho de um classificador. Na área de Mineração de dados, *10-fold cross-validation* continua a ser o procedimento mais amplamente utilizado para validação. Em todos os testes conduzidos neste estudo, foi utilizado *10-fold cross-validation* (90% dos dados para construir o modelo e testar sua acurácia nos 10% restantes) (KOHAVI, 1995).

2.2.3 *Feature Selection*

A redução de atributos é um dos processos-chave para aquisição de conhecimento. Alguns atributos podem ser irrelevantes para a tarefa de mineração, ou redundantes. Deixar de fora os atributos relevantes ou manter atributos irrelevantes pode ser prejudicial, causando confusão no algoritmo de mineração empregado (AZHAGUSUNDARI; THANAMANI, 2013).

A seleção de um subconjunto de atributos é uma estratégia de *Feature Selection*, cujo objetivo é encontrar um conjunto mínimo de atributos de forma que a distribuição de probabilidade resultante das classes de dados é tão próxima quanto possível da distribuição original obtida utilizando todos os atributos (AZHAGUSUNDARI; THANAMANI, 2013).

As principais vantagens de se utilizar *Feature Selection* são:

- Seu uso pode ajudar a aumentar a acurácia em muitos problemas de aprendizado de máquina (GENG et al., 2007);
- Seu uso pode ajudar a melhorar a eficiência do treinamento do classificador (GENG et al., 2007);
- É um meio poderoso para se evitar o *Overfitting* do modelo (AZHAGUSUNDARI; THANAMANI, 2013).

Basicamente, os métodos de *Feature Selection* são divididos em três categorias (GUYON; ELISSEEFF, 2003): *filter*, *wrapper* (KOHAVI; JOHN, 1997) e *embedded method* (BREIMAN et al., 1984).

Um método do tipo *filter* calcula uma pontuação para cada característica e, em seguida, seleciona as características de acordo com a pontuação delas (MLADENIC; GROBELNIK, 1999). Yang *et al.* (YANG; PEDERSEN, 1997) e Forman (FORMAN, 2003) conduziram estudos comparativos em métodos do tipo *filter* e descobriram que *Information Gain* e *Chi-square* estão entre os métodos mais efetivos de *Feature Selection* para problemas de classificação.

2.3 Definição de Conceitos Fundamentais Empregados no Trabalho

2.3.1 O *Software WEKA*

O *WEKA* é um *software* de mineração de dados escrito em JAVA que foi desenvolvido por pesquisadores da Universidade de *Waikato* na Nova Zelândia. Este *software* possui uma coleção de algoritmos de Aprendizado de Máquina para tarefas de mineração de dados. Além disso, o *WEKA* implementa também algoritmos para: pré-processamento de dados, classificação, regressão, agrupamento e regras de associação.

O *software* inclui ferramentas de visualização e encontra-se licenciado ao abrigo da *General Public License* (GPL), sendo portanto possível estudar e alterar o respectivo código-fonte (HALL et al., 2009). A extensão do arquivo de dados normalmente utilizado pelo *WEKA* é o formato ARFF (*Attribute-Relation File Format*), que consiste em *tags* especiais para indicar diferentes elementos no arquivo de dados (ex. nomes de atributos, tipos de atributos e valores de atributos).

2.3.2 Interfaces de Programação de Aplicativos

As interfaces de programação de aplicativos (APIs) habilitam a reutilização de bibliotecas e *frameworks* no desenvolvimento de *software*. Em essência, uma API é um contrato entre o componente que fornece uma funcionalidade e o componente que usa essa funcionalidade (i.e. o cliente). Essas interfaces fornecem poderosos mecanismos de abstração que permitem que funcionalidades complexas sejam usadas por programas clientes (SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014). Mesmo que o comportamento de um componente possa ser completamente especificado por sua interface, os desenvolvedores frequentemente precisam de conhecimento auxiliar sobre esse elemento: como ele se relaciona com os termos do domínio, como combiná-lo com outros elementos, etc (ROBIL-

LARD; DELINE, 2011). Esse conhecimento geralmente é fornecido pela documentação, em particular, pela **documentação de referência da API**.

A documentação de referência da API é um conjunto de documentos indexados pelo nome do elemento da API, em que cada documento fornece especificamente informações sobre um elemento (classe, método, etc.). Por exemplo, a documentação da API do *Java Development Toolkit* (JDK) é um conjunto de páginas da *Web*, uma para cada pacote ou tipo na API (Maalej; Robillard, 2013). Embora existam muitas formas de documentação da API, normalmente há uma distinção clara entre a documentação de referência e outras formas de documentação com uma intenção mais pedagógica (por exemplo, tutoriais, livros e perguntas frequentes) (Maalej; Robillard, 2013).

2.3.3 O serviço de mídia social *Stack Overflow*

O *Stack Overflow* (SO) é um exemplo notável de serviço técnico de perguntas e respostas (Q&A) que ganhou popularidade entre os desenvolvedores de *software* desde sua criação em 2008. Este serviço oferece uma plataforma *Web* para os desenvolvedores discutirem problemas técnicos, de forma com que eles possam trocar conhecimento e resolver problemas envolvendo bibliotecas públicas não-documentadas, tarefas de programação pouco claras, ou explorar novas tecnologias e *frameworks* (PONZANELLI; BACCHELLI; LANZA, 2013c).

O SO emprega a técnica de **Gamificação** (DETERDING et al., 2011) (i.e. a utilização de elementos de *design* de jogos, tais como bonificações) para engajar mais os usuários. Desse modo, as perguntas e respostas são votadas pela comunidade e o número de votos é refletido na reputação e nos distintivos dos membros da comunidade. Os diferentes incentivos sócio-técnicos oferecidos pelo SO aumentam a participação nessa comunidade, facilitam as contribuições dos usuários e promovem a produtividade (VASILESCU et al., 2014).

Capítulo 3

Recomendação de correções para *bugs* relacionados com uso incorreto de API

A primeira estratégia de recomendação proposta neste capítulo faz uso do conhecimento disponível no SO para auxiliar os desenvolvedores durante as tarefas de depuração de código. Mais especificamente, tal estratégia é voltada para lidar com *bugs* relacionados com o uso incorreto de API (i.e. uma classe de *bugs* de *software* de caráter geral, independente de domínio de aplicação). Para isso, esta estratégia recebe como entrada um trecho de código contendo algum *API-usage-related bug* e recomenda para o usuário *posts* do SO que podem conter a descrição do *bug* e como resolvê-lo. Assim, esta estratégia de recomendação habilita os desenvolvedores a corrigir esta classe particular de *bugs*.

O uso incorreto de APIs é uma causa predominante de *bugs* de *software*, falhas e vulnerabilidades (MONPERRUS; MEZINI, 2013) (SUSHINE; HERBSLEB; ALDRICH, 2015) (AMANI et al., 2016) (FAHL et al., 2012) (EGELE et al., 2013) (NADI et al., 2016) (GEORGIEV et al., 2012) (CAMPOS; MONPERRUS; MAIA, 2016). Embora a documentação de alta qualidade de restrições de uso de uma API possa ajudar, muitas vezes não é suficiente, pelo menos na sua forma atual, para resolver o problema (DEKEL; HERBSLEB, 2009). Idealmente, os IDEs deveriam auxiliar os desenvolvedores a utilizar corretamente as APIs e detectar essas violações de uso (AMANN et al., 2018). Diversos trabalhos procuraram abordar o problema do uso incorreto de API e vários detectores de violações de uso de API foram propostos com o objetivo de ajudar os desenvolvedores a escrever melhores programas, avisando-os sobre os potenciais usos incorretos de API nos códigos-fonte dos seus programas (ex. *PR-Miner* (LI; ZHOU, 2005), *Colibri/ML* (LINDIG, 2007), *Jadet* (WASYLKOWSKI; ZELLER; LINDIG, 2007), *RGJ07* (RAMANATHAN; GRAMA; JAGANNATHAN, 2007b), *Chronicler* (RAMANATHAN; GRAMA; JAGANNATHAN, 2007a), *Grouminer* (NGUYEN et al., 2009), *AX09* (ACHARYA; XIE, 2009), *Car-Miner* (THUMMALAPENTA; XIE, 2009b), *Alattin* (THUMMALAPENTA; XIE, 2009a), *Tikanga* (WASYLKOWSKI; ZELLER, 2009), *Dmmc* (MONPERRUS; MEZINI, 2013), *DroidAssist* (NGUYEN et al., 2015)). Esses

detectores de violações de uso de API geralmente mineram padrões de uso (i.e. usos equivalentes da API que ocorrem frequentemente) e posteriormente relatam quaisquer anomalias em relação a esses padrões como potenciais usos incorretos. Apesar da grande quantidade de trabalhos nessa área, esses usos incorretos de API ainda existem na prática, como mostram alguns estudos recentes (NADI et al., 2016), (ACAR et al., 2016) e (LEGUNSEN et al., 2016).

A batalha contra *bugs* de *software* sempre existiu. Entretanto, esta é uma batalha que parece estar longe de ser ganha. Kim & Whitehead (KIM; WHITEHEAD JR., 2006) reportaram que o tempo médio para corrigir um único *bug* é de cerca de 200 dias. O processo de depuração de código está fortemente relacionado com a natureza do *bug*. Por exemplo, depurar um erro de falha de segmentação na linguagem C significa focar em alocação e desalocação de memória, enquanto depurar um erro de *NullPointerException* na linguagem Java significa encontrar qual linha de código está acessando algum atributo ou operação (método de API) em uma variável nula.

A estratégia de recomendação proposta neste capítulo visa endereçar uma classe particular de *bugs*, i.e. *bugs* derivados do não entendimento de desenvolvedores sobre como utilizar alguns métodos de API. Estes *bugs* devem ter uma característica em comum: eles deveriam ocorrer repetidamente em diferentes domínios de aplicação e serem independentes dos requisitos funcionais do *software*. Por exemplo, existe uma grande quantidade de desenvolvedores da linguagem JavaScript que tiveram problemas ao usar a função *parseInt*: ao executar *parseInt("08")*, o valor retornado é 0 (zero) enquanto o resultado esperado é 8. Esta classe particular de *bugs* é denominada "*API-usage-related bugs*" pois é provável que este tipo de *bug* tenha ocorrido em diferentes projetos e existe uma descrição do problema em algum lugar da *Web*, juntamente com a sua explicação e correção. Em outras palavras, existe uma grande chance de outros desenvolvedores terem enfrentado o mesmo problema e já terem identificado como corrigir o *bug*.

Quando uma falha de *software* ocorre, são necessárias três atividades para os desenvolvedores serem capazes de eliminar a mesma. A primeira atividade, **localização da causa da falha**, consiste em identificar quais comandos do programa são responsáveis pela falha. A segunda atividade, **compreensão da causa da falha**, envolve entender a causa raiz da falha. Finalmente, a **correção do defeito** consiste em determinar como modificar o código-fonte de modo a remover este defeito. Em outras palavras, é o defeito no código que leva a uma falha na operação do *software* (Avizienis, Algirdas and Laprie, Jean-Claude and Randell, Brian and Landwehr, Carl, 2004). Estas três atividades são referidas coletivamente com o termo depuração (do inglês, *debugging*) (PARNIN; ORSO, 2011).

Nos últimos anos, várias técnicas de pesquisa foram propostas para dar suporte automatizado ou semi-automatizado a diversas atividades de depuração (ex. (BALL; NAIK; RAJAMANI, 2003), (CLEVE; ZELLER, 2000), (GROCE; KROENING; LERDA, 2004),

(JONES; HARROLD; STASKO, 2002), (RENIERIS; REISS, 2003), (ZELLER; HILDEBRANDT, 2002), (ZHANG; GUPTA; GUPTA, 2006), (ZHANG; GUPTA; ZHANG, 2003)). Coletivamente, estas técnicas alavancaram o estado da arte em depuração. Entretanto, um problema comum com a maioria das abordagens existentes é que elas tendem a assumir um perfeito entendimento do *bug*, ou seja, elas assumem que simplesmente examinar um comando com falha isoladamente é sempre suficiente para um desenvolvedor detectar, entender e corrigir o *bug* correspondente (PARNIN; ORSO, 2011). Uma solução mais flexível permitiria que os desenvolvedores informassem um trecho de código suspeito de conter algum *API-usage-related bug* e obtivessem um conhecimento extra que poderia ajudar a entender o que possivelmente estaria errado com um determinado padrão de código. Sabe-se que um comportamento comum dos desenvolvedores é postar trechos de código-fonte que contém *bugs* em *sites* de pergunta-resposta (Q&A) com o objetivo de obter um diagnóstico da correção de um *bug* desejado. Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) apontaram que o SO [1] é particularmente efetivo para revisões de código, para perguntas conceituais e para novatos.

Abdalkareem *et al.* (ABDALKAREEM; SHIHAB; RILLING, 2017) analisaram *commits* relacionados com o SO (i.e. *commits* cujas mensagens de log continham a palavra-chave *"stackoverflow"* e suas variantes: a primeira letra em maiúscula, todas as letras em maiúscula ou com espaços) presentes em projetos do GitHub e identificaram 14 diferentes motivos pelos quais os desenvolvedores de *software* utilizam o SO (para maiores detalhes, veja a Tabela 2 do artigo (ABDALKAREEM; SHIHAB; RILLING, 2017)), que foram agrupadas em cinco categorias de alto nível: "Utilização de Conhecimento", "Documentação de *Bugs*", "Promoção do SO", "Melhorias de Sistema ou Característica" e "Reuso de Código". Além disso, eles encontraram que a comunidade do SO provê principalmente conhecimento técnico para os desenvolvedores. Por exemplo, 21% dos *commits* analisados estavam relacionados com uso incorreto de método de API. Outro exemplo é que para 13,08% dos *commits* investigados, o desenvolvedor corrigiu um *bug* no projeto e forneceu o endereço eletrônico de um *post* no SO onde o *bug* foi descrito.

3.1 Motivação para a Abordagem de Recomendação Proposta

Para os desenvolvedores que experimentam algum *API-usage-related bug*, a comunidade do SO pode ser consultada como uma forma de obter a correção do *bug*. Para isso, os desenvolvedores poderiam usar seus próprios trechos de código suspeitos de conter o *bug*. Entretanto, tal abordagem exigiria uma maneira eficaz de pesquisar a "sabedoria da multidão" (i.e. termo utilizado para se referir ao conjunto de informação presente em

[1] StackOverflow, http://stackoverflow.com/

serviços de mídia social) utilizando estes trechos de código suspeitos.

Diferentes técnicas de mineração de código e ferramentas têm sido propostas para recuperar componentes de *software* relevantes a partir de diferentes repositórios de *software* com o objetivo de auxiliar os desenvolvedores durante o processo de Reuso de *Software* (ex. *SourceForge* [2], *SearchCode* [3], *Exemplar* (MCMILLAN et al., 2012)). Entretanto, alguns motores de busca para código customizados foram projetados para indexar todas as informações dos projetos de *software* ao invés de somente trechos de código. Além disso, alguns motores de busca para código não foram projetados para permitir trechos de código serem utilizados como consulta. Por exemplo, motores de busca como *Sourceforge* e *Exemplar* permitem palavras-chave como entrada. Em outras palavras, motores de busca para código geralmente não são adequados para fazer o casamento (i.e. do inglês, *matching*) entre os trechos de código dos desenvolvedores e os trechos de código da comunidade do SO.

Este capítulo do trabalho de doutorado propõe auxiliar os desenvolvedores na correção desta classe particular de *bugs* (i.e. *API-usage-related bugs*) através da recomendação de *posts* do SO que possam conter correções para estes *bugs*. Para isso, a abordagem consiste em fazer o casamento entre os trechos de código presentes nas perguntas dos *posts* do SO com os trechos de código que o desenvolvedor está depurando. Para ser possível fazer este casamento, os trechos de código do desenvolvedor e do SO deverão invocar os mesmos métodos de API. A motivação para isto é que os textos das respostas providos pelos membros da comunidade do SO juntamente com os trechos de código podem ser úteis para auxiliar os desenvolvedores durante as tarefas de depuração uma vez que estes textos mostram como utilizar os trechos de código, i.e. o *post* inteiro pode ajudar a superar a distância cognitiva (KRUEGER, 1992) para entender problemas nestes trechos de código.

3.2 *API-usage-related Bugs*

API-usage-related bugs não são *bugs* em funções ou métodos de uma API. Neste trabalho, supõe-se que a API não contém *bugs* e que os *API-usage-related bugs* ocorrem devido a um uso incorreto de funções ou métodos de API por parte do desenvolvedor. Ao contrário dos *API-usage-related bugs*, *bugs* específicos de projeto estão relacionados a um não entendimento ou implementação incorreta de conceitos do domínio da aplicação.

3.2.1 Definição de *"API-usage-related Bugs"*

Definição: *Um "API-usage-related bug" é um bug que causa uma saída inesperada e incorreta ou comportamento resultante de um uso incorreto de uma API.*

[2] http://sourceforge.net/ (verificado em 22/04/2019)
[3] https://searchcode.com/ (verificado em 22/04/2019)

Esta definição será ilustrada abaixo com alguns exemplos. Em Java, existe um método da biblioteca *"java.lang.Math"* denominado **cos**. A assinatura completa deste método é *"public static double cos(double)"*. O argumento deste método é um ângulo, em radianos. Este método retorna o cosseno trigonométrico deste ângulo. Apesar da descrição aparentemente simples e do nome auto-descrito, este método gera problemas para muitos desenvolvedores, como testemunhado por dezenas de perguntas e respostas neste tópico nos *sites* de pergunta-resposta (Q&A) [4]. Muitas Q&As relacionadas com o mesmo problema: *Por que Math.cos() gera um resultado incorreto?*. Uma das razões é que se o argumento do método **Math.cos** for passado em graus ao invés de radianos, o método gera um resultado incorreto. Por que motivo esta pergunta é feita repetidas vezes? Acredita-se que a semântica do método **Math.cos** é contra-intuitiva para muitas pessoas (estas pessoas assumem erroneamente que o ângulo deve ser passado em graus), e consequentemente, o mesmo problema ocorre em várias situações de desenvolvimento, independentemente do domínio da aplicação. A correção para este problema é usar o método **Math.toRadians** para converter primeiro o ângulo para radianos, i.e. *"Math.cos(Math.toRadians(angle))"*.

Considere agora um outro exemplo em uma linguagem de programação diferente. Em JavaScript, existe uma função chamada **parseInt**, que converte uma dada *string* de entrada no seu valor inteiro correspondente. Esta função também gera problemas a muitos desenvolvedores, como testemunhado por dezenas de Q&As neste tópico nos *sites* Q&A [5]. Novamente, várias Q&As relacionadas com o mesmo problema: *Por que parseInt("08") produz o valor 0 e não 8?* A resposta é que se o argumento da função **parseInt** começa com 0, a *string* de entrada é convertida para o seu respectivo valor inteiro utilizando a base octal (i.e. base 8) ao invés da base decimal numérica (i.e. base 10). Acredita-se que a semântica da função **parseInt** é contra-intuitiva para muitas pessoas (essas pessoas assumem erroneamente que a base decimal será utilizada ao invés da base 8, talvez pelo fato do sistema decimal numérico ser mais frequentemente utilizado em operações gerais envolvendo números). A correção para este problema é especificar a base decimal como parâmetro do método *parseInt*, i.e. *"parseInt('08', 10);"* retorna o valor 8.

Ocariza *et al.* (OCARIZA et al., 2017) descobriram que cerca de 83% das falhas em JavaScript são causadas por erros manualmente introduzidos pelos programadores ao invés de código gerado automaticamente pelo servidor. Além disso, aproximadamente 75% das falhas em JavaScript pertencem à categoria de "Parâmetro de Método Incorreto".

3.2.2 Características dos *"API-usage-related Bugs"*

Como corrigir um "API-usage-related bug"? Para corrigir um *API-usage-related bug*, o desenvolvedor precisa ter certeza que o comportamento incorreto observado não é específico do domínio da aplicação. Uma vez que todas as explicações específicas

[4] <http://stackoverflow.com/search?q=Math.cos+java>
[5] <https://stackoverflow.com/search?q=[javascript]+title%3AparseInt&submit=search>

de domínio são descartadas, a maneira mais comum e sensata de corrigir o *bug* consiste em pesquisar por problemas similares na *Web*. Como os *API-usage-related bugs* não estão relacionados a conhecimentos e lógicas específicas do domínio da aplicação, eles podem ser depurados pela própria comunidade do SO. A descrição do sintoma é suficiente para que outros desenvolvedores relembrem a ocorrência do mesmo *bug* e forneçam a correção. O presente trabalho visa propor uma estratégia de recomendação (ROBILLARD; WALKER; ZIMMERMANN, 2010b) baseada no casamento de trechos de código entre o trecho suspeito que está sendo depurado pelo desenvolvedor e trechos de código relacionados presentes em perguntas do SO. Assim, quando o desenvolvedor estiver enfrentando algum *API-usage-related bug*, ele pode utilizar esta estratégia de recomendação para identificar uma resposta relevante no SO. Tal estratégia é fundamentada no fato de que as Q&As do SO frequentemente contém correções de *API-usage-related bugs*. Em suma, para corrigir um *API-usage-related bug*, pergunte à comunidade do SO utilizando o trecho de código suspeito de conter o *bug* (MONPERRUS; MAIA, 2014).

Por que a abordagem proposta utiliza trechos de código como consulta de entrada ao invés de termos em linguagem natural?

A resposta para esta pergunta consiste no fato de que compreender a causa raiz de uma falha para os desenvolvedores tipicamente envolve atividades complexas como navegar nas dependências do programa e executar novamente o programa com diferentes entradas (PARNIN; ORSO, 2011). Além disso, os *API-usage-related bugs* não geram rastros da pilha de exceção que permitem a localização do método de API contra-intuitivo. Por isso, uma abordagem que receba trecho de código como consulta ao invés de termos em linguagem natural parece ser a mais adequada para este caso.

Qual é a origem dos "API-usage-related bugs"? Eles aparecem quando o desenvolvedor da API toma decisões de *design* que vão contra o senso comum, em que "senso comum" é definido como a intuição e a expectativa compartilhada por muitos desenvolvedores. No exemplo da função **parseInt**, a maioria das pessoas esperam que **parseInt("08")** produza como resultado o valor 8. Existem pelo menos três razões principais por trás dos *API-usage-related bugs*. São elas:

1. A API sob consideração pode parecer compreensível o suficiente para os desenvolvedores. Assim, eles não lêem a documentação oficial da API;

2. A API pode assumir alguma coisa implicitamente. Por exemplo, a função **parseInt** implicitamente assume que prefixar a *string* com "0" significa que a base octal foi escolhida;

3. A API sob consideração pode estar mal documentada (XIE; PEI, 2006). Por exemplo, a documentação oficial da API pode não conter trechos de código que ensinam como utilizar um determinado método de API (MORENO et al., 2015)(SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014)(TREUDE; ROBILLARD, 2016). Desta

forma, o desenvolvedor fica perdido e pode utilizar algum método da API de maneira incorreta. Por exemplo, suponha que um desenvolvedor utilizou um método de API erroneamente. Além disso, considere que tal método não gerou erro em tempo de compilação do programa porém o mesmo gerou uma saída incorreta em tempo de execução do programa devido à forma com que os argumentos foram passados para o método. Esse exemplo mostra a introdução de um *API-usage-related bug* em um projeto de *software*.

Em Moreno *et al.* (MORENO et al., 2015), os autores focaram em um problema ainda em aberto que consiste em minerar exemplos de código concretos que ensinam como utilizar um determinado método de API. A abordagem proposta por eles denominada MUSE (**M**ethod **US**age **E**xamples) automaticamente retorna exemplos de código concretos para um método particular. Estes exemplos de código frequentemente possuem todos os passos necessários para invocar um dado método de API desejado (ex. quais parâmetros devem ser passados para o método e como instanciar estes parâmetros). O trabalho de Moreno *et al.* trata o mesmo problema proposto no presente trabalho de doutorado: recomendar exemplos de código que ensinam o desenvolvedor a utilizar um determinado método de API. Porém, existem algumas diferenças básicas entre os trabalhos:

- Este capítulo aborda a consequência (i.e. introdução dos *API-usage-related bugs* no código da aplicação cliente) de um não entendimento por parte do desenvolvedor de como usar um dado método de API. Já o trabalho de Moreno *et al.* não aborda os problemas relacionados com o uso incorreto destes métodos de API;

- Este capítulo propõe a recomendação de *posts* do SO que contenham exemplos de código e que ensinam como usar esses exemplos de código ao invés de recomendar somente trechos de código (como no trabalho de Moreno *et al.* e Holmes *et al.* (HOLMES; MURPHY, 2005)). O motivo para recomendar *posts* inteiros é que os textos em linguagem natural providos nas respostas podem ajudar a superar a distância cognitiva (KRUEGER, 1992) para entender problemas nestes trechos de código.

Em suma, apesar dos esforços da comunidade acadêmica e da natureza geral dos *API-usage-related bugs* (i.e. *bugs* independentes de domínio de aplicação), muitos problemas ainda não foram resolvidos com relação a esta classe de *bugs*:

- A correção deles ainda é um processo manual e que consome tempo e esforço por parte do desenvolvedor. Por exemplo, o desenvolvedor precisa primeiro isolar qual é o método de API que contém o *bug* e depois utilizar motores de busca de propósito geral como o *Google* para encontrar na *Web* a correção para o *bug*. Esta solução

pode estar em vários lugares na *Web*, como em fóruns, blogs ou em serviços Q&A como o *Stack Overflow* [6], *DaniWeb* [7] e *Quora* [8];

- ❏ Não existe uma abordagem de recomendação com resultados promissores que sugere correções para este tipo de *bug*. Esta abordagem poderia receber como entrada um trecho de código-fonte contendo uma chamada de método de API suspeita e retornar para o desenvolvedor um *ranking* com os top-10 *posts* do SO que contenham respostas de como solucionar o *bug* sob consideração.

Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) apresentaram um estudo empírico das perguntas e respostas do SO. O resultado chave deles em relação ao presente trabalho de doutorado é que perguntas postadas no SO contendo trechos de código-fonte são respondidas com maior frequência do que as outras. Em outras palavras, o SO é impulsionado por trechos de código. O trabalho de Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) sugere que a base de conhecimento do SO inclui diariamente muitas soluções para tarefas de programação bem como correções de *bugs* gerais enfrentados por vários programadores. Isto é um fator motivador para o presente trabalho de doutorado uma vez que vários *API-usage-related bugs* podem estar sendo continuamente adicionados no *site* e a correção deles presente nas respostas dos *posts* do SO pode ser recomendada ao desenvolvedor.

Neste sentido, a Seção 3.3 descreve os desafios e objetivos do presente trabalho de doutorado.

3.3 Desafios e Objetivos da Pesquisa

Existem vários **desafios** a serem superados no que tange ao tema. Dentre eles, destacam-se:

- ❏ Para verificar se a abordagem de recomendação proposta auxilia efetivamente os desenvolvedores a corrigir os *API-usage-related bugs*, é necessário recolher um conjunto de contextos de código provenientes de projetos reais de *software*. Além disso, estes contextos de código devem necessariamente invocar métodos de API suspeitos de provocar *API-usage-related bugs* dependendo da forma como são utilizados pelos desenvolvedores. Logo, um *data set* precisa ser criado com o objetivo de avaliar a eficácia da abordagem de recomendação. No entanto, a tarefa de construir este *data set* consome bastante tempo e esforço, além de ser uma tarefa difícil;

- ❏ Para a abordagem proposta ser capaz de recomendar soluções para *API-usage-related bugs*, é necessário que a pesquisa por código seja eficiente. Entretanto,

[6] http://stackoverflow.com/
[7] https://www.daniweb.com/
[8] https://www.quora.com/

este não é um problema trivial. Existe um grande corpo de trabalho na literatura para melhorar busca por código em sítios de pergunta-resposta (MISHNE; SHOHAM; YAHAV, 2012)(STOLEE; ELBAUM; DOBOS, 2014)(STOLEE; ELBAUM; DWYER, 2016) e incorporar a busca por código dentro dos IDEs com o objetivo de reduzir a troca de contexto entre os IDEs e os navegadores *Web* (BACCHELLI; PONZANELLI; LANZA, 2012)(BRANDT et al., 2010) (PONZANELLI et al., 2014);

- ❑ Para avaliar a qualidade da recomendação proposta e verificar se os *posts* do SO recomendados ajudam os desenvolvedores nas tarefas de correção de *API-usage-related bugs*, será necessário realizar uma análise qualitativa dos *posts* do SO retornados pela abordagem proposta para cada contexto de código pesquisado presente neste *data set*.

Os **objetivos** que se pretende alcançar são:

- ❑ Propor uma maneira eficiente e flexível para ajudar os desenvolvedores a corrigir *API-usage-related bugs* em seus projetos de *software*;
- ❑ Propor uma estratégia de recomendação com ampla aplicabilidade (i.e. a ideia é que a abordagem proposta possa ser utilizada tanto por desenvolvedores novatos quanto por desenvolvedores com maior experiência em programação).

3.4 Questão de pesquisa

Esta seção apresenta a questão de pesquisa (Q) investigada neste capítulo bem como a sua respectiva hipótese.

Q: *O ranqueamento produzido pela estratégia de recomendação proposta é eficaz para encontrar correções para potenciais API-usage-related bugs presentes em projetos reais de software?*

☛ *Hipótese (H): A estratégia de recomendação proposta possui um desempenho aceitável para recomendar correções para potenciais API-usage-related bugs presentes em projetos reais de software.*

A pergunta de pesquisa visa investigar como a estratégia de recomendação proposta lida com potenciais *API-usage-related bugs* presentes nos trechos de código de projetos de *software* do mundo real. Para responder a esta pergunta de pesquisa, conduziu-se um experimento dedicado para *API-usage-related bugs*. Esse experimento utilizou trechos de código-fonte retirados de projetos de *software* do mundo real hospedados no *site Open Hub*

Code Search [9]. Estes trechos de código contêm algumas chamadas de métodos de API que podem levar à introdução de *API-usage-related bugs* dependendo da forma com que estes métodos de API são utilizados pelos desenvolvedores. Para realizar este experimento, foram coletadas algumas chamadas de métodos de API no *site SO* que possuem este comportamento com o objetivo de construir um *data set* de avaliação da abordagem proposta.

3.5 Depuração de *API-usage-related bugs*: Proposta

Este capítulo propõe uma abordagem inédita para depuração de *API-usage-related bugs*. Esta abordagem modela o problema de depuração como sendo um problema de recomendação. Em outras palavras, a abordagem recebe como entrada um trecho de código suspeito de conter um *API-usage-related bug* e recomenda *posts* do SO que visam auxiliar o desenvolvedor no processo de correção deste *bug*. A abordagem de depuração proposta é relevante para *bugs* associados ao uso incorreto de API. Tais *bugs* possuem um caráter geral, podendo acontecer com diversos desenvolvedores em diferentes domínios de aplicação.

Para avaliar uma abordagem de recomendação que recebe trechos de código-fonte como entrada (i.e. trechos de código contendo potenciais *"API-usage-related bugs"*), construiu-se um banco de dados local contendo os dados do SO. A Subseção 3.5.1 explica o processo de construção deste banco de dados local. Além disso, definiu-se 3 tópicos para a avaliação da abordagem de recomendação proposta. Tais tópicos bem como o motivo de ter selecionado cada um deles são descritos na Subseção 3.5.2.

3.5.1 Banco de Dados Local do SO

Para ser possível a condução dos experimentos, obteve-se uma versão do *data dump* [10] público do SO (versão de Março de 2013) e esses dados foram importados em um banco de dados relacional. A tabela *"posts"* deste banco de dados armazena todas as perguntas e respostas postadas pelos usuários no *site* até a data em que o *dump* foi realizado. Por meio deste banco de dados, foram acessados os trechos de código-fonte presentes nos *posts*. Finalmente, definiram-se um total de seis funções de pré-processamento de código para investigar qual delas possuía o melhor desempenho para consultas baseadas em código. A Subseção 3.5.3 detalha cada uma delas.

[9] Open Hub Code Search, https://www.openhub.net/ (verificado em 22/02/2017)
[10] http://blog.stackoverflow.com/category/cc-wiki-dump/

3.5.2 Tópicos Considerados

Os seguintes tópicos foram considerados neste capítulo: **Java Android**, **Java não-Android** e **JavaScript**. O tópico **Java Android** foi investigado separadamente pois o desenvolvimento de aplicações móveis é um tópico atual, com uma tendência ascendente que aumenta rapidamente (LINARES-VáSQUEZ; DIT; POSHYVANYK, 2013). Além disso, existe uma ampla adoção do sistema operacional **Android** nos *smartphones* e *tablets* (WEI; LIU; CHEUNG, 2016). Já o tópico **JavaScript** explora o domínio do desenvolvimento *Web*, que está entre os tópicos mais ativos no SO. Uma pesquisa recente envolvendo mais de 26.000 desenvolvedores conduzida pelo SO relatou que o JavaScript é a linguagem de programação mais utilizada atualmente. [11]

3.5.3 Funções de Pré-processamento de Trechos de Código

A maioria dos trechos de código presentes nos *posts* do SO não são compiláveis (YANG; HUSSAIN; LOPES, 2016) (CHEN; KIM, 2015)(SUBRAMANIAN; HOLMES, 2013). Terragni *et al.* (TERRAGNI; LIU; CHEUNG, 2016) analisaram 491.906 *posts* do SO e descobriram que 91,59% desses *posts* contém trechos de código não-compiláveis, indicando que esses trechos não são executáveis e são semanticamente incompletos para uma análise estática precisa (MISHNE; SHOHAM; YAHAV, 2012). Esse fenômeno ocorre porque os trechos de código em sítios Q&A são escritos para fins ilustrativos, onde a compilação não é uma preocupação (TERRAGNI; LIU; CHEUNG, 2016).

Para lidar com o problema de trechos de código incompletos no SO e melhorar a busca baseada nesses trechos, definiu-se diversas funções de pré-processamento de código. Por exemplo, essas funções podem remover os caracteres de pontuação bem como os caracteres alfanuméricos de um trecho de código. O objetivo delas é extrair a essência de um trecho de código (i.e. o que difere um trecho de código de outro). Após esta etapa, diferentes índices foram construídos contendo trechos de código pré-processados por estas funções. Dessa forma, foi possível explorar a eficiência dessas técnicas durante a busca por código. Para a criação destes índices, optou-se por utilizar o motor de busca *Apache Solr* [12]. Este, por sua vez, é baseado no *Apache Lucene* ([13]). A pontuação *Lucene* usa uma combinação do Modelo Espaço Vetorial (VSM) (SALTON; WONG; YANG, 1975)(ERK; PADó, 2008) de Recuperação de Informações e o modelo Booleano (SALTON; FOX; WU, 1983) para determinar a relevância de um determinado documento para a consulta de um usuário.

Outros trabalhos na literatura também utilizaram este motor de busca para a construção de índices (ex. *Seahawk* (PONZANELLI; BACCHELLI; LANZA, 2013c), *Prompter* (PONZANELLI et al., 2016), *Exemplar* (MCMILLAN et al., 2012)). O trabalho proposto investiga o potencial de 6 funções de pré-processamento (apresentadas abaixo) para as

[11] Stack Overflow. 2015 Developer Survey. http://stackoverflow.com/research/developer-survey-2015
[12] http://lucene.apache.org/solr/ (verificado em 23/01/2018)
[13] https://lucene.apache.org/core/2_9_4/scoring.html (verificado em 11/04/2018)

linguagens de programação Java e JavaScript. Existem tanto funções léxicas quanto sintáticas para ambas linguagens. Outras funções de pré-processamento de código poderiam ser definidas para essas linguagens (ex. outras expressões regulares poderiam ter sido utilizadas, outros fatos sintáticos poderiam ter sido extraídos dos trechos de código, etc.). Esta subseção apresenta as funções de pré-processamento investigadas no presente estudo.

3.5.3.1 Função *"raw"*

Esta função não modifica o conteúdo do trecho de código-fonte, i.e. não realiza nenhum pré-processamento no trecho de código.

3.5.3.2 Função Léxica para Java e JavaScript (*pp1*)

Esta função retorna todas as sequências alfanuméricas presentes no trecho de código utilizando a expressão regular **"[0-9a-zA-Z $]+"**. Como efeito colateral, esta função de pré-processamento remove todos os caracteres de pontuação.

3.5.3.3 Função Sintática para Java (*pp2*)

Esta função considera elementos da gramática livre de contexto da linguagem: processa o trecho de código Java e retorna um conjunto de nós da árvore de sintaxe abstrata deste trecho de código (são eles: declaração de variável, declaração de método e invocação de método). A geração desta árvore para o trecho de código é realizada utilizando o componente *Eclipse JDT Core* [14].

3.5.3.4 Função Léxica para JavaScript (*pp3*)

Esta função retorna uma lista de valores léxicos (i.e. nomes de variáveis e literais *string*) para os trechos de código JavaScript utilizando a API *Rhino* [15].

3.5.3.5 *Lightweight Syntactic Analysis and Binding for Java* (lsab-java)

Esta função processa o trecho de código Java e extrai as respectivas chamadas de métodos. Decidiu-se por realizar uma análise sintática *"Lightweight"* ao invés de uma análise sintática regular, uma vez que esta última requer a construção de uma Árvore Sintática Abstrata Completa para o trecho de código. Todavia, como a maior parte dos trechos de código presentes no SO são incompletos, optou-se por desenvolver um *parser* próprio para lidar com esses trechos de código não-compiláveis. As características deste *parser* são:

[14] http://www.eclipse.org/jdt/core/ (verificado em 23/01/2018)
[15] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino (verificado em 23/01/2018)

- *Lightweight Syntactic Analysis*: O *parser* processa o trecho de código e identifica todas as declarações de variáveis bem como todas as chamadas de método (estas chamadas de método são extraídas com o auxílio de uma expressão regular) que ocorrem no trecho de código. Cada variável é mapeada com o seu respectivo tipo (ex. **Calendar x;** => a variável "x" é mapeada com seu respectivo tipo, ou seja, **Calendar**).

- *Lightweight Binding*: O *parser* resolve os *bindings* identificados na etapa anterior, ou seja, na fase de Análise Sintática (ex. **Calendar x; x.getInstance()** => a variável "x" é substituída pelo respectivo tipo, resultando em **Calendar.getInstance()**). Este *parser* gera como saída os fatos sintáticos: **Calendar.getInstance** e **getInstance**. Para cobrir os casos em que não é possível resolver o *binding* (i.e. alguns trechos de código do SO são incompletos e não possuem o comando de declaração da variável), optou-se por indexar somente o nome do método (ou seja, o nome que aparece após o ".") .

3.5.3.6 *Lightweight Syntactic Analysis and Binding for JavaScript* (lsab-js)

Esta função processa o trecho de código JavaScript e extrai as respectivas chamadas de métodos. Ela realiza os mesmos passos feitos pela função *lsab-java*, porém contém estratégias de *parser* específicas para a linguagem JavaScript (ex. a palavra *"function"* é uma palavra reservada da linguagem JavaScript, enquanto esta mesma palavra não é uma palavra reservada da linguagem Java). Quando um desenvolvedor declara um nome de função de domínio específico da aplicação, ele o faz utilizando a palavra reservada *"function"* (ex. **function nomeDaFunção()**). Neste estudo, os nomes de função de domínio específico do problema não são indexados, uma vez que o objeto alvo do estudo são os métodos de APIs. Estes, por sua vez, tem uma alta probabilidade de serem discutidos pela comunidade do SO, dado que são independentes do domínio da aplicação.

A Tabela 1 mostra as funções de pré-processamento de trechos de código-fonte para as linguagens de programação consideradas neste estudo.

Tabela 1 – Funções de pré-processamento de trechos de código-fonte para as linguagens de programação consideradas (i.e. Java e JavaScript).

Linguagem	Sem pré-processamento	Léxica	Sintática
Java	raw	pp1	pp2, lsab-java
JavaScript	raw	pp1, pp3	lsab-js

A Seção 3.6 descreve o experimento conduzido com potenciais *API-usage-related bugs* presentes em projetos reais de *software*. Este experimento foi realizado para responder a pergunta de pesquisa.

3.6 Experimento com *API-usage-related bugs*

Esta seção aborda o experimento realizado para estudar a eficácia de uma estratégia de recomendação para *API-usage-related bugs* baseado no SO. Antes de detalhar o experimento, será feita uma breve explicação do que é o procedimento de indexação e como a busca é realizada. Esta explicação é fornecida na próxima subseção.

3.6.1 Os procedimentos de indexação e busca

A técnica padrão de busca baseada em indexação consiste em duas funções de transformação t_{index} e t_{query}. A primeira função t_{index} transforma os documentos (i.e. *posts* do SO) em termos, enquanto a segunda função t_{query} transforma a consulta em termos também. O ranqueamento consiste em realizar o casamento entre os termos do documento e os termos presentes na consulta. As funções t_{index} e t_{query} não são necessariamente idênticas, uma vez que a natureza dos documentos (estilo, estrutura, etc.) é frequentemente diferente da natureza das consultas.

Na abordagem proposta, os termos são trechos de código pré-processados por uma das funções de pré-processamento de código. Para a construção desses índices contendo trechos de código pré-processados, utilizou-se o motor de busca *Apache Solr* (VELASCO, 2016) [16]. É importante salientar que a estratégia de recomendação proposta é independente do *software* de busca e indexação de documentos utilizado. Dessa forma, qualquer outro motor de busca poderia ser usado na implementação da abordagem (ex. *ElasticSearch* [17], *Sphnix* [18], etc.).

O resultado da busca em um índice do *Apache Solr* é uma lista ranqueada de documentos (i.e. *posts* do SO), em que o primeiro é o mais textualmente similar com a consulta de entrada e o último é o menos similar. Cada *post* neste ranqueamento possui um valor numérico denominado *Solr's score* que representa sua similaridade textual com a consulta (i.e. isto é feito pelo *Apache Solr* em sua implementação atual). Portanto, o primeiro *post* neste ranqueamento possui o maior *Solr's score* enquanto o último *post* possui o menor valor.

Para a realização do experimento com *API-usage-related bugs* presentes em diferentes projetos de *software* do mundo real, construiu-se um *data set*. O processo de construção deste *data set* é uma tarefa difícil. No SO, os usuários fazem diferentes tipos de perguntas. Nasehi *et al.* (NASEHI et al., 2012b) identificaram quatro categorias de perguntas relacionadas com as principais preocupações dos questionadores e o que eles queriam resolver no SO: *Debug-corrective*, *How-to-do-it*, *Need-to-know* e *Seeking-different-solution*. A categoria *Debug-corrective* é bastante próxima com o problema de pesquisa investigado

[16] *Solr* é uma plataforma de pesquisa de código aberto, escrita em Java, do projeto *Apache Lucene*
[17] https://www.elastic.co/
[18] http://sphinxsearch.com/

pelo fato de que os desenvolvedores postam o trecho de código contendo um *bug* no SO com o intuito de obter o diagnóstico de correção daquele *bug*. Todavia, muitos destes *bugs* não estão relacionados com uso incorreto de métodos de API. A próxima subseção detalha os passos necessários que foram realizados para a construção deste *data set*.

3.6.2 Construção do *data set* de *API-usage-related Bugs*

Os seguintes passos foram realizados para a construção deste *data set*:

- **Passo 1**: Utilizou-se 3 filtros nas perguntas dos *posts* do SO (i.e. a mesma deve conter código, estar marcada e ter sido respondida) para selecionar *posts* do SO candidatos a discutirem *API-usage-related bugs*;

- **Passo 2**: Uma análise qualitativa foi conduzida com os primeiros 50 *posts* do SO retornados para cada linguagem de programação (Java e JavaScript). O resultado desta análise foi a identificação de 30 *API-usage-related bugs*, sendo 15 da linguagem Java e 15 da linguagem JavaScript;

- **Passo 3**: Com o objetivo de obter código a partir de projetos de *software* do mundo real, analisou-se manualmente chamadas de método encontradas em trechos de código presentes em *posts* do SO. Após esta análise, selecionou-se os nomes de métodos de API relacionados com o respectivo *API-usage-related bug* (ex. um método de API contra-intuitivo). O nome de método correspondente foi consultado no *site Open Hub Code Search* [19]. Selecionou-se a primeira classe Java retornada e a primeira função JavaScript como sendo os trechos de código representativos para o potencial *API-usage-related bug*. Com o objetivo de produzir os trechos de código finais das classes Java, removeram-se dessas classes as variáveis de instância bem como os métodos que não invocavam o método de API consultado no *site*;

- **Passo 4**: Os trechos de código com potenciais *API-usage-related bugs* encontrados no *site Open Hub Code Search* foram selecionados para compor o *data set*. Após esta etapa, construiu-se um par de endereços *Web* para cada potencial *API-usage-related bug* presente neste *data set*: o endereço do trecho de código com potencial *API-usage-related bug* no *site Open Hub Code Search* e o endereço de um *post* arbitrário do SO que corrige este potencial *API-usage-related bug*.

O *data set* contendo os potenciais *API-usage-related Bugs* utilizados neste estudo está disponível *online* [20].

[19] Open Hub Code Search, https://www.openhub.net/
[20] <https://github.com/eduardocunha11/API-usage-related-bugs>

3.6.3 Experimento com os trechos de código do *data set* de *API-usage-related bugs*

O impacto de se utilizar trechos de código do mundo real no experimento é simular um cenário real de manutenção de *software*. Para cada potencial *API-usage-related bug* presente no *data set* (representado por um trecho de código de um projeto *open-source*, que pode levar a um *API-usage-related bug* e sua respectiva correção no SO), deseja-se calcular a posição da solução Q&A nos 10 primeiros resultados após a consulta com o trecho de código real. Então, para cada trecho de código consultado na estratégia de recomendação proposta, a posição do *post* do SO que soluciona o potencial *bug* foi calculada. Seria uma possibilidade avaliar os top-10 resultados para cada trecho de código consultado para obter a qualidade geral da recomendação uma vez que pode existir mais de um *post* do SO que discute a correção de um potencial *API-usage-related bug*. Entretanto, decidiu-se investigar a posição da correção do *bug* no ranqueamento para obter a acurácia da recomendação.

Antes de explicar como foram realizadas as consultas com potenciais *API-usage-related bugs* nos índices do *Apache Solr*, é necessário entender sobre o filtro de palavras-chave que foi utilizado para a construção da consulta. A próxima subseção apresenta a metodologia utilizada para construir este filtro de palavras-chave.

3.6.4 Construção do filtro de palavras-chave

A Tabela 2 mostra o filtro de palavras-chave utilizado na consulta do *Apache Solr*. O filtro é adicionado na consulta no momento em que a mesma é realizada no índice. Estas palavras-chave foram extraídas a partir de uma análise qualitativa envolvendo os 30 *posts* do SO presentes no *data set* que discutiam correções de *API-usage-related bugs*.

A Tabela 2 apresenta as palavras-chave mais recorrentes nos títulos dos *posts* selecionados (i.e. cada palavra-chave apareceu no mínimo 5 vezes no título destes *posts*). A próxima subseção explica como usar o filtro de palavras-chave para consultar um índice do *Apache Solr* desejado.

Tabela 2 – Filtro utilizado na consulta do Apache Solr.

Palavras-chave consideradas no filtro
unexpected, incorrect, wrong, output, return, returns, result, results, behavior, weird, strange, odd, problem, problems

A Figura 1 mostra uma visão geral da abordagem de recomendação proposta neste capítulo para recomendar correções de *API-usage-related bugs* presentes no SO.

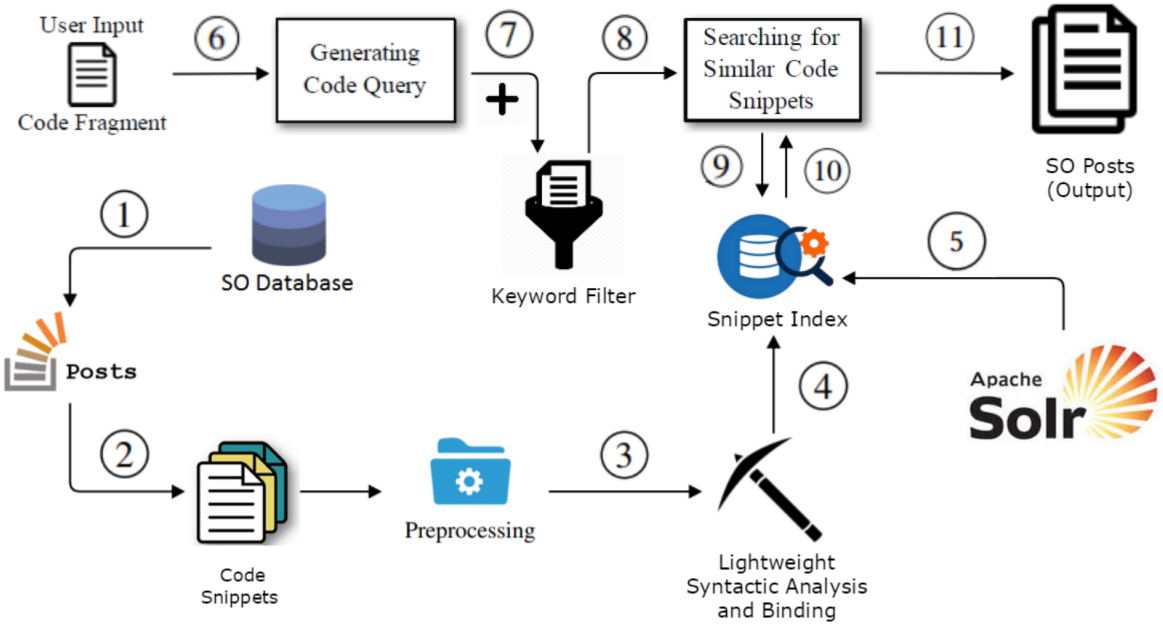

Figura 1 – Visão geral da abordagem de recomendação proposta.

3.6.5 Construção de consultas para o *Apache Solr* utilizando o filtro de palavras-chave

Considere o código abaixo escrito na linguagem Java presente no *data set* de *API-usage-related bugs* mostrado na Figura 2. Os demais trechos de código Java e JavaScript presentes no *data set* de *API-usage-related Bugs* estão disponíveis *online*: [21].

A Figura 3 mostra a utilização do filtro de palavras-chave na consulta do *Apache Solr* para o exemplo de código mostrado na Figura 2. Cada documento adicionado no índice do *Apache Solr* possui um conjunto de campos, que armazenam as informações sobre o registro do documento. Dois campos são mostrados na Figura 3: *"postTitle"* e *"codeText"*. O primeiro refere-se ao título do *post* do SO, enquanto o último contém a saída gerada pela função de pré-processamento para o exemplo de código mostrado na Figura 2. Na Figura 3, o exemplo de código foi pré-processado utilizando a função de pré-processamento *lsab-java*, que retorna aqueles termos denotando os nomes dos métodos de API invocados no exemplo de código.

A consulta relacionada com a Figura 3 possui o seguinte significado: retorne os *posts* do SO (i.e. documentos do índice) que possuem no mínimo uma das palavras-chave definidas na Tabela 2 presente no título deles e cujos trechos de código possuem as chamadas de métodos de API presentes no trecho de código pesquisado usando a estratégia de recomendação proposta. As consultas que não utilizam o filtro de palavras-chave não possuem o campo *"postTitle"*, mas somente o campo *"codeText"*.

Para cada um dos 15 potenciais *API-usage-related bugs* presente no *data set* estudado

[21] <https://github.com/eduardocunha11/API-usage-related-bugs>

```java
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import org.netbeans.lib.cvsclient.util.LoggedDataInputStream;

class ModTimeResponse implements Response {

    @Override
    public void process(LoggedDataInputStream dis, ResponseServices services) throws ResponseException {
              try {
                String dateString = dis.readLine();
                Date date = dateFormatter.parse(dateString.substring(0, dateString.length() - 6));
                if (date.getTime() < 0) {
                        if (date.getYear() < 100 && date.getYear() >= 70) {
                                date.setYear(date.getYear() + 1900);
                        } else if (date.getYear() >= 0 && date.getYear() < 70) {
                          date.setYear(date.getYear() + 2000);
                        } else {
                          date.setYear(2000 + date.getYear());
                        }
                }
              } catch (Exception e) {
                throw new ResponseException(e);
              }
    }
}
```

Figura 2 – Exemplo de código escrito na linguagem Java presente no *data set* de *API-usage-related Bugs* (Id: #15).

selecionados para a linguagem Java, foram realizadas 6 consultas diferentes utilizando o *Apache Solr*, sendo 2 consultas por índice (i.e. *pp1* **sem filtro**, *pp1* **com filtro**, *pp2* **sem filtro**, *pp2* **com filtro**, *lsab-java* **sem filtro** e *lsab-java* **com filtro**).

Para cada um dos 15 potenciais *API-usage-related bugs* presente no *data set* estudado selecionados para a linguagem JavaScript, foram realizadas 6 consultas diferentes utilizando o *Apache Solr*, sendo 2 consultas por índice (i.e. *pp1* **sem filtro**, *pp1* **com filtro**, *pp3* **sem filtro**, *pp3* **com filtro**, *lsab-js* **sem filtro** e *lsab-js* **com filtro**). É possível observar que a primeira consulta de cada índice não utiliza o filtro de palavras-chave, enquanto a segunda consulta de cada índice utiliza este filtro.

As funções de pré-processamento *pp1* e *raw* sofrem do mesmo problema: a presença de palavras reservadas da linguagem de programação (ex. palavras de parada como: *"public"*, *"class"*, etc.). Portanto, no experimento, os índices **Java-raw** e **JavaScript-raw** não foram considerados.

3.6.6 Comparação de resultados com o *Google*

Há evidências substanciais na literatura para apoiar a premissa de que os desenvolvedores usam mecanismos de busca de propósito geral durante o desenvolvimento de *software*

```
((postTitle:unexpected) OR (postTitle:incorrect OR
(postTitle:wrong) OR (postTitle:output) OR
(postTitle:return) OR (postTitle:returns) OR
(postTitle:result) OR (postTitle:results) OR
(postTitle:behavior) OR (postTitle:weird) OR
(postTitle:strange) OR (postTitle:odd) OR
(postTitle:problem) OR (postTitle:problems)) AND
codeText:(LoggedDataInputStream.readLine
readLine dateFormatter.parse parse Date.getTime
getTime Date.getYear getYear Date.setYear
setYear Date.getYear getYear Date.setYear setYear
Date.setYear setYear)
```

Figura 3 – Exemplo de consulta do *Apache Solr* utilizando o filtro de palavras-chave.

(ex. (MARTIE; HOEK; KWAK, 2017), (SIM et al., 2011), (STOLEE; ELBAUM; DOBOS, 2014), (XIA et al., 2017)). Sim *et al.* (SIM et al., 2011) realizaram um estudo comparativo sobre várias técnicas de pesquisa de código de desenvolvedores e observaram que o motor de busca de propósito geral funciona melhor para encontrar exemplos de referência do que as ferramentas dedicadas de pesquisa de código consideradas (i.e. *Koders*, *Krugle* e *Google Code Search*). Além disso, o trabalho de Sim *et al.* (SIM et al., 2011) mostrou que o *Google* obteve melhor desempenho em consultas por blocos de código do que os motores de busca específicos para código-fonte.

Os resultados da busca utilizando a abordagem de recomendação proposta foram comparados com os resultados de busca utilizando o motor de busca *Google*, que é o motor de busca de propósito geral mais usado para pesquisas relacionadas ao desenvolvimento de *software* (SIM et al., 2011).

Outros motores de busca específicos para código-fonte foram descontinuados (ex. *Google Code Search*, *Black Duck Open Hub Code Search*, *Koders*, *Krugle*). Além disso, esses motores de busca não são adequados para esta comparação pois os mesmos não indexam o conteúdo do SO. Em outras palavras, eles abordam um escopo de problema diferente.

Para cada um dos 30 potenciais *API-usage-related bugs* presentes no *data set* selecionados para as linguagens de programação Java e JavaScript, utilizou-se o motor de busca *Google* para pesquisar somente no *site* SO com o objetivo de entender até que ponto este motor de busca recupera algum *post* do SO mapeado no *data set*.

Definiu-se um protocolo para consultar o motor de busca *Google*, baseado nas seguintes regras:

- ❏ A consulta restringiu-se ao *site* SO. Para atender esta regra, adicionou-se o seguinte parâmetro de busca **"site:stackoverflow.com"** na consulta;

- ❏ O motor de busca *Google* não possui bom desempenho com consultas longas. Portanto, a consulta foi limitada até um máximo de 32 palavras (limiar adotado pelo próprio motor de busca *Google*).

Além disso, investigou-se o desempenho do motor de busca *Google* para buscas de código em que a consulta era composta por trechos de código pré-processados. As funções de pré-processamento *pp1*, *lsab-java* e *lsab-js* foram aplicadas nos trechos de código Java e JavaScript presentes no *data set*. Realizaram-se também consultas no motor de busca *Google* sem o pré-processamento destes trechos de código, i.e. consulta *raw*.

3.6.7 Comparação dos resultados com a abordagem do estado da arte FaCoY

Os resultados de busca utilizando a abordagem de recomendação proposta foram comparados com os resultados de busca utilizando um motor de busca para código do estado da arte denominado FaCoY (KIM et al., 2018).

Neste trabalho, utilizou-se o motor de busca para código FaCoY (KIM et al., 2018) pelos seguintes motivos:

1. Ao contrário dos mecanismos de pesquisa *on-line* de código para código, como o *Krugle* [22] e *Searchcode* [23], o motor de busca FaCoY pode identificar fragmentos de código semelhantes para um conjunto mais diversificado de implementações de funcionalidade. Esses fragmentos de código podem ser **sintaticamente diferentes** da consulta apesar de possuírem uma implementação de funcionalidade semelhante.

2. O motor de busca FaCoY encontra mais clones de código semânticos no *benchmark* **BigCloneBench** (SVAJLENKO et al., 2014) do que os detectores de clone do estado da arte, como *Deckard* (JIANG et al., 2007), *NiCad* (CORDY; ROY, 2011), *CCFinderX* (KAMIYA; KUSUMOTO; INOUE, 2002) e *SourcererCC* (SAJNANI et al., 2016);

3. O motor de busca FaCoY pode ser útil na recomendação da correção para defeitos de *software* (do inglês, *patch* ou *bugfix*).

Para a condução do experimento, utilizou-se os mesmos trechos de código Java presentes no *data set* de *API-usage-related bugs*. Com relação aos trechos de código JavaScript

[22] http://krugle.com/ (Novembro de 2018)
[23] https://searchcode.com/ (Novembro de 2018)

presentes neste *data set*, os mesmos não foram considerados no experimento pois a abordagem FaCoY foi designada apenas para a linguagem Java.

Para cada um dos 15 potenciais *API-usage-related bugs* presentes no *data set* selecionados para a linguagem de programação Java, utilizou-se o motor de busca para código FaCoY para recuperar *posts* do SO que continham trechos de código semanticamente similares aos trechos de código de entrada. O objetivo do experimento consistiu em verificar se o ranqueamento produzido pela abordagem FaCoY retornava os *posts* do SO mapeados no *data set*.

Não foi possível comparar a abordagem proposta com outros motores de busca para código do estado da arte por causa das razões listadas na Tabela 3.

Tabela 3 – Indisponibilidade de ferramentas e técnicas de busca de código.

Abordagem	Motivo
Portfolio (MCMILLAN et al., 2011)	A ferramenta não está disponível e suporta apenas C++.
Exemplar (MCMILLAN et al., 2012)	A ferramenta não está disponível.
Sourcerer (LINSTEAD et al., 2009)	A equipe não respondeu sobre o uso de seu mecanismo de pesquisa de código.
MUSE (MORENO et al., 2015)	A ferramenta não está disponível.
SNIFF (CHATTERJEE; JUVEKAR; SEN, 2009)	o motor de busca não funcionou devido a um problema com o *plugin* do Eclipse.
CodeHow (LV et al., 2015)	A ferramenta não está disponível (apenas um vídeo de demonstração *on-line*).
Prompter (PONZANELLI et al., 2016)	A ferramenta não está disponível. O autor foi contactado mas não conseguiu resolver o problema.

3.7 Resultados e Discussão do Experimento

Esta seção apresenta e discute os resultados obtidos no experimento. A Tabela 4 possui a seguinte estrutura: a primeira coluna contém o identificador (i.e. *id*) para cada trecho de código do *Open Hub* presente no *data set*, a segunda coluna contém o endereço *Web* de um *post* arbitrário do SO mapeado no *data set* que corrige um determinado potencial *API-usage-related bug* escrito na linguagem Java. As colunas restantes referem-se às posições ocupadas por esses *posts* do SO no ranqueamento **top-10** produzido pelo motor de busca *Apache Solr* e pelo motor de busca *Google*. Com relação ao motor de busca *Apache Solr*, foram consideradas as funções de pré-processamento de código *pp1*, *pp2* e *lsab-java* com ou sem o uso do filtro de palavras-chave na consulta do *Apache Solr*. Com relação ao motor de busca *Google*, foram consideradas as funções de pré-processamento de código *raw*, *pp1*

e *lsab-java*. Cada linha da Tabela 4 corresponde a um potencial *API-usage-related bug* escrito em Java mapeado no *data set*.

A Tabela 5 possui uma estrutura semelhante à Tabela 4, porém está relacionada com potenciais *API-usage-related bugs* escritos na linguagem JavaScript. Com relação ao motor de busca *Apache Solr*, foram consideradas as funções de pré-processamento de código *pp1*, *pp3* e *lsab-js* com ou sem o uso do filtro de palavras-chave na consulta do *Apache Solr*. Com relação ao motor de busca *Google*, foram consideradas as funções de pré-processamento de código *raw*, *pp1* e *lsab-js*.

As células coloridas das Tabelas 4 e 5 indicam as posições nos **top-10** resultados de busca encontrados pelo *Google* ou *Apache Solr*. Observou-se que o *Google* não obteve bons resultados na pesquisa por correções de potenciais *API-usage-related bugs*, mesmo com os trechos de código pré-processados pelas funções de pré-processamento *lsab-java* e *lsab-js*. Estes resultados indicam que estas funções de pré-processamento combinadas com o uso do filtro de palavras-chave superam o *Google* na busca por correções de potenciais *API-usage-related bugs*. As Tabelas 4 e 5 mostram a probabilidade de encontrar um *post* do SO que corrige um potencial *API-usage-related bug* escritos nas linguagens de programação Java ou JavaScript aumenta quando se faz uso do filtro de palavras-chave na consulta do *Apache Solr* (i.e. menos resultados não-encontrados: células "-" das tabelas). Após a realização do experimento, foi possível responder a pergunta de pesquisa.

Resumo da Questão de pesquisa. O experimento permitiu concluir pela não rejeição da hipótese H, dada a ausência de evidências que a provem o contrário. Além disso, o experimento revelou que a abordagem proposta obteve melhores resultados para Java do que para JavaScript. Para ambas as linguagens, a abordagem proposta obteve os melhores (e mais promissores) resultados usando, respectivamente, as funções de pré-processamento de código **lsab-java** e **lsab-js**, combinado com o uso do filtro de palavras-chave. De maneira geral a abordagem proposta superou o *Google*. No entanto, este por sua vez, encontrou correções de *API-usage-related bugs* em casos específicos em que a abordagem proposta não conseguiu encontrar (quatro dos trinta casos). Além disso, a abordagem proposta também superou o motor de busca para código FaCoY para encontrar correções no SO para esta classe particular de defeitos de *software*.

Apesar das funções de pré-processamento de código *lsab-java* e *lsab-js* serem baseadas na mesma ideia (i.e. extração das chamadas de método de API de um trecho de código e resolução de *bindings*), os resultados do experimento mostram um melhor desempenho da abordagem proposta para Java em relação ao JavaScript. Isso pode ser explicado pelo fato da linguagem JavaScript conter características únicas, tais como: tipagem fraca (PRADEL; SCHUH; SEN, 2015) (JENSEN; MØLLER; THIEMANN, 2009), acesso e criação de campo dinâmico (RICHARDS et al., 2010), *callbacks* (i.e. funções de ordem superior) aninhadas e assíncronas (GALLABA; MESBAH; BESCHASTNIKH, 2015)(GALLABA

Tabela 4 – Resultados das consultas realizadas nos índices Java *pp1*, *pp2* e *lsab-java* (sem/com o uso do filtro de palavras-chave na consulta do *Apache Solr*; "-" = *Post* do SO não encontrado no ranqueamento; "P" refere-se às posições ocupadas pelos *posts* do SO no ranqueamento).

Id	*Post* que corrige o *bug* stackoverflow/questions/	*Solr sem Filtro*			*Solr com Filtro*			*Google*			*FaCoY*
		P_{pp1}	P_{pp2}	P_{lsab}	P_{pp1}	P_{pp2}	P_{lsab}	P_{raw}	P_{pp1}	P_{lsab}	P_{raw}
1	13896614	-	-	-	-	-	-	-	-	-	-
2	9065727	-	-	2	-	-	1	-	-	3	-
3	12975924	-	-	1	-	-	1	1	-	2	-
4	10603336	-	-	-	-	-	9	-	-	-	6
5	11597244	-	-	-	6	-	5	-	-	2	2
6	6899019	-	-	-	6	2	-	-	-	-	2
7	9956471	-	-	-	-	-	-	4	-	-	-
8	12175674	2	-	2	2	1	1	-	-	-	1
9	1755199	-	-	-	8	-	-	-	-	-	-
10	12213877	-	-	1	-	-	1	-	-	-	-
11	14102593	-	-	-	-	-	6	-	-	-	-
12	9230961	-	-	-	-	-	-	-	-	-	1
13	14213778	-	-	-	-	8	6	-	-	-	-
14	14836004	-	-	-	-	9	3	-	-	-	4
15	7215621	-	-	-	-	4	3	-	-	-	-

Tabela 5 – Resultados das consultas realizadas nos índices JavaScript *pp1*, *pp3* e *lsab-js* (sem/com o uso do filtro de palavras-chave na consulta do *Apache Solr*; "-" = *Post* do SO não encontrado no ranqueamento; "P" refere-se às posições ocupadas pelos *posts* do SO no ranqueamento).

Id	*Post* que corrige o *bug* stackoverflow/questions/	*Solr sem Filtro*			*Solr com Filtro*			*Google*		
		P_{pp1}	P_{pp3}	P_{lsab}	P_{pp1}	P_{pp3}	P_{lsab}	P_{raw}	P_{pp1}	P_{lsab}
16	12318830	-	-	-	-	-	-	3	-	-
17	9859995	-	-	-	-	-	5	-	-	-
18	11477415	-	-	-	-	-	-	-	-	-
19	8979093	-	-	-	-	-	-	-	-	-
20	2587345	-	-	-	-	-	-	-	-	-
21	834757	-	-	5	3	9	1	-	2	-
22	1845001	-	-	-	-	-	-	-	-	-
23	9766201	-	-	-	-	-	1	-	-	-
24	8160550	-	-	-	-	-	-	-	-	-
25	11363526	-	-	-	2	-	-	2	-	-
26	6223616	-	-	2	-	-	-	-	-	-
27	8524933	-	-	-	-	-	2	-	-	-
28	10706272	-	-	-	7	1	2	-	-	-
29	18509996	-	-	-	-	-	9	-	-	-
30	262427	-	-	-	-	-	-	3	1	1

et al., 2017), etc.

Linguagens dinâmicas como o JavaScript dão aos programadores a liberdade de ignorar os tipos e permitem que eles escrevam código conciso em pouco tempo. Entretanto, esta liberdade pode acarretar problemas no código associados à inconsistência de tipos (PRADEL; SCHUH; SEN, 2015). Por exemplo, Jensen *et al.* (JENSEN; MØLLER; THIEMANN, 2009) (JENSEN; MADSEN; MØLLER, 2011) encontraram inconsistências de tipo em trechos de código JavaScript. Pradel *et al.* (PRADEL; SCHUH; SEN, 2015) propuseram uma abordagem denominada *TypeDevil* para detectar tipos inconsistentes em programas JavaScript. Devido ao fato da linguagem JavaScript ser fracamente tipada (ao contrário da linguagem Java que é fortemente tipada), a função de pré-processamento *lsab-js* não é capaz de resolver os *bindings* (i.e. substituir o nome da variável pelo respectivo tipo) adequadamente. Um outro aspecto que dificulta o pré-processamento de

trechos de código JavaScript é que os mesmos utilizam extensivamente *callbacks* aninhadas e assíncronas (GALLABA; MESBAH; BESCHASTNIKH, 2015) (GALLABA et al., 2017). Este aninhamento de *callbacks* (i.e. uma função *callback* que possui chamadas para outras funções assíncronas) aumenta a complexidade do código (GALLABA; MESBAH; BESCHASTNIKH, 2015), dificultando assim, a extração das chamadas de método de API desejadas pela função de pré-processamento *lsab-js*.

3.7.1 Ameaças à Validade

A principal **ameaça à validade externa** é que o *data set* utilizado é pequeno e pode não ser representativo dos *data sets* do mundo real. Entretanto, todos os trechos de código utilizados na avaliação são potenciais *API-usage-related bugs* presentes em projetos reais de *software*. Observe que muitos estudos existentes avaliaram a abordagem deles considerando trechos de código do mundo real. Além disso, estes estudos possuem *data sets* cujos tamanhos são menores do que o tamanho do *data set* investigado neste estudo.

Outra **ameaça à validade externa** é que a abordagem proposta depende diretamente da base de dados do SO. Atualmente, os resultados são apresentados considerando o *data dump* de Março de 2013. No entanto, a base de dados do SO está crescendo rapidamente. Existem aproximadamente 12 milhões de perguntas na base de dados atual (Junho de 2016), enquanto em Março de 2013, havia cerca de 3 milhões de perguntas.

A principal **ameaça à validade interna** é que recentemente *Zhang et al.* (ZHANG et al., 2018) mostraram que existem violações severas de uso de API em *posts* do SO que poderiam produzir comportamento inesperado no programa, como falhas e vazamentos de recursos. O trabalho deles revelou que mesmo os *posts* do SO aceitos como respostas corretas ou votados por outros programadores não são necessariamente mais confiáveis do que os outros *posts* em termos de uso incorreto de API. Para mitigar este problema, os trechos de código do SO podem ser complementados com exemplos relacionados de uso da API em projetos do *GitHub*, como proposto recentemente na literatura (i.e. *ExampleCheck* (ZHANG et al., 2018)).

Outra **ameaça à validade interna** é o tamanho dos trechos de código com potenciais *API-usage-related bugs* consultados na abordagem de recomendação proposta. Acredita-se que quanto maior é o tamanho do trecho de código consultado, maior é a probabilidade de introduzir algum ruído nos resultados da busca, uma vez que o número de métodos de API considerados também é maior (i.e. diferentes métodos de API podem confundir a abordagem proposta). Por outro lado, a abordagem proposta é uma solução flexível pois permite que os desenvolvedores informem seus próprios trechos de código suspeitos de diferentes tamanhos e obtenham conhecimento extra a partir das discussões do SO.

Capítulo 4

Recomendação de Soluções presentes em Sítios Q&A para Tarefas que Envolvem Uso de API

A segunda estratégia de recomendação também faz uso do conhecimento disponível no SO. Todavia, esta estratégia recebe como entrada uma consulta feita pelo usuário em linguagem natural e recomenda pares de pergunta-resposta (i.e. pares Q&A) da categoria *How-to-do* que podem ser úteis para auxiliar os desenvolvedores durante as tarefas de programação com uma dada API. Assim, esta estratégia de recomendação habilita os desenvolvedores a fazer algum progresso em suas tarefas de programação.

O SO possui no seu *website* um motor de busca que permite aos usuários realizar consultas informando um conteúdo textual (ex. "como mudar a cor de uma *label* utilizando a API *Swing*"). O resultado desta consulta é um conjunto de discussões (*threads*), cada uma composta por uma pergunta e uma série de respostas. Os usuários podem ordenar as *threads* conforme um critério como: o número de votos que a pergunta recebeu ou a relevância com a consulta. Entretanto, considerar somente um critério pode trazer algum problema para o usuário, como por exemplo: a consulta pode retornar *threads* que, apesar de serem relevantes com a consulta, possuem avaliação negativa pela comunidade, ou pode retornar *threads*, que apesar de serem bem votadas pela comunidade, não são relevantes para a consulta do desenvolvedor. Portanto, considerar mais de um critério parece ser mais apropriado em uma estratégia de recomendação.

Nesta segunda estratégia de recomendação, os pares Q&A são recomendados considerando três aspectos. O primeiro considera a similaridade textual que o par Q&A possui com a tarefa que o desenvolvedor possui em mãos. O segundo aspecto considera o *score* das perguntas e respostas para recomendar pares Q&A que foram bem avaliados pela comunidade do SO. Um par Q&A é composto por uma pergunta e uma resposta para esta pergunta. Por exemplo, se um *post* do SO possui n respostas, existem n possíveis pares Q&A para aquele *post*. O terceiro aspecto está relacionado com a classificação do par

Q&A: a abordagem visa recomendar somente soluções para tarefas de desenvolvimento, as quais podem ser modeladas como *"posts How-to"*. Tais *posts* serão filtrados por um algoritmo de classificação. O resultado do processo de recomendação é uma lista ranqueada de pares Q&A. A Seção 4.1 descreve os passos realizados para a construção de um classificador utilizando Aprendizado de Máquina Supervisionado.

4.1 Classificação de Pares Q&A

Os usuários perguntam diferentes tipos de perguntas no SO. De acordo com Nasehi *et al.* (NASEHI et al., 2012a), "Os tipos de perguntas do SO podem ser descritos baseados em duas dimensões diferentes. A primeira dimensão está relacionada com o tópico da pergunta: ele mostra a principal tecnologia ou conceito sobre o qual a pergunta gira em torno e geralmente pode ser identificado através das *tags* da pergunta que o usuário pode adicionar com o objetivo de ajudar outros (ex. potenciais respondentes) a descobrir o tema central da pergunta". Por exemplo, se o objetivo for recomendar pares Q&A para o tópico *Swing*, a abordagem proposta somente considera os pares Q&A pertencentes às *threads* de discussão em que a pergunta possui a *tag* "swing" entre suas *tags* (uma pergunta no SO pode ter até cinco *tags*). Ainda de acordo com Nasehi *et al.* (NASEHI et al., 2012a), "As perguntas do SO podem ser classificadas em uma segunda dimensão que é sobre as principais preocupações dos questionadores e o que eles querem resolver no SO". Nesta segunda dimensão, foram identificadas quatro categorias de perguntas: *Debug-corrective*, *Need-to-know*, *How-to-do-it* e *Seeking-different-solution*. Similarmente, foram consideradas as seguintes cinco categorias nesta segunda dimensão:

- *How-to-do*: O questionador provê um cenário e uma pergunta sobre como implementá-lo (algumas vezes informando uma dada tecnologia ou API). Exemplo de *post* no SO [1];

- *Conceptual*: Perguntas conceituais em um tópico particular (ex. definição de conceitos, melhores práticas para uma dada tecnologia). O questionador está esperando por uma explicação de um assunto particular ou justificação sobre um determinado comportamento. Exemplo de *post* no SO [2];

- *Seeking-something*: O questionador está procurando por alguma coisa mais objetiva (ex. um livro, um tutorial, um *framework*, uma biblioteca) ou mais subjetivo (ex. um conselho, uma opinião, uma sugestão, uma recomendação). Exemplo de *post* no SO [3];

[1] <https://stackoverflow.com/questions/4556313/how-do-i-add-a-click-handler-to-the-gwt-buttoncell>
[2] <https://stackoverflow.com/questions/3373854/what-is-a-memory-leak>
[3] <https://stackoverflow.com/questions/4664798/android-book-for-development>

- *Debug-corrective*: Perguntas que lidam com problemas no código de desenvolvimento, como erros em tempo de execução, notificações ou comportamento imprevisível. O questionador normalmente procura por uma revisão no seu código. Exemplo de *post* no SO [4];

- *Miscellaneous*: O questionador possui muitos interesses diferentes. Então, ele faz várias perguntas. Isto normalmente leva a uma mistura entre as outras categorias (ex. o questionador pode estar procurando por um livro e querer uma receita para um problema). Exemplo de *post* no SO [5].

As categorias *Need-to-know* e *Seeking-different-solution* propostas por Nasehi *et al.* (NASEHI et al., 2012a) correspondem respectivamente às categorias *Conceptual* e *Seeking-something* com alguns ajustes propostos no presente trabalho. A categoria *How-to-do* está bem próxima do cenário em que um desenvolvedor possui uma tarefa de programação em mãos e precisa resolvê-la. Por esta razão, a abordagem proposta visa filtrar somente os pares Q&A que são classificados como *How-to-do*. Com o intuito de automatizar a seleção desta categoria de par Q&A, desenvolveu-se um classificador, que será apresentado no restante desta seção.

4.1.1 Algoritmo de Classificação

Com o objetivo de escolher o algoritmo de classificação que melhor classifica os pares Q&A do SO, realizou-se uma comparação entre os seguintes classificadores: *Logistic Regression* (LR) (POHAR; BLAS; TURK, 2004; CESSIE; HOUWELINGEN, 1992), *Naïve Bayes* (NB) (LINARES-VáSQUEZ et al., 2014), *Multi-Layer Perceptron* (MLP) (HAYKIN, 1998), *Support Vector Machine* (SVM) (LINARES-VáSQUEZ et al., 2014), *J4.8 Decision Tree* (J4.8) (LINARES-VáSQUEZ et al., 2014; SEHGAL; MOHAN; SANDHU, 2012), *Random Forest* (RF) (LEMPITSKY et al., 2009) e *K-Nearest Neighbors* (KNN) (LINARES-VáSQUEZ et al., 2014).

Decidiu-se por classificar o par Q&A ao invés de classificar somente o corpo da pergunta pois observou-se que o corpo da resposta pode prover informações relevantes para ajudar o classificador na tomada de decisão sobre a categoria do par Q&A (ex. distinguir entre pares das categorias *How-to-do* e *Debug-corrective*).

4.1.2 SO Data set

Realizou-se o *download* de uma versão pública do banco de dados do SO (versão de Março de 2013) e os dados foram importados dentro de um banco de dados relacional

[4] <https://stackoverflow.com/questions/9230961/unexpected-output-while-converting-a-string-to-date-in-java>
[5] <https://stackoverflow.com/questions/937313/fling-gesture-detection-on-grid-layout>

com o objetivo de classificar os pares Q&A. A tabela "*posts*" deste banco de dados armazena todas as perguntas e respostas que foram dadas para cada pergunta, caso exista, considerando a data em que o *dump* foi realizado.

Foram selecionados aleatoriamente deste banco de dados relacional um conjunto de 400 pares Q&A para serem classificados manualmente. No processo de classificação de cada par, foram consideradas as cinco categorias descritas acima: *How-to-do*, *Conceptual*, *Seeking-something*, *Debug-corrective* e *Miscellaneous*. É importante observar que esses 400 pares Q&A estavam relacionados a diferentes tecnologias e distribuídos entre as cinco categorias mencionadas acima. O objetivo disso foi evitar o problema de sobreajuste (do inglês: *overfitting*) do modelo (DIETTERICH, 1995) ou viés em relação às categorias de pares mencionadas. Um modelo estatístico sobre-ajustado apresenta alta precisão quando testado com seu conjunto de dados, porém tal modelo apresenta baixa precisão quando testado com outros dados, uma vez que perde a capacidade de generalização.

A Tabela 6 mostra os resultados da classificação manual realizada nos 400 pares Q&A selecionados. Dois sujeitos humanos pesquisadores em Engenharia de *Software* e com experiência de 3 anos na indústria de *software* (*Autor A* e *Autor B*) realizaram uma classificação manual destes pares. Primeiro, cada um dos avaliadores realizou uma avaliação individual de todos os pares Q&A selecionados. Depois, uma avaliação consensual foi realizada com o objetivo de resolver os conflitos e decidir pela melhor classificação. Destes 400 pares Q&A, 74 (18,5%) tiveram que ser rediscutidos com o objetivo de resolver os conflitos entre os avaliadores. A principal dificuldade identificada neste processo de classificação foi diferenciar entre as categorias *How-to-do* e *Debug-corrective*. A diferença entre estas categorias é sutil. Ambas possuem trechos de código na pergunta. Quando uma pergunta com trecho de código é postada no SO, o questionador pode estar querendo basicamente uma destas duas soluções: (i) uma correção de *bug* para o trecho de código de entrada (no caso da categoria *Debug-corrective*) ou (ii) um exemplo de código que ensina a realizar uma dada tarefa de programação (no caso da categoria *How-to-do*).

Tabela 6 – Classificação manual dos 400 pares Q&A selecionados.

Categoria	#Pares Q&A Classificados
How-to-do	109
Conceptual	106
Seeking-something	121
Debug-corrective	10
Miscellaneous	54

Como o número de pares Q&A da categoria *Debug-corrective* foi somente 10, esta categoria não foi considerada na construção do *data set* de treinamento do classificador. Além disso, nenhuma aplicação prática foi encontrada para a categoria *Miscellaneous*. Desta forma, esta última categoria também não foi considerada na construção deste *data set*.

Após esta etapa de classificação manual, gerou-se um arquivo ARFF (*Attribute-Relation File Format*), contendo as instâncias rotuladas e a informação dos atributos do classificador. Este arquivo foi carregado posteriormente na interface gráfica do *software* WEKA (HALL et al., 2009). Assim, realizou-se um estudo experimental com 336 pares Q&A do SO rotulados em três categorias de domínio: *How-to-do*, *Conceptual* e *Seeking-something*. Os experimentos foram realizados utilizando a técnica *10-fold cross validation* (KOHAVI, 1995) para avaliar a capacidade de generalização de um classificador, a partir de um conjunto de dados. O *data set* utilizado neste trabalho está disponível *online* [6].

4.1.3 Definição dos Atributos

Foram definidos 10 atributos para caracterizar os pares Q&A. Destes 10 atributos, 6 estão relacionados com o número de ocorrências de palavras-chave no "título", "corpo da pergunta" e "corpo da resposta" de um par Q&A. Os 4 atributos restantes são booleanos e estão relacionados com a presença ou ausência de código-fonte ou *links* no "corpo da pergunta" e "corpo da resposta" para um dado par. Os atributos baseados em palavras-chave considerados são mostrados na Tabela 7 enquanto os atributos booleanos são mostrados na Tabela 8.

As palavras-chave utilizadas para classificação de perguntas *How-to-do* mostradas na Tabela 7 são o resultado de uma análise qualitativa manual de uma amostra de 100 *posts* do SO. Esta amostra foi selecionada aleatoriamente e o *Autor A* identificou as palavras e expressões frequentes que apareciam nesta amostra de *posts*. Esta análise foi realizada com o objetivo de definir os atributos baseados em palavras-chave mostrados na Tabela 7.

Decidiu-se por não utilizar *stemming* nos atributos baseados em palavras-chave do classificador pois o significado das palavras é bastante importante para auxiliar o classificador no processo de decisão da categoria do par Q&A. Apesar do processo de *stemming* ter suas vantagens uma vez que esta técnica transforma diferentes inflexões e derivações de uma mesma palavra para um tronco comum (do inglês, *stem*), um problema em relação ao *stemming* é o problema de *overstemming*. Este problema ocorre quando palavras não-relacionadas são reduzidas para o mesmo tronco (KARAA; GRIBâA, 2013). Por exemplo, as palavras *"general"* (relacionada com o atributo **optimalQty**) e "generator" (relacionada com o atributo **lookingForQty**) são stemizadas sob o mesmo tronco "gener". Logo, o classificador irá tratar estas palavras como sendo a mesma coisa, perdendo a noção do significado destas palavras. Embora o algoritmo de *Porter* (i.e. *Porter stemmer* (PORTER, 1997)) é conhecido por ser poderoso, ele ainda enfrenta muitos problemas. Os principais são erros de *overstemming* ou *understemming* (KARAA; GRIBâA, 2013).

Com o objetivo de identificar as palavras frequentes ou expressões que aparecem nos *posts* do SO, construiu-se um índice invertido (o primeiro conceito importante na área

[6] <https://github.com/eduardocunha11/How-to-Classifier> (acessado e verificado em 25/04/2019)

Tabela 7 – Atributos e suas respectivas palavras-chave.

Atributo	Palavras-chave
howQty	***how to***, ***how do***, ***how does***, ***how can***, ***how i***, ***how we***, ***how you***, ***way(s)***, *method(s), function(s), algorithm(s), anyway, manner, mode, solution(s), pseudocode, script(s), workaround, solve, resolve, implement, step(s), approach, approaches*
debugQty	*exception(s), error(s), debug, debugging, fail, failed, warning, notice, notification, fault, problem, matter, trouble, wrong, incorrect, denied, breakpoint, unhandled, fix, bug(s), issue(s), tracker(s), permission(s), bug/feature*
optimalQty	*optimal, efficient, better, reliable, elegant, general, appropriate, suitable, adequate, proper, safest, fast, fastest, quickly, security, secure, robust, performant, performance, reasonably, smoother, viable, fast, lightweight, easy, easiest, cleanest, small, open-source, user-friendly, good, portable, correct, standard*
lookingForQty	***tool(s)***, ***tutorial(s)***, ***manual(s)***, ***book(s)***, ***looking for***, ***looking forward***, ***looking at***, ***looking around***, ***searching for***, ***searching forward***, ***searching at***, ***searching around***, *client(s), find, app, application(s), lib(s), library, libraries, framework(s), migrate, migration(s), migrating, upgrade, convert, converting, conversion, porting, article(s), where, freeware, plugin(s), plug-in, research, search, seeking, google, system(s), video(s), resource(s), technique(s), editor(s), cms, erp, vmware, vpn, strategy, getting, started, when, ide(s), scanning, googling, blog(s), debugger(s), interpreter(s), compiler(s), comment(s), suggestion(s), looked, look, software(s), platform, profiler(s), generator(s), repository, repositories, should, advice(s), experience(s), experienced, used, replacement, idea(s), caveats, tips, tricks, recommend, recommendation, guideline(s), guide(s), guidance, orientation(s), help, helpful, suggest, suggestion(s), opinion(s), hint(s), point, pointers, experience(s), alternative(s), choice(s), thought(s), option(s), share, clue(s), insight, light, deal, dealt, package(s), available, threshold(s), freeware, direction(s), free, learning, material, beginner, possibilities, provider(s)*
conceptualQty	***difference(s) between***, ***is the***, ***is this***, ***are the***, ***are this***, ***best practice(s)***, ***why***, ***explain***, ***clarify***, ***explicate***, ***explanation***, ***explain***, *meaning, significance, possible, what, what's, which, elucidate, illuminate, expound, tell, how much, how many, missing, level, metrics, statistics, reason, cause(s), justification(s), potential, concept, distinction(s), consensus, motive(s), mean, signify, signification, lesson(s), understand, explanatory, purpose(s), does, conceptual*
questionQty	Retorna o número de perguntas feitas no par Q&A, baseado no número de vezes que o ponto de interrogação (?) aparece

Tabela 8 – Definição de atributos booleanos.

Atributo	Definição
questionHasCode	Valor booleano que indica se existe código-fonte na pergunta.
answerHasCode	Valor booleano que indica se existe código-fonte na resposta.
questionHasLink	Valor booleano que indica se existe *link(s)* na pergunta.
answerHasLink	Valor booleano que indica se existe *link(s)* na resposta.

de Recuperação da Informação). Cada *post* do SO possui um único identificador (*postID*). Com este índice, criou-se um dicionário de termos (às vezes também referido como vocabulário). Decidiu-se por considerar as palavras de parada (do inglês, *stop words*) neste vocabulário pelo fato que elas ajudam a identificar as principais preocupações dos questionadores e o que eles querem resolver no SO. Assim, para cada termo (palavra ou expressão), criou-se uma lista que armazena quais *posts* do SO em que o termo aparece

(i.e. cada termo está mapeado a uma lista de *postIDs*). Visto que um termo geralmente aparece em vários *posts*, esta organização de dados reduz os requisitos de armazenamento do índice.

O dicionário também armazena algumas estatísticas, como o número de *posts* do SO que contém cada termo (i.e. a frequência do *post* p_f, que também é o tamanho de cada lista). Definiu-se que um termo é frequente no dicionário quando $p_f \geq 30$ (i.e. o termo aparece no mínimo em 30 de 100 *posts* do SO considerados). Testou-se com diferentes valores para p_f, mas a acurácia do classificador *How-to* foi muito menor. Portanto, decidiu-se por considerar este limiar (do inglês, *threshold*) pois obteve-se uma melhor taxa de acerto para o classificador *How-to* com esta opção.

Para cada atributo relacionado com as palavras-chave, foram anotadas as palavras e expressões que apareciam com maior frequência nos *posts* do SO. Observou-se que a presença de código-fonte na pergunta frequentemente ocorre com pares Q&A pertencentes às categorias *How-to-do* e *Debug-corrective* pois o questionador coloca o código-fonte na pergunta com o objetivo de obter uma correção/solução para seu problema. Observou-se que frequentemente algumas respostas dos pares Q&A pertencentes às categorias *How-to-do* e *Seeking-something* possuíam *links* para fontes externas (ex. tutoriais, documentações oficiais de APIs). Estes *links* frequentemente continham informação adicional sobre a tarefa de programação em mãos. Portanto, considerou-se no experimento a informação contida dentro dos *links*. Para isso, adotou-se os mesmos critérios de avaliação de um par Q&A considerados neste capítulo.

As palavras-chave em negrito na Tabela 7 possuem peso 5 enquanto as demais possuem peso 1. Adotou-se um critério de ponderação pois existem palavras-chave que parecem ser mais importantes do que outras (i.e. algumas palavras-chave ajudam melhor no processo de decisão da categoria do par Q&A). Assim, a presença de uma palavra-chave aumenta em uma unidade ou cinco unidades o valor de um atributo. Diferentes pesos foram testados e escolheu-se aqueles que apresentaram melhores resultados (i.e. o peso em que os classificadores obtiveram as maiores taxas de acerto). Foram conduzidos experimentos com os pares Q&A rotulados utilizados para construir o *data set* de treinamento do classificador descrito acima na Subseção 4.1.2. Mais especificamente, este *data set* contém 336 pares Q&A rotulados divididos em três categorias de domínio: *How-to-do*, *Conceptual* e *Seeking-something*. Processou-se os textos em linguagem natural presentes nas perguntas e respostas daqueles pares Q&A considerando diferentes mecanismos de peso para as palavras-chave relevantes, e para cada mecanismo foi gerado um arquivo ARFF diferente (contendo as instâncias rotuladas e as informações dos atributos) que foi posteriormente carregado na interface gráfica do *WEKA* (HALL et al., 2009). Os experimentos foram realizados utilizando uma técnica de validação cruzada denominada *k-fold* ($k = 10$ *folds*), considerando-se os 5 atributos mais relevantes.

A Tabela 9 mostra as taxas de acertos de diferentes classificadores e mecanismos de

peso experimentados. Como mostrado na Tabela 9 as maiores taxas de acerto foram atingidas com peso 5 (i.e. coluna Peso = 5) para as palavras consideradas mais relevantes no processo de decisão da categoria do par Q&A (i.e. as palavras-chave que aparecem em negrito na Tabela 7). Para todos os mecanismos de peso experimentados, considerou-se que o peso para as palavras-chave menos relevantes é igual a 1 (i.e. as palavras-chave que não aparecem em negrito na Tabela 7).

Tabela 9 – Taxas de Acerto para diferentes classificadores e mecanismos de peso considerando os 5 atributos mais relevantes (Peso: o peso para as palavras consideradas mais relevantes; AC: acurácia do classificador).

Acurácia	Peso = 1	Peso = 2	Peso = 3	Peso = 4	Peso = 5	Peso = 6
AC(LR)	65,4223%	66,6714%	71,1315%	74,2140%	76,1905%	75,2210%
AC(NB)	61,2917%	62,3345%	68,9127%	69,8113%	72,9167%	70,2357%
AC(MLP)	64,2140%	66,7132%	71,2840%	73,2546%	75,8929%	74,1018%
AC(SVM)	59,1439%	59,7890%	66,2453%	68,4562%	70,2381%	69,4627%
AC(J4.8)	58,4567%	58,9214%	63,2718%	67,3321%	69,6429%	68,3492%
AC(RF)	60,2143%	62,7716%	65,9122%	68,1014%	71,7262%	70,2910%
AC(KNN) (k = 5)	58,8570%	60,4310%	63,2451%	65,3160%	69,9405%	67,2314%

Portanto, cada palavra-chave possui seu próprio peso (um ou cinco). Palavras-chave com peso igual a 1 são consideradas menos importantes e são contadas somente uma vez, enquanto as palavras-chave com peso igual a 5 são consideradas mais importantes e são contadas 5 vezes quando elas aparecem no par Q&A. Além disso, observou-se que palavras-chave com peso igual a 5 aparecem frequentemente em diferentes pares Q&A e descrevem claramente a preocupação do questionador ao realizar uma pergunta no SO. A próxima Subseção detalha o processo de Seleção de Características (do inglês, *Feature Selection*) conduzido com o intuito de filtrar os atributos mais relevantes e apresenta os resultados de classificação que foram obtidos.

4.1.4 Seleção de Características e Resultados de Classificação

Realizou-se uma seleção de características utilizando o filtro de Ganho de Informação (YANG; PEDERSEN, 1997; FORMAN, 2003) para reduzir o espaço de características e eliminar os atributos menos relevantes. O processo de filtragem selecionou seis atributos: *conceptualQty, lookingForQty, howQty, answerHasCode, questionHasCode, answerHasLink*.

Utilizou-se o filtro de Ganho de Informação nos atributos em combinação com o método de pesquisa *Ranker* [7] (i.e. este método efetua o ranqueamento dos atributos considerando as avaliações individuais deles) considerando um *threshold* [8]. Este *threshold* é o valor *default* que o *software Weka* adota para este filtro.

[7] http://weka.sourceforge.net/doc.dev/weka/attributeSelection/Ranker.html
[8] *threshold* = -1.7976931348623157E308

A Tabela 10 mostra os atributos em ordem decrescente de valor de Ganho de Informação (quanto maior o valor de Ganho de Informação, melhor o atributo contribui para o processo de classificação).

Tabela 10 – Atributos Ranqueados: Ganho de Informação

Atributo	Valor de Ganho de Informação
conceptualQty	0,3309
lookingForQty	0,2486
howQty	0,2211
answerHasCode	0,0757
questionHasCode	0,0389
answerHasLink	0,029
debugQty	0
questionHasLink	0
optimalQty	0
questionQty	0

A Tabela 11 mostra os resultados de classificação para os top-5 e top-6 atributos mais relevantes selecionados pelo filtro Ganho de Informação. Obteve-se melhores resultados com os 5 atributos mais relevantes.

Tabela 11 – Resultados com os top-5 e top-6 atributos mais relevantes selecionados.

Classificador	top-5 atributos Taxa de Acerto	top-6 atributos Taxa de Acerto
LR	76,1905%	74,4048%
NB	72,9167%	70,8333%
MLP	75,8929%	72,3214%
SVM	70,2381%	70,5357%
J4.8	69,6429%	69,6429%
RF	71,7262%	74,7024%
KNN (k = 5)	69,9405%	69,3452%

O processo de classificação pode ser realizado em qualquer computador *desktop* atual executando o *software* WEKA. O processo não requer recursos de memória e/ou processamento extravagantes. O maior custo está na criação do arquivo de entrada para o classificador (i.e. arquivo ARFF) e não no processo de classificação propriamente dito. Este arquivo de entrada contém informação sobre as características dos pares Q&A pertencentes a uma API particular. O processo de classificação todo para uma dada API foi executado em menos de uma hora.

Os melhores resultados foram obtidos com um classificador *Logistic Regression* (LR) que atingiu uma taxa geral de acerto de 76,19% nas 3 categorias (*How-to-do*, *Conceptual* e *Seeking-something*) e de 79,81% na categoria *How-to-do*. Obteve-se também uma taxa geral de acerto de 75,89% com um classificador *Multi-Layer Perceptron* (MLP). Com o objetivo de responder se os classificadores LR e MLP são significativamente diferentes, utilizou-se um teste não-paramétrico denominado teste de *Mc Nemar* (MCNEMAR,

1947)(BOSTANCI; BOSTANCI, 2013). De acordo com o teste de *Mc Nemar*, dois algoritmos de classificação podem ter 4 possíveis resultados dispostos em uma Tabela de Contingência 2x2 (LIDDELL, 1976) como mostrado na Tabela 12.

Tabela 12 – Teste de *Mc Nemar*: Possíveis resultados de dois algoritmos.

	Algoritmo A falhou	Algoritmo A bem-sucedido
Algoritmo B falhou	N_{ff}	N_{sf}
Algoritmo B bem-sucedido	N_{fs}	N_{ss}

N_{ff} denota o número de instâncias em que ambos os algoritmos de classificação falharam e N_{ss} denota o número de instâncias em que ambos os algoritmos de classificação foram bem-sucedidos. Estes dois casos não dão muita informação sobre os desempenhos dos algoritmos, uma vez que não indicam como os seus desempenhos diferem. Entretanto, os outros dois parâmetros (N_{fs} e N_{sf}) mostram os casos em que um dos algoritmos falham e o outro foi bem-sucedido indicando as discrepâncias de desempenho. Com o objetivo de quantificar estas diferenças, o teste de *Mc Nemar* emprega a pontuação v (Equação 1) (BOSTANCI; BOSTANCI, 2013).

$$v = \frac{(|N_{sf} - N_{fs}| - 1)}{\sqrt{N_{sf} + N_{fs}}} \tag{1}$$

A pontuação v é interpretada como se segue: quando $v = 0$, os dois algoritmos mostram desempenho semelhante. À medida que este valor diverge de 0 em direção positiva, isto significa que o desempenho deles difere significativamente (BOSTANCI; BOSTANCI, 2013). Utilizou-se o Teste de *Mc Nemar* para os algoritmos LR e MLP considerando os melhores resultados de classificação que foram obtidos (i.e. com os 5 atributos mais relevantes, as taxas de acerto global para LR e MLP foram respectivamente, 76,1905% and 75,8929% em um total de 336 instâncias). A Tabela 13 mostra a tabela de contingência 2x2 para os algoritmos de classificação LR e MLP. Obteve-se uma pontuação v igual a 0. Portanto, os classificadores LR e MLP não são significativamente diferentes. De acordo com o Teste de *Mc Nemar*, estes classificadores possuem desempenho similar. Desta forma, neste caso pode-se escolher qualquer um dos dois algoritmos pois não seria uma grande diferença se adotarmos um classificador com uma taxa de acerto ligeiramente inferior. Neste estudo, adotou-se o classificador LR para selecionar pares Q&A da categoria *How-to-do*.

Tabela 13 – Teste de *Mc Nemar*: tabela de contingência para os algoritmos LR e MLP ($v = 0$).

	Algoritmo LR falhou	Algoritmo LR bem-sucedido
Algoritmo MLP falhou	$N_{ff} = 80$	$N_{sf} = 1$
Algoritmo MLP bem-sucedido	$N_{fs} = 0$	$N_{ss} = 255$

Os resultados finais da estratégia de recomendação proposta neste trabalho de doutorado dependem diretamente de uma boa classificação. Por exemplo, se um classificador possui baixa taxa de acerto, o mesmo irá identificar incorretamente um grande número de pares Q&A como sendo da categoria *How-to-do*. É essencial classificar corretamente pares Q&A *How-to-do* pois eles possuem a característica de mostrar passo a passo como resolver uma tarefa de programação em mãos. A etapa de classificação também é importante para melhorar a qualidade geral do ranqueamento. Não seria interessante utilizar outro classificador neste trabalho pois até o momento não foi obtido um classificador com maior acurácia do que o LR. Somente o classificador MLP atingiu um desempenho similar (de acordo com o Teste de *Mc Nemar* realizado entre o par de classificadores LR e MLP). Portanto, a qualidade geral do ranqueamento gerado por eles seria muito semelhante.

4.2 Abordagem de Recomendação Proposta

Esta seção apresenta o objetivo de pesquisa, os três tópicos utilizados nos experimentos e detalha a abordagem de recomendação de pares Q&A *How-to-do* provenientes do SO.

4.2.1 Objetivo de Pesquisa

O objetivo da abordagem de recomendação proposta neste trabalho é recomendar pares Q&A para auxiliar os desenvolvedores nas suas tarefas de desenvolvimento, considerando que os pares Q&A recomendados devem: (1) ter alta similaridade textual com a tarefa de desenvolvimento, (2) ser bem avaliados pela comunidade do SO e (3) ser caracterizados como *How-to-do*. Um *post* do SO é formado por uma pergunta e uma série de respostas para esta pergunta. Decidiu-se recomendar pares Q&A ao invés de *posts* inteiros pelo fato de que as respostas para uma mesma pergunta podem ter diferentes pontuações, i.e. algumas respostas podem ser muito melhores do que as outras. A intenção é somente recomendar conteúdo de alta qualidade presente no SO. Considera-se que a pontuação de um par Q&A (i.e. o número de votos positivos menos o número de votos negativos) é um indicativo da sua qualidade, uma vez que esta pontuação representa a avaliação realizada pela comunidade. Além disso, espera-se recomendar pares Q&A que sejam altamente relevantes no contexto da tarefa do usuário.

4.2.2 Tópicos considerados

Foram conduzidos experimentos em três tópicos para diferentes linguagens de programação (Java, C++ e C#) amplamente utilizadas na indústria de desenvolvimento de *software*: *Swing*, *Boost* e *LINQ*, respectivamente.

Swing é um *kit* de ferramentas que habilita a criação de Interfaces Gráficas de Usuário em Java. Os desenvolvedores podem usar *Swing* para criar aplicações em larga escala

com uma grande variedade de componentes poderosos (ECKSTEIN; LOY; WOOD, 1998; DARWIN, 2004).

Boost: É uma coleção de bibliotecas C++. Cada biblioteca foi revisada por muitos desenvolvedores profissionais antes de ser aceita nesta coleção. Estas bibliotecas foram testadas em várias plataformas utilizando muitos compiladores e as implementações da biblioteca padrão do C++ (POLUKHIN, 2013). O *Boost* fornece bibliotecas nas seguintes áreas: algoritmos, estrutura de dados, programação concorrente, programação genérica, grafos, ponteiros inteligentes, processamento de cadeias de caracteres, etc.

LINQ (*Language Integrated Query*): É um modelo de programação criado pela *Microsoft*, que adiciona recursos de consulta para as linguagens de programação .NET. Estas extensões fornecem uma sintaxe mais curta e expressiva para a manipulação de dados (HILYARD; TEILHET, 2007).

4.2.3 Construção do Índice

Utilizou-se o motor de busca *Apache Lucene* (BIALECKI; MUIR; INGERSOLL, 2012) para indexar os dados. Para um dado tópico (ex. *Swing*), foram obtidos todos os *posts* presentes no banco de dados local do SO cujas perguntas possuem uma *tag* específica (ex. *"swing"*). Após isso, a partir deste conjunto de *posts*, foram obtidos todos os pares Q&A. Por exemplo, se um *post* possui uma pergunta e n respostas, são gerados n pares Q&A para este *post*. A Tabela 14 mostra o número de pares Q&A obtidos para cada tópico considerado.

O próximo passo foi classificar os pares Q&A com o objetivo de considerar somente pares da categoria *How-to-do*. A Tabela 15 mostra o resultado da classificação destes pares. Para cada par Q&A classificado como *How-to-do*, foram removidas suas *tags* HTML e *stop words* e foi realizado o processo de *stemming* no seu conteúdo (textos do título, pergunta e resposta, excluindo os trechos de código) utilizando o algoritmo de *Stemming* definido por *Porter* (PORTER, 1997). Como as perguntas e respostas do SO podem possuir trechos de código-fonte que não são apropriados para serem processados pelo *parser* do *Lucene*, estes trechos de código-fonte foram tratados de maneira diferente.

Para a API *Swing*, foram desenvolvidas expressões regulares para obter os nomes das classes/interfaces/métodos que estão sendo criados ou invocados. Por exemplo, seja um método com o nome *"addActionListener"*. Como o padrão de codificação *CamelCase* é o padrão mais utilizado em Java, os nomes foram divididos em seus termos constituintes. Para o nome de método *"addActionListener"*, foram obtidos os seguintes termos: *"add"*, *"action"* e *"listener"*. Foram adicionados dentro do documento que corresponde ao par Q&A tanto o nome original quanto os seus termos constituintes, depois de remover as *stop words* e realizar o processo de *stemming* neles (no exemplo seriam adicionados ao documento que corresponde ao par Q&A os seguintes termos *"addActionListener"*, *"add"*, *"action"* e *"listener"*). A razão por adicionar no documento os nomes das classes/inter-

faces/métodos é devido ao fato de que o autor da pergunta pode estar querendo executar uma tarefa específica utilizando um elemento particular da API (ex. Como abrir um arquivo utilizando *JFileChooser*). Portanto, incluir esta informação no documento pode ajudar a abordagem a encontrar um par que é aderente aos termos.

Para a API *Boost*, foram desenvolvidas também expressões regulares visando identificar os nomes das classes ou nomes de métodos/*structs* que estão sendo criados ou invocados.

Para a API *LINQ*, foi feito algo diferente por não se tratar de uma API clássica: foi verificado se o trecho de código-fonte está utilizando algum dos seus operadores (ex. *"OrderByDescending"*, *"SelectMany"*) e um processo similar ao descrito para o *Swing* foi realizado no nome desses operadores. As expressões regulares utilizadas para a detecção de termos de código estão disponíveis *online* [9]. Foram criados 3 índices de busca (i.e. um índice de busca para cada API) utilizando o *Lucene*. Cada índice de busca é formado por um conjunto de documentos (*corpus*) referente a uma determinada API. Na próxima Subseção (ver Subseção 4.2.4), é apresentada a utilização deste índice para pesquisar pares Q&A. A Tabela 16 mostra o número de documentos utilizados para a construção do índice para cada API. Observe que o número de documentos é o mesmo que o número de pares Q&A classificados na categoria *"How-to-do"*. Isto é explicado pois para cada par Q&A classificado nesta categoria, é gerado um documento que irá compor o *corpus* utilizado para construir o índice de busca. A Tabela 16 também mostra o número de termos diferentes nos documentos para cada tópico.

Tabela 14 – Total de pares Q&A por tópico.

Tópico	Linguagem de Programação	Total de pares Q&A
Boost	C++	14.558
Swing	Java	45.239
LINQ	C#	60.035

Tabela 15 – Classificação de pares Q&A por tópico.

Tópico	*How-to-do*	*Conceptual*	*Seeking-something*
Boost	7.125	4.112	3.321
Swing	26.374	10.629	8.236
LINQ	39.592	13.962	6.481

Tabela 16 – Informação do índice por tópico.

Tópico	Número de documentos	Número de termos distintos
Boost	7.125	55.383
Swing	26.374	187.914
LINQ	39.592	263.502

[9] <https://github.com/eduardocunha11/How-to-Classifier> (acessado e verificado em 25/04/2019)

4.2.4 Consulta nos Índices Lucene

O índice *Lucene* construído para uma API pode ser utilizado para pesquisar pares Q&A que são relevantes para uma dada consulta. Foram realizados dois tipos de consultas neste índice que corresponde aos seguintes cenários: Cenário *NKAE - Not Known API Element* (Elemento da API não-conhecido) e Cenário *KAE - Known API Element* (Elemento da API conhecido).

O *Cenário NKAE* corresponde à situação em que o desenvolvedor possui uma tarefa de desenvolvimento em mãos, que deve ser solucionada utilizando uma API particular (ex. *"Boost"*), mas o desenvolvedor não sabe qual elemento da API (ex. classe ou método) pode ajudá-lo a resolver seu problema. Por exemplo, o desenvolvedor poderia precisar "ler um arquivo de texto utilizando *Boost*". O título da tarefa (neste exemplo, "ler um arquivo de texto utilizando *Boost*"), após ser pré-processado (i.e. após a remoção de *stop words* e a realização do processo de *stemming*), é utilizado como consulta para retornar os pares Q&A.

O *Cenário KAE* corresponde à situação em que o desenvolvedor utiliza um elemento particular de uma determinada API (ex. uma classe) para resolver uma tarefa de desenvolvimento. Por exemplo, algum desenvolvedor poderia precisar usar a API *Swing* para "mudar a cor de um *JButton*", onde *"JButton"* é a classe do *widget* da API *Swing*. Neste caso, o desenvolvedor já sabe qual elemento da API deve ser utilizado.

Foram consultados os pares Q&A no *Cenário KAE* da mesma maneira do que foi descrito para o *Cenário NKAE*. A diferença é que foi adicionado ao título da tarefa uma *string* correspondente ao nome da classe (no caso do *Swing* e *Boost*) ou um nome de operador (no caso do *LINQ*) que é crucial na solução apresentada no *cookbook* para aquele problema. As tarefas consideradas nos experimentos foram extraídas a partir de *cookbooks* relacionados a estas APIs. Por exemplo, para o *Swing*, uma das tarefas selecionadas para o experimento possuía o título *"Action Handling: Making Buttons Work"*. A classe *"ActionListener"* é importante na solução da tarefa. Portanto, adicionou-se a *string "ActionListener"* ao título da tarefa e a *string* resultante *"Action Handling: Making Buttons Work ActionListener"*, após ser pré-processada (i.e. após a remoção de *stop words* e a realização do processo de *stemming*), foi utilizada como entrada para a pesquisa no índice Lucene referente à API *Swing*.

O resultado da pesquisa no índice *Lucene* é uma lista ranqueada de documentos (i.e. pares Q&A), no qual o primeiro é o mais similar com a consulta e o último é o menos similar. Cada par Q&A no *ranking* possui um valor numérico (denominado *score Lucene*) que representa a sua similaridade com a consulta. Portanto, o primeiro par do *ranking* possui o maior *score Lucene* e o último possui o menor *score Lucene*.

4.2.5 Ranqueamento dos Pares Q&A pelo *Score* do SO

Ao utilizar o *ranking* retornado pela pesquisa no índice *Lucene*, são obtidos pares Q&A que possuem similaridade textual com a consulta, mas não é possível garantir nada sobre a qualidade destes pares, i.e. entre os pares Q&A retornados para a consulta, podem existir pares bem avaliados pela comunidade mas também pares que não foram bem avaliados pela comunidade do SO. O *score* de um *post* (pergunta ou resposta) tem sido usado como um *proxy* para avaliar sua qualidade pois o mecanismo de votação no SO é a principal forma que os membros possuem de avaliar o conteúdo do *site* ((PONZANELLI et al., 2014), (NASEHI et al., 2012a)).

Devido ao fato de que cada *post* individual no SO possui seu próprio *score* e que a estratégia de recomendação proposta irá sugerir pares Q&A, sendo que cada par é composto por uma pergunta e uma resposta para esta pergunta, definiu-se uma métrica para indicar a qualidade do par Q&A como um todo. Uma possível abordagem para atingir isso é considerar o valor médio do *score* da pergunta e o *score* da resposta de um par. Todavia, não consideramos esta abordagem pois a resposta parece ser mais importante do que a pergunta referente à ela. A razão para essa hipótese é que a resposta carrega mais informação sobre o problema. Logo, o *score* do par foi definido como sendo a média ponderada entre os *scores* individuais da sua resposta e pergunta. Obteve-se melhores resultados com os valores 0,7 e 0,3 para os pesos da resposta e pergunta de um par Q&A (Na Subseção 4.2.9 é apresentada uma justificativa formal para considerar estes pesos). Desta forma, é possível calcular o *score* do SO para um par Q&A da categoria *How-to-do* referente a uma determinada API.

A equação abaixo foi utilizada para o cálculo do *Score* de um par Q&A:

$$ScoreParQ\&A = (0,7 * ScoreResposta) + (0,3 * ScorePergunta); \quad (2)$$

Na próxima Subseção (ver Subseção 4.2.6), o *score* do SO de um par Q&A e o seu *score Lucene* serão combinados para construir um ranqueamento dos pares que será utilizado na estratégia de recomendação proposta.

4.2.6 Combinação dos *Scores* para Ranqueamento dos Pares Q&A

O *score Lucene* de um par indica o quanto ele é textualmente similar com uma dada consulta, enquanto o *score* do SO indica o quanto o par foi bem votado pela comunidade do *site*. Ambos os aspectos são considerados na estratégia de recomendação do trabalho proposto, uma vez que o objetivo é prover ao desenvolvedor, pares Q&A que ao mesmo tempo, estejam relacionados com o problema de desenvolvimento em mãos e que tenham conteúdo de boa qualidade.

Para combinar as duas métricas em uma única medida, realizou-se uma etapa de normalização. A técnica de normalização consiste basicamente em colocar o valor do *score Lucene* e do *score* do SO no intervalo de [0,1]. A razão para essa etapa de normalização é que geralmente o *score* do SO é muito maior que o *score Lucene*, i.e. estas métricas possuem naturezas diferentes. Depois desta etapa de normalização, calculou-se a média aritmética para cada par Q&A. O valor desta média é o *score* final e foi utilizado para ranquear os pares Q&A em ordem decrescente. Os 10 primeiros pares deste *ranking* foram recomendados para o usuário que pesquisou no sistema de recomendação. Este valor pode ser configurado através do sistema proposto.

A Figura 4 mostra uma visão geral da abordagem de recomendação proposta neste capítulo cujo objetivo é recomendar soluções presentes no SO para tarefas de programação envolvendo APIs.

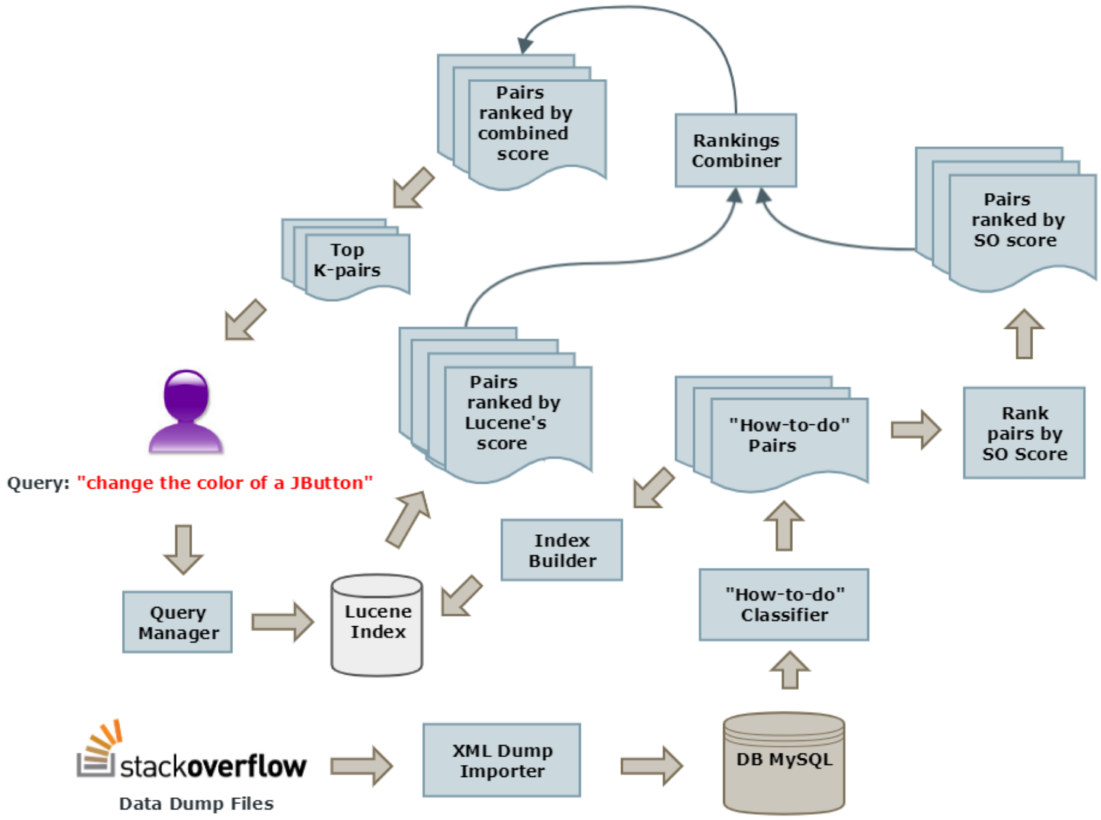

Figura 4 – Visão geral da abordagem de recomendação proposta: *Lucene+Score+How-to*.

4.2.7 Critérios de Avaliação

Nesta subseção são apresentados os critérios utilizados para realizar a avaliação qualitativa de cada par Q&A recomendado pela abordagem proposta.

O primeiro critério é chamado **Relevância** (*Relev*). Este critério é utilizado para verificar com qual extensão a informação contida no par Q&A pode ser utilizada para

ajudar o desenvolvedor a resolver a tarefa de programação que foi pesquisada no sistema. Para saber se o par Q&A é relevante ou não, será comparada a solução dada no par Q&A com a solução apresentada no *cookbook*. A nota dada neste critério varia de 0 a 4. O valor 0 significa que o par Q&A recomendado não está relacionado com a tarefa de programação pesquisada. Já o valor 4 significa que a informação contida no par Q&A pode ser utilizada para resolver completamente a tarefa de programação pesquisada. Esta métrica não é *booleana* pois algumas vezes a informação contida no par Q&A pode ser utilizada para resolver parcialmente uma dada tarefa de programação: o valor 3 significa que a informação contida no par Q&A pode ser utilizada para resolver quase toda a tarefa de programação pesquisada. O valor 2 significa que apenas parte da tarefa de programação pesquisada pode ser resolvida com a informação presente no par Q&A. Finalmente, o valor 1 significa que a informação contida no par Q&A está relacionada com a tarefa de programação pesquisada porém não contribui para a sua resolução.

O segundo critério é chamado **Reprodutibilidade** (*Reprod*). Este critério é utilizado para avaliar com qual extensão os trechos de código-fonte disponíveis nos corpos das perguntas e respostas de um par Q&A recomendado podem ser facilmente compilados e executados. Enquanto o critério *Relev* possui um aspecto semântico, i.e. seu principal objetivo é verificar se uma tarefa de programação pode ser solucionada utilizando a informação recomendada, *Reprod* é uma métrica sintática pois este critério avalia o quão fácil os trechos de código-fonte podem ser compilados e executados. É altamente desejável que os trechos de código-fonte sejam executados isoladamente porque os desenvolvedores frequentemente usam exemplos de código-fonte como base para interagir com uma API (HOLMES et al., 2009). Portanto, eles precisam de exemplos de uso de alta qualidade com o intuito de aprender como utilizar uma API desejada. Um estudo revelou que o maior obstáculo para a aprendizagem de uma API na prática são exemplos de uso "insuficientes ou inadequados" (Robillard, 2009). Subramanian *et al.* (SUBRAMANIAN; HOLMES, 2013) analisaram 39.000 trechos de código presentes nas respostas de perguntas do SO e encontraram que somente 6.766 (17%) eram arquivos completos contendo declarações de classe e método, 6.302 (16%) dos trechos de código eram somente corpos de métodos sem declarações de classe e os 67% restantes eram trechos de código incompletos (i.e. a maioria não eram fragmentos de código compiláveis com declarações completas de classe e corpo de método). Como os trechos de código geralmente não são completos, a informação presente nestes trechos frequentemente não é suficiente para resolver acessos de método de API. Além disso, eles observaram que a maioria das respostas abrange os detalhes fornecidos na pergunta; por causa disso, alguns aspectos do trecho de código, como declarações de variáveis são muitas vezes ignorados (SUBRAMANIAN; HOLMES, 2013).

Com relação ao critério *Reprod*, foram avaliados também trechos de código que estavam presentes em fontes externas pois vários *sites* continham informação detalhada sobre como resolver uma tarefa de programação particular. Por exemplo, se a página que o *link*

representa continha um trecho de código-fonte completo, a resposta que referenciava o *link* era avaliada como "reproduzível". Enquanto o critério *Relev* possui um aspecto semântico, i.e. seu principal objetivo é verificar se a tarefa pode ser solucionada utilizando a informação recomendada, o critério *Reprod* é uma métrica sintática pois ele avalia a facilidade com que os trechos de código podem ser compilados e executados, independentemente se eles estão relacionados com a consulta realizada no sistema de recomendação proposto.

A nota dada ao critério *Reprod* também varia de 0 a 4. O valor 0 significa que o par Q&A recomendado não possui trechos de código-fonte. Já o valor 4 significa que os trechos de código-fonte podem ser facilmente compilados e executados principalmente sem nenhuma adaptação. Esta métrica também não é *booleana* pois em alguns casos, os pares Q&A possuem trechos de código-fonte que, embora não possam ser diretamente executados dentro da IDE, os mesmos podem ser compilados depois de alguns pequenos ajustes (ex. muitos trechos de código-fonte estão incompletos pois eles estão faltando declaração de variável, mas se elas forem declaradas, esses trechos se tornam completos e podem ser compilados). Para estes casos em que pequenos ajustes devem ser feitos, definiu-se o valor 3. Nos casos em que o trecho de código possui apenas uma linha de código independente é dado o valor 1. Finalmente, para os casos em que os trechos de código possuem uma ou mais linha(s) omitida(s), porém apresentam mais do que uma linha de código, é atribuído o valor 2.

4.2.8 Delineamento Experimental

Nesta subseção são apresentados os experimentos com as três APIs consideradas para avaliar se a estratégia de recomendação proposta pode ajudar os desenvolvedores nas suas tarefas de desenvolvimento. Foram consideradas um total de 35 tarefas de desenvolvimento: 12 para *Swing*, 12 para *Boost* e 11 para *LINQ*.

As tarefas do *Swing* foram retiradas do capítulo 13 do livro *Java Cookbook* (DARWIN, 2004), que contém somente tarefas relacionadas com GUI (*Graphical User Interfaces*). Existem 14 tarefas neste capítulo e foram selecionadas 12 delas.

As tarefas do *Boost* foram extraídas do livro *Boost Cookbook* (POLUKHIN, 2013). Foram selecionadas aleatoriamente 12 das 91 tarefas disponíveis neste *cookbook*.

As tarefas do *LINQ* foram retiradas de um *blog* [10] desenvolvido pela equipe de *Visual Basic* da *Microsoft*. Existem 12 tarefas neste *blog*, entretanto foram selecionadas somente 11 delas pois uma tarefa só tinha instruções de como configurar um banco de dados que foi utilizado em outras tarefas descritas no *blog* e portanto, não é apropriada para ser utilizada em um experimento para recomendar pares Q&A para *LINQ*, uma vez que ela está muito mais relacionada com uma área de banco de dados do que com *LINQ*.

[10] <http://blogs.msdn.com/b/vbteam/archive/tags/linq+cookbook/> (verificado em 26/04/2019)

A ideia de retirar essas atividades de *Cookbooks* é que os pares Q&A que são recomendados ao desenvolvedor são pertencentes à categoria *How-to-do* e possuem estrutura semelhante às receitas de um *Cookbook*, i.e. possuem o seguinte formato: título da receita, cenário do problema e solução para o problema.

Desta forma, é possível comparar se as soluções dadas aos problemas de programação se assemelham com as soluções apresentadas na receita do *Cookbook* para um determinado problema de programação. A ideia é verificar se os pares Q&A recomendados estão utilizando as mesmas classes e métodos da API que são usados na receita do *Cookbook* para um determinado problema de programação.

Uma análise manual das top-10 recomendações foi conduzida para todas as 35 tarefas de programação considerando os critérios *Relev* e *Reprod* representando 700 pontos de avaliação. Além disso, a mesma análise foi replicada utilizando-se o motor de busca do *Google* como base de comparação, representando mais 700 pontos de avaliação. O desenho da análise com o *Google* é apresentado na Subseção 4.2.10. A Tabela 17 mostra o desenho experimental. Para cada tópico (*Swing*, *Boost* e *LINQ*), realizou-se experimentos para testar o *Cenário NKAE* e o *Cenário KAE*. Para as 12 tarefas previamente selecionadas para o *Swing*, foram selecionadas aleatoriamente 6 para o *Cenário NKAE* e 6 para o *Cenário KAE*. O mesmo foi realizado para o *Boost*. Para as 11 tarefas do *LINQ*, foram selecionadas aleatoriamente 6 e 5 tarefas para os *Cenários NKAE e KAE*, respectivamente. A consulta de entrada para o *Cenário NKAE* foi o título das tarefas (depois do processo de *stemming* e remoção das *stop words*). Já a consulta de entrada para o *Cenário KAE* foi um texto formado pelo título da tarefa concatenado com o nome da classe (no caso do *Swing* ou *Boost*) ou nome do operador (no caso do *LINQ*) que foi importante na solução apresentada nos *cookbooks* originais dos quais as tarefas de programação foram extraídas.

A Tabela 18 mostra as 12 tarefas relacionadas com o tópico *Swing* selecionadas para o experimento. As primeiras 6 foram incluídas no *Cenário NKAE* e as 6 restantes foram incluídas no *Cenário KAE*. A numeração da tarefa na primeira coluna corresponde à numeração no documento original. As tarefas para o *Cenário KAE* já estão mostradas com seu título modificado. Por exemplo, o título original da tarefa 13.5 é *"Action Handling: Making Buttons Work"*. Na Tabela 18, o título desta tarefa é apresentado como *"Action Handling: Making Buttons Work ActionListener"*, pois *"ActionListener"* foi o nome da classe escolhido para ser acrescentado no título, uma vez que ela é a classe chave utilizada na solução daquela tarefa no *cookbook*. Após a remoção de *stop words* (caso tiver alguma) e a realização do processo de *stemming*, o texto resultante é utilizado como consulta de entrada para recuperar pares Q&A. Tabelas similares são mostradas para os tópicos *Boost* e *LINQ* (Tabelas 19 e 20, respectivamente). Nessas tabelas, os nomes das classes ou operadores utilizados para o *Cenário KAE* são mostrados em negrito.

Dois sujeitos humanos pesquisadores em Engenharia de *Software* e com experiência de 3 anos na indústria de *software* (*Autor A* e *Autor B*) individualmente avaliaram para

Tabela 17 – Desenho Experimental.

Tópico	Cenários	Tarefas	Critérios	Abordagem de Ranqueamento
Swing	Cenário NKAE	6 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*
	Cenário KAE	6 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*
Boost	Cenário NKAE	6 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*
	Cenário KAE	6 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*
LINQ	Cenário NKAE	6 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*
	Cenário KAE	5 tarefas	*Relev*	*Google* *Lucene+Score+How-to*
			Reprod	*Google* *Lucene+Score+How-to*

Tabela 18 – Tarefas relacionadas com o tópico *Swing*.

Tarefa	Cenário	Título da Tarefa
13.14	*NKAE*	*Program: Custom Font Chooser*
13.13	*NKAE*	*Changing a Swing Program's Look and Feel*
13.11	*NKAE*	*Choosing a Color from all the colors available on your computer*
13.3	*NKAE*	*Designing a Window Layout*
13.1	*NKAE*	*Choosing a File*
13.8	*NKAE*	*Dialogs: When Later Just Won't Do*
13.12	*KAE*	*Centering a Main Window* ***JFrame***
13.2	*KAE*	*Adding and Displaying GUI Components to a window* ***JFrame***
13.9	*KAE*	*Getting Program Output into a Window* ***PipedInputStream***
13.4	*KAE*	*A Tabbed View of Life* ***JTabbedPane***
13.5	*KAE*	*Action Handling: Making Buttons Work* ***ActionListener***
13.6	*KAE*	*Action Handling Using Anonymous Inner Classes* ***ActionListener***

cada uma das 35 tarefas, os top-10 pares Q&A recomendados. Para cada par Q&A, eles avaliaram os dois critérios previamente descritos na Subseção 4.2.7. Na Tabela 21, a coluna *"Kappa Antes"* mostra o *Kappa Ponderado* (do inglês, *Weighted Kappa* (COHEN, 1968)) calculado para medir a concordância entre os dois avaliadores. Nesta tabela, cada linha representa uma tripla *"Tópico / Cenário / Critério"*. Portanto, na primeira linha o *Kappa Ponderado* foi calculado para comparar as notas dadas pelos dois avaliadores para os pares Q&A retornados para as 6 tarefas selecionadas para o *Cenário NKAE* para o tópico *Swing* (i.e. cada linha representa uma comparação entre 60 valores, uma vez que

Tabela 19 – Tarefas relacionadas com o tópico *Boost*.

Tarefa	Cenário	Título da Tarefa
2.8	*NKAE*	*Parsing date-time input*
3.1	*NKAE*	*Doing something at scope exit*
12.5	*NKAE*	*Using portable math functions*
12.7	*NKAE*	*Combining multiple test cases in one test module*
10.7	*NKAE*	*The portable way to export and import functions and classes*
3.5	*NKAE*	*Reference counting pointers to arrays used across methods*
7.7	*KAE*	*Using a reference to string type* ***string_ref***
10.2	*KAE*	*Detecting RTTI support* ***type_index***
1.11	*KAE*	*Making a noncopyable class* ***noncopyable***
9.2	*KAE*	*Using an unordered set and map* ***unordered_set***
7.2	*KAE*	*Matching strings using regular expressions* ***regex***
8.8	*KAE*	*Splitting a single tuple into two tuples* ***vector***

Tabela 20 – Tarefas relacionadas com o tópico *LINQ*.

Tarefa	Cenário	Título da Tarefa
2	*NKAE*	*Find all capitalized words in a phrase and sort by length*
10	*NKAE*	*Pre-compiling Queries for Performance*
1	*NKAE*	*Change the font for all labels on a windows form*
5	*NKAE*	*Concatenating the selected strings from a CheckedListBox*
11	*NKAE*	*Desktop Search Statistics*
12	*NKAE*	*Calculate the Standard Deviation*
3	*KAE*	*Find all the prime numbers in a given range* ***Count***
7	*KAE*	*Selecting Pages of Data from Northwind* ***Skip***
4	*KAE*	*Find all complex types in a given assembly* ***Distinct***
9	*KAE*	*Dynamic Sort Order* ***OrderByDescending***
8	*KAE*	*Querying XML Using LINQ* ***Contains***

os avaliadores analisaram os top-10 pares Q&A recomendados para cada tarefa).

Em um passo posterior, as avaliações feitas pelos dois avaliadores foram comparadas. Os pares Q&A cujas diferenças entre as notas dada pelos dois avaliadores foi maior ou igual a 2 foram marcados para uma discussão posterior (ex. o *Autor A* avaliou o critério *Relev* como 4 para um par Q&A e o *Autor B* avaliou o critério *Relev* como 1 para o mesmo par). A razão para considerar somente diferenças maiores ou iguais a 2 é baseada na interpretação de que uma diferença de 1 pode ser considerada como sendo uma concordância parcial entre os avaliadores ao invés de um desacordo.

Após a marcação daqueles pares com maior divergência, os avaliadores discutiram cada um deles e chegaram a um acordo sobre cada um deles. Após a modificação das notas neste passo de discussão, o *Kappa Ponderado* foi calculado novamente e após este passo, observou-se que ocorreu uma melhoria na concordância entre os avaliadores (um valor de *Kappa Ponderado* igual a "1" significa uma concordância perfeita). Considerando que foi obtido uma alta concordância geral, decidiu-se por considerar somente as avaliações feitas pelo *Autor A*.

Tabela 21 – Abordagem *Lucene+Score+How-to*: Kappa Ponderado (Comparação de Concordância).

Tópico / Cenário / Critério	Kappa Antes	Kappa Depois
Swing / NKAE / *Relev*	0,60	0,89
Swing / NKAE / *Reprod*	0,84	0,95
Swing / KAE / *Relev*	0,58	0,92
Swing / KAE / *Reprod*	0,86	0,98
Boost / NKAE / *Relev*	0,54	0,95
Boost / NKAE / *Reprod*	0,94	0,95
Boost / KAE / *Relev*	0,81	0,94
Boost / KAE / *Reprod*	0,67	0,98
LINQ / NKAE / *Relev*	0,95	0,97
LINQ / NKAE / *Reprod*	0,81	0,93
LINQ / KAE / *Relev*	0,68	0,92
LINQ / KAE / *Reprod*	0,87	0,94

4.2.9 Avaliação de diferentes mecanismos de peso para um par Q&A

Com o objetivo de determinar se as respostas deveriam ter um peso maior comparado com as perguntas, foram analisados diferentes mecanismos de peso que aumenta o peso da resposta. Nesta avaliação W_q e W_a referem-se respectivamente aos pesos para a pergunta e resposta de um par Q&A. A Tabela 22 mostra os mecanismos de peso investigados:

Tabela 22 – Mecanismos de peso investigados (W_q = Peso para a pergunta de um par Q&A; W_a = Peso para a resposta de um par Q&A).

W_q	W_a	Observação
0,5	0,5	Nenhuma diferença de ponderação
0,3	0,7	-
0,4	0,6	-
0,2	0,8	-

Para descobrir qual mecanismo de peso produz o melhor ranqueamento geral, utilizou-se uma métrica denominada *Normalized Discounted Cumulative Gain* (NDCG). Esta métrica foi utilizada para se ter uma avaliação numérica do ranqueamento dos pares Q&A recomendados nos experimentos. NDCG (Equação 2) é geralmente utilizado para avaliar os resultados retornados pelos motores de busca e utiliza uma noção de relevância multivalorada (MANNING; RAGHAVAN; SCHüTZE, 2008):

$$NDCG(Q,k) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{M(j,m)}-1}{log_2(1+m)} \tag{3}$$

onde k é o tamanho do conjunto de resultados. Nos experimentos realizados $k = 10$, uma vez que foram recomendados 10 pares Q&A para o usuário. *M(j,d)* é o valor da métrica dado para o documento *d* para a consulta *j*. Devido ao fato de que foram considerados dois critérios nos experimentos, *M(j,d)* pode ser *Relev* ou *Reprod*. O valor NDCG

foi calculado para ambos os critérios. Z_{kj} é o fator de normalização calculado de modo que o NDCG seja igual a 1. Para o cálculo deste fator, utilizou-se a mesma abordagem conduzida em um trabalho relacionado (PONZANELLI; BACCHELLI; LANZA, 2013a). Nesta abordagem, o fator é calculado em um cenário em que todos os documentos retornados possuem a nota máxima. Decidiu-se por realizar uma avaliação com esta métrica com o intuito de comparar os resultados de ambos os trabalhos (não totalmente porque a abordagem proposta neste capítulo do trabalho de doutorado recomenda os top-10 pares Q&A, enquanto a abordagem do trabalho relacionado recomenda os top-15 *posts* do SO). No entanto, é preciso interpretar o valor NDCG com cautela, pois apenas as consultas que tiverem a maioria dos 10 pares Q&A com relevância máxima (i.e. valor 4) terão valor NDCG próximo de 1 (i.e. valor máximo), sendo esta métrica extremamente rigorosa. Considerando uma dada tarefa de programação a ser realizada com uma determinada API, o mais importante para auxiliar o desenvolvedor a resolver a tarefa é ter um ou mais pares Q&A relevantes bem ranqueados no top-10, i.e. não necessariamente todos os pares Q&A precisam ser altamente relevantes no contexto da tarefa. O valor do fator de normalização utilizado foi $Z_{kj} \approx 0{,}01$. $|Q| = 35$ pois foram utilizadas 35 tarefas de programação nos experimentos.

Para encontrar o melhor mecanismo de peso para a estratégia de recomendação proposta (i.e. aquele que produz o melhor ranqueamento geral), calculou-se os valores de NDCG para cada critério (i.e. *Relev* e *Reprod*) e para cada mecanismo de peso mostrado na Tabela 22. Além disso, comparou-se estes valores entre si. Quanto maior for o valor do NDCG, melhor seria a qualidade do ranqueamento.

Uma análise qualitativa manual envolvendo as top-10 recomendações para todas as 35 tarefas de programação considerando os critérios *Relev* e *Reprod* foi necessária para conduzir o cálculo do NDCG. Assim, iniciou-se esta análise (mostrada acima na Subseção 4.2.8) escolhendo um mecanismo de peso, onde qualquer um deles poderia ser igualmente escolhido porque a ordem não afeta o resultado. O mecanismo de peso $W_q = 0{,}3$ and $W_a = 0{,}7$ foi selecionado arbitrariamente e o ranqueamento gerado por ele foi considerado como base de comparação com relação aos ranqueamentos gerados pelos mecanismos de peso remanescentes.

Após esta etapa, calculou-se o ranqueamento (i.e. top-10 recomendações para cada tarefa de programação consultada no sistema de recomendação) gerado para cada mecanismo de peso para as 35 tarefas de programação listadas acima nas Tabelas 18, 19 e 20 com o objetivo de identificar qual mecanismo de peso gera o melhor ranqueamento geral. Cada um destes mecanismos de peso gera um ranqueamento diferente. Porém, descobriu-se que a diferença entre os ranqueamentos produzidos por estes diferentes mecanismos de peso foi muito pequena (i.e. os top-10 pares Q&A recomendados foram repetidos na sua maioria para diferentes mecanismos de peso considerando a mesma tarefa de desenvolvimento).

Observou-se que os novos pares Q&A recomendados não estavam presentes no ranqueamento gerado pelo mecanismo de peso $W_q = 0{,}3$ e $W_a = 0{,}7$ (i.e. a base de comparação) mas estavam presentes no ranqueamento gerado pelos mecanismos de peso restantes (i.e. $W_q = 0{,}5$ e $W_a = 0{,}5$, $W_q = 0{,}4$ e $W_a = 0{,}6$ or $W_q = 0{,}2$ e $W_a = 0{,}8$). Estes novos pares Q&A foram avaliados individualmente pelos mesmos 2 sujeitos humanos mencionados acima nas Subseções 4.1.2 e 4.2.8 (i.e. *Autor A* e *Autor B*). Para cada um dos 13 novos pares Q&A recomendados, eles deram notas para os dois critérios previamente descritos (*Relev* e *Reprod*). Na Tabela 23, a coluna "Kappa Antes" mostra o *Kappa Ponderado* (COHEN, 1968) calculado para medir a concordância entre os dois avaliadores. Nesta tabela, cada linha representa um critério (*Relev* ou *Reprod*). Portanto, na segunda linha desta tabela o *Kappa Ponderado* foi calculado para comparar as notas de *Reprod* dadas pelos dois avaliadores para os 13 novos pares Q&A recomendados.

Tabela 23 – Novos pares Q&A recomendados: Kappa Ponderado (Comparação de concordância).

Critério	Kappa Antes	Kappa Depois
Relev	0,85	0,85
Reprod	0,56	0,89

As avaliações dos dois autores foram comparadas subsequentemente. Os pares Q&A em que a diferença das notas dadas pelos autores foi maior ou igual a 2 foram marcados para uma discussão posterior (ex. o *Autor A* avaliou o critério *Reprod* como sendo 0 para um par e o *Autor B* avaliou o critério *Reprod* como sendo 2 para o mesmo par). As razões para se considerar somente as diferenças maiores ou iguais a 2 é baseada na interpretação de que uma diferença de 1 pode ser considerada como sendo uma concordância parcial entre os avaliadores ao invés de um desacordo.

Após a marcação destes pares com maior divergência, os avaliadores discutiram cada um dos 5 casos divergentes para o critério *Reprod* e chegaram a um consenso sobre eles. Após a modificação das notas neste passo de discussão, o *Kappa Ponderado* foi calculado novamente e é mostrado na coluna "Kappa Depois" da Tabela 23. Comparando os valores antes e depois deste passo, pode-se observar que a concordância foi melhorada (um valor de *Kappa Ponderado* igual a "1" significa uma concordância perfeita). Na primeira linha da Tabela 23, os valores para o critério *Relev* foram os mesmos pois não havia diferença maior ou igual a 2 entre as notas dos avaliadores nos 13 novos pares Q&A recomendados. Visto que foi obtido uma alta concordância geral entre os avaliadores, decidiu-se por considerar somente as avaliações feitas pelo *Autor A*.

A Tabela 24 mostra os 13 novos pares Q&A avaliados pelos dois avaliadores. Para cada par Q&A, esta tabela mostra o Identificador da Tarefa, o Cenário, a Posição no Ranqueamento nos top-10 resultados (i.e. valores variando de 1 a 10), o Mecanismo de Peso e as notas finais para os critérios *Relev* e *Reprod*.

Tabela 24 – Novos pares Q&A recomendados pelos dois avaliadores (W_q e W_a referem-se respectivamente aos pesos para as perguntas e respostas de um par Q&A).

Tarefa	Tópico	Cenário	Posição Ranqueamento	Mecanismo de Peso	*Relev*	*Reprod*
3.1	Boost	*NKAE*	10	$W_q = 0{,}2$ $W_a = 0{,}8$	0	4
1.11	Boost	*KAE*	10	$W_q = 0{,}2$ $W_a = 0{,}8$	0	1
9.2	Boost	*KAE*	8	$W_q = 0{,}2$ $W_a = 0{,}8$	0	3
4	LINQ	*KAE*	10	$W_q = 0{,}2$ $W_a = 0{,}8$	0	1
3.1	Boost	*NKAE*	8	$W_q = 0{,}4$ $W_a = 0{,}6$	0	1
12.7	Boost	*NKAE*	9	$W_q = 0{,}4$ $W_a = 0{,}6$	0	1
7.7	Boost	*KAE*	10	$W_q = 0{,}4$ $W_a = 0{,}6$	0	3
9.2	Boost	*KAE*	10	$W_q = 0{,}4$ $W_a = 0{,}6$	0	2
7.2	Boost	*KAE*	10	$W_q = 0{,}4$ $W_a = 0{,}6$	4	3
12.7	Boost	*NKAE*	8	$W_q = 0{,}5$ $W_a = 0{,}5$	0	1
12.7	Boost	*NKAE*	6	$W_q = 0{,}5$ $W_a = 0{,}5$	0	1
10.7	Boost	*NKAE*	10	$W_q = 0{,}5$ $W_a = 0{,}5$	0	1
3.5	Boost	*NKAE*	10	$W_q = 0{,}5$ $W_a = 0{,}5$	0	4

Após este passo de avaliação, para cada mecanismo de peso considerado, calculou-se os valores de $NDCG_{Relev}$ e $NDCG_{Reprod}$ para a abordagem de recomendação proposta neste trabalho (i.e. *Lucene+Score+How-to*). A Tabela 25 mostra os resultados obtidos.

Tabela 25 – Valores de $NDCG_{Relev}$ e $NDCG_{Reprod}$ para cada mecanismo de peso considerado (W_q e W_a referem-se respectivamente aos pesos para a pergunta e resposta de um par Q&A).

NDCG	$W_q = 0{,}2$ $W_a = 0{,}8$	$W_q = 0{,}4$ $W_a = 0{,}6$	$W_q = W_a = 0{,}5$	$W_q = 0{,}3$ $W_a = 0{,}7$
Relev	0,3579	0,3565	0,3567	0,3583
Reprod	0,5249	0,5215	0,5228	0,5243

Como mostrado na Tabela 25, não existe uma diferença muito grande nos valores de NDCG para os mecanismos de peso considerados. O valor do $NDCG_{Relev}$ para o mecanismo de peso $W_q = 0{,}3$ e $W_a = 0{,}7$ foi um pouco maior em relação aos outros casos. Todavia, o valor do $NDCG_{Reprod}$ para o mecanismo de peso $W_q = 0{,}2$ e $W_a = 0{,}8$ foi um pouco melhor do que os outros casos. Neste trabalho, os resultados apresentados foram obtidos utilizando-se o mecanismo de peso $W_q = 0{,}3$ e $W_a = 0{,}7$ (melhor valor para o $NDCG_{Relev}$). No entanto, os resultados finais não seriam muito diferentes se qualquer um dos outros mecanismos de peso fosse utilizado.

4.2.10 Comparação de resultados com o *Google*

A abordagem de recomendação proposta (*Lucene+Score+How-to*) foi comparada com o motor de busca *Google*. O *Google* foi escolhido pois ele é o motor de busca de propósito geral mais popular, sendo geralmente utilizado para consultar o SO. Além disso, Sim *et al.* (SIM et al., 2011) realizaram um experimento com 36 participantes para avaliar três abordagens diversas para consulta de código na *Web* (recuperação de informações

de propósito geral, reutilização de componentes e motores de busca específicos para pesquisa de código-fonte), conforme representado por cinco sítios (*Google* [11], *SourceForge* [12], *Koders* [13], *Krugle* [14] e *Google Code Search* [15]). Eles descobriram que era mais fácil encontrar exemplos de referência do que componentes para reutilização e que os participantes obtiveram os melhores resultados usando o *Google*.

Neste trabalho de doutorado, foram consultadas também algumas tarefas utilizando o motor de busca do SO. Entretanto, para várias tarefas, as consultas no SO retornaram poucos *posts*. Dessa forma, optou-se por utilizar o *Google* ao invés do motor de busca do SO. Definiu-se um protocolo para pesquisar no *Google*, baseado nas seguintes regras:

- A consulta deve obrigatoriamente conter o nome da API e deve estar restrita ao *site* do SO (ex. *"Program: Custom Font Chooser* ***Swing*** *site:stackoverflow.com"* é uma consulta que corresponde a uma tarefa de programação utilizando a API Java Swing e está limitada ao *site* do SO). Decidiu-se por fazer isso para restringir a avaliação para somente *posts* do SO, uma vez que os critérios de avaliação considerados (*Relev* e *Reprod*) são mais adequados para avaliar este tipo de conteúdo (ex. é difícil avaliar se um livro é relevante para uma tarefa de programação em mãos, uma vez que a solução para esta tarefa é bastante específica);

- O *Google* deve ser configurado para buscar somente os *posts* que foram postados no SO até a data em que o *dump* foi realizado, i.e. 5 de Março de 2013. Decidiu-se que esta configuração era necessária para garantir que as duas abordagens (*Lucene+Score+How-to* e *Google*) estivessem lidando com os mesmos dados (i.e. o conjunto de *posts* do SO até a data do *dump*).

Foram consideradas as mesmas 35 tarefas de programação avaliadas no passo anterior. Novamente, o *Autor A* e o *Autor B* foram os participantes nesta avaliação. Para cada tarefa, construiu-se uma consulta considerando as regras definidas acima e as mesmas foram pesquisadas no *site* do *Google*. O *Autor A* e o *Autor B* individualmente avaliaram para cada uma das tarefas, os 10 primeiros *posts* do SO recomendados considerando os mesmos critérios (*Relev* e *Reprod*). Um *post* do SO contém uma pergunta e uma ou mais respostas. O número de respostas para uma pergunta é o número de pares Q&A para esta pergunta. Decidiu-se por avaliar somente o par Q&A com maior pontuação que pertence ao *post* do SO, para ser possível fazer uma comparação justa com a abordagem proposta (*Lucene+Score+How-to*), que recomenda pares Q&A ao invés de *posts* inteiros do SO. Para cada par Q&A, o *Autor A* e o *Autor B* deram notas para os critérios previamente descritos. Repetiu-se o mesmo processo que foi feito para a abordagem

[11] http://www.google.com/
[12] http://sourceforge.net/
[13] http://www.koders.com/
[14] http://www.krugle.com/
[15] http://www.google.com/codesearch

Lucene+Score+How-to (i.e. as avaliações dos autores foram comparadas e os pares Q&A com maior divergência entre notas foram rediscutidos). A Tabela 26 possui a mesma estrutura de colunas que a Tabela 21 e mostra o valor do *Kappa* Ponderado (COHEN, 1968) calculado para medir a concordância entre os dois avaliadores. Novamente, uma vez que foi obtido uma alta concordância geral, decidiu-se por considerar somente as avaliações feitas pelo *Autor A*.

Tabela 26 – *Google*: Kappa Ponderado (Comparação de Concordância).

Tópico / Cenário / Critério	Kappa Antes	Kappa Depois
Swing / NKAE / *Relev*	0,71	0,95
Swing / NKAE / *Reprod*	0,53	0,96
Swing / KAE / *Relev*	0,82	0,94
Swing / KAE / *Reprod*	0,56	0,96
Boost / NKAE / *Relev*	0,87	0,97
Boost / NKAE / *Reprod*	0,80	0,95
Boost / KAE / *Relev*	0,80	0,97
Boost / KAE / *Reprod*	0,65	0,97
LINQ / NKAE / *Relev*	0,88	0,99
LINQ / NKAE / *Reprod*	0,66	0,95
LINQ / KAE / *Relev*	0,81	0,98
LINQ / KAE / *Reprod*	0,48	0,83

4.3 Resultados

Esta seção apresenta os resultados do experimento proposto. As Tabelas 27 a 38 possuem a mesma estrutura de colunas: o identificador da tarefa e as 10 primeiras posições do ranqueamento (P1 a P10) para cada abordagem (i.e. *Lucene+Score+How-to* e *Google*). Cada posição do ranqueamento corresponde a um par Q&A recomendado. As tabelas mostram o ranqueamento obtido para cada tarefa de programação, considerando o tópico (*Swing*, *Boost* ou *LINQ*), o cenário (*NKAE* ou *KAE*) e o critério (*Relev* ou *Reprod*).

Por exemplo, na primeira linha de dados da Tabela 27, são apresentados os resultados para a tarefa 13.14 para ambas as abordagens (*Lucene+Score+How-to* e *Google*). Para a abordagem *Lucene+Score+How-to*, é possível observar que o par Q&A melhor ranqueado (coluna P1) recebeu a nota 2 (neutro). Além disso, é possível observar pares Q&A altamente relevantes nas posições P3 e P6. Na mesma linha, é possível observar o ranqueamento produzido pela abordagem *Google* para esta tarefa de programação. Além disso, é possível notar pares Q&A altamente relevantes nas posições P1 e P7.

Foram calculados os valores de NDCG_{Relev} e NDCG_{Reprod} para cada abordagem. A Tabela 39 mostra os resultados obtidos. Pode-se afirmar que o valor de NDCG_{Relev} para o *Google* foi 5,62% maior do que o valor para a abordagem *Lucene+Score+How-to* e que o valor de NDCG_{Reprod} para o *Google* foi 4,75% maior do que o valor para a abordagem *Lucene+Score+How-to*. Estes resultados mostram que a qualidade geral dos

Tabela 27 – *Swing* - Cenário NKAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.14	2	2	4	2	2	4	2	2	2	2	4	2	2	0	0	0	4	0	0	0
13.13	0	4	4	4	3	3	1	3	3	4	4	4	4	4	3	2	0	3	4	4
13.11	1	2	2	0	2	2	3	0	1	2	1	0	0	4	0	0	0	0	0	0
13.3	3	0	0	2	0	0	2	3	0	0	0	1	1	1	0	1	4	4	1	0
13.1	4	4	0	3	3	4	4	4	2	3	4	2	4	2	4	2	4	4	4	2
13.8	3	1	2	3	4	2	2	2	1	4	1	1	2	1	0	1	4	3	0	4

Tabela 28 – *Swing* - Cenário NKAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.14	0	0	4	3	0	3	4	3	2	0	4	3	3	3	4	4	4	1	2	0
13.13	3	0	0	4	4	0	0	0	4	3	3	4	4	3	0	0	0	0	4	4
13.11	0	0	0	3	3	3	4	4	3	0	0	0	3	2	2	4	4	4	2	2
13.3	0	4	0	0	0	0	4	0	0	0	0	0	0	3	0	1	4	4	1	0
13.1	4	4	4	4	4	4	4	0	3	3	3	4	4	3	4	0	0	3	4	4
13.8	0	0	0	3	4	3	0	4	4	4	4	0	4	4	3	2	4	4	0	4

Tabela 29 – *Swing* - Cenário KAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.12	4	2	3	3	4	0	2	0	2	2	4	4	4	4	4	0	3	4	1	4
13.2	3	1	2	1	4	4	1	0	0	2	3	4	4	4	4	2	4	1	2	0
13.9	4	4	4	1	1	0	0	4	0	0	4	4	0	4	4	0	0	1	3	0
13.4	4	2	2	2	3	2	1	3	3	3	4	0	3	0	1	4	2	0	4	4
13.5	4	2	4	2	2	2	2	2	2	3	4	4	0	4	3	4	3	3	4	1
13.6	2	4	4	3	3	3	3	3	4	4	4	4	0	4	4	3	2	4	4	4

pares Q&A recomendados pelo *Google* foi um pouco melhor do que para a abordagem *Lucene+Score+How-to*.

Considerando que é desejável ter pares Q&A com altas notas entre os 10 primeiros recomendados, decidiu-se por analisar o número de pares Q&A recomendados pela abordagem *Lucene+Score+How-to* e pelo motor de busca *Google* cujas notas foram maiores ou iguais a 3 para os critérios *Relev* ou *Reprod*. Para esta análise, utilizou-se os gráficos mostrados nas Figuras 5 e 6. Estas figuras mostram os gráficos de dispersão para os critérios *Relev* e *Reprod* de cada abordagem de recomendação. Os três gráficos na primeira coluna da Figura 5 consideram o critério *Relev* para a abordagem *Lucene+Score+How-to* nos três tópicos avaliados neste estudo (*Boost*, *Swing* e *LINQ*), enquanto os três gráficos presentes na segunda coluna desta figura consideram o mesmo critério para a abordagem *Google*. A Figura 6 é similar à Figura 5, porém leva em conta o critério *Reprod* ao invés do critério *Relev*.

Cada gráfico considera as atividades selecionadas para o *Cenário NKAE* (símbolo de diamante) e aquelas selecionadas para o *Cenário KAE* (símbolo de quadrado). As atividades aparecem nos gráficos na ordem mostrada nas Tabelas 18, 19 e 20. Portanto,

Tabela 30 – *Swing* - Cenário KAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.12	0	0	0	4	3	3	4	0	2	0	0	4	4	4	3	3	4	4	4	4
13.2	4	0	0	0	0	0	4	0	2	2	1	4	4	4	4	4	4	4	4	2
13.9	4	4	4	3	4	4	4	4	0	0	4	4	4	4	4	4	4	4	4	0
13.4	4	0	0	3	4	4	0	4	4	4	4	0	4	3	0	4	2	0	4	4
13.5	0	4	0	0	0	4	0	0	0	3	4	4	4	4	4	4	3	4	4	2
13.6	3	4	3	3	4	4	4	4	4	4	4	3	1	4	3	3	4	4	4	3

Tabela 31 – *Boost* - Cenário NKAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2.8	3	2	3	3	3	2	2	2	3	3	4	4	4	4	4	4	2	3	4	4
3.1	1	2	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0
12.5	1	0	0	0	4	0	4	4	4	4	4	0	0	0	0	0	0	0	0	0
12.7	2	4	0	0	0	2	2	2	4	0	2	4	4	0	0	0	0	0	0	0
10.7	3	2	0	1	2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
3.5	3	2	2	2	3	2	2	3	2	3	3	3	1	3	3	3	3	4	3	0

Tabela 32 – *Boost* - Cenário NKAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2.8	2	4	4	4	4	0	4	4	4	4	4	4	4	4	4	3	2	4	4	4
3.1	4	4	4	0	4	1	0	2	0	0	4	3	0	4	3	3	3	2	3	3
12.5	0	0	0	0	4	0	4	4	0	0	3	3	0	0	0	1	3	3	0	0
12.7	0	0	0	3	4	0	0	4	3	4	1	3	3	0	0	0	0	0	0	0
10.7	4	0	4	0	4	0	3	4	4	0	0	1	0	0	1	0	0	0	1	0
3.5	4	0	4	0	0	4	0	0	0	4	0	3	4	0	0	3	3	0	0	3

na primeira linha da Figura 5 (i.e. os gráficos "*Lucene+Score+How-to*: Boost - *Relev*" e "Google: Boost - *Relev*"), a atividade 1 de cada cenário corresponde a *"2.8 - Parsing date-time input"* (*Cenário NKAE*) e *"7.7 Using a reference to string type string_ref"* (*Cenário KAE*), como pode ser observado na Tabela 19. Além de considerar o valor 4 para o critério *Relev*, considerou-se também o valor 3 pois, embora não sendo o melhor caso, os pares Q&A avaliados com a nota 3 para *Relev* podem ser utilizados para resolver quase todo o problema representado pela consulta. Similarmente, os pares Q&A com *Reprod* igual a 3 possuem trechos de código quase completos.

É possível observar nos gráficos da Figura 5 para a abordagem *Lucene+Score+How-to* que todas as atividades relacionadas com a API *Swing* possuem no mínimo um par Q&A com *Relev* ≥ 3. Somente 8 das 35 atividades (22,85%) não possuem pares Q&A com *Relev* ≥ 3 entre os 10 primeiros pares recomendados. Com relação aos gráficos desta mesma figura para o *Google*, é possível observar que todas as atividades para o *Swing* possuem no mínimo um par Q&A com *Relev* ≥ 3. Das 35 tarefas, 10 (28,57%) não possuem pares Q&A com *Relev* ≥ 3 entre os 10 primeiros pares recomendados, o que configura um desempenho pior comparado com a abordagem *Lucene+Score+How-to*.

É possível observar nos gráficos da Figura 6 que todas as 35 atividades possuem no

Tabela 33 – *Boost* - Cenário KAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
7.7	2	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0
10.2	2	2	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1.11	3	1	3	1	1	3	3	2	2	1	4	2	0	3	4	0	4	0	0	0
9.2	2	1	1	1	3	1	3	1	1	0	2	0	0	3	2	2	4	3	4	0
7.2	4	4	4	4	3	4	0	2	2	4	4	4	4	0	0	4	4	4	2	4
8.8	1	1	1	1	1	1	1	1	1	1	1	2	0	0	0	0	0	0	0	0

Tabela 34 – *Boost* - Cenário KAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
7.7	4	0	4	1	4	2	4	4	4	4	4	0	4	3	0	4	4	4	3	1
10.2	4	0	0	2	4	4	0	0	0	0	4	2	0	0	0	0	0	0	0	0
1.11	0	3	0	4	2	0	4	4	4	2	4	4	0	2	4	4	3	0	2	1
9.2	0	4	0	3	4	4	3	4	2	2	0	2	4	3	4	0	1	4	4	2
7.2	0	4	4	4	4	4	4	4	0	4	4	4	4	0	4	4	4	4	4	4
8.8	4	4	4	4	2	4	3	3	3	4	1	0	3	2	2	4	3	4	3	0

Tabela 35 – *LINQ* - Cenário NKAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2	0	0	0	2	2	0	2	0	0	0	2	0	0	0	0	0	0	0	0	0
10	1	2	4	1	1	0	0	0	0	0	4	4	2	4	2	0	0	4	4	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	2	0	2	1	2	2	2	2	0	2	0	3	3	4	3	0	0	0	0	0
11	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
12	4	4	4	4	1	4	3	1	3	0	4	4	0	4	0	0	0	0	0	0

mínimo um par Q&A *Reprod* ≥ 3. Com relação aos gráficos desta mesma figura para o Google, apenas 1 das 35 tarefas (2,85%) não possui pares Q&A *Reprod* ≥ 3. Esta atividade corresponde à *"10.7 - The portable way to export and import functions and classes"* (*Boost - Cenário NKAE*). Para esta atividade, o Google retornou alguns *posts* do SO relacionados com a linguagem de programação *Python*. Estes *posts* do SO não continham nada sobre a biblioteca *Boost* da linguagem de programação C++. Este tipo de comportamento estranho na recomendação do Google não é desejável e observou-se que este comportamento não é um caso isolado. Para demonstrar este ponto, conduziu-se uma análise qualitativa de todas as tarefas em que o Google e a abordagem *Lucene+Score+How-to* produziram uma recomendação ruim.

Com o objetivo de entender exatamente quais tipos de consultas em que o *Google* lidava melhor do que a abordagem *Lucene+Score+How-to*, e vice-versa, conduziu-se uma análise qualitativa mais aprofundada. Para cada caso (i.e. *"Tarefa / Tópico / Cenário / Critério"*), calculou-se a quantidade de notas 3 e 4 (i.e. bons valores para os critérios *Relev* e *Reprod*). Se esta quantidade fosse maior ou igual a 6, assumiu-se como sendo um caso onde houve uma boa recomendação. Caso contrário, assumiu-se como sendo um caso onde houve uma recomendação ruim. Além disso, para cada caso, analisou-se a qualidade

Tabela 36 – *LINQ* - Cenário NKAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2	3	0	3	4	4	4	4	0	3	3	4	3	0	4	4	4	1	1	0	3
10	0	0	3	3	2	0	3	0	0	0	4	3	0	4	2	0	0	3	3	0
1	4	1	0	0	2	2	2	4	4	0	4	4	4	2	4	4	0	4	0	1
5	3	2	4	0	3	4	3	3	3	3	2	2	3	3	2	2	3	3	3	3
11	3	4	4	4	4	4	4	4	3	0	0	0	0	0	3	3	0	3	0	4
12	3	3	3	3	0	3	4	2	4	0	3	3	4	4	4	0	3	3	4	4

Tabela 37 – *LINQ* - Cenário KAE (0 = Irrelevante, 4 = Altamente Relevante).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
3	1	1	2	1	4	1	0	2	1	1	4	0	0	2	4	0	0	0	0	0
7	4	2	2	4	4	4	4	4	4	4	0	2	0	0	2	0	0	0	0	0
4	0	1	2	0	3	0	1	0	0	0	0	0	0	2	0	0	0	0	0	0
9	2	4	4	2	2	4	4	1	4	2	4	4	4	4	4	4	0	4	4	4
8	0	2	1	2	1	0	0	3	4	3	4	4	2	4	4	4	2	4	2	2

Tabela 38 – *LINQ* - Cenário KAE (0 = Não Reproduzível, 4 = Altamente Reproduzível).

	Lucene+Score+How-to										*Google*									
Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
3	4	4	4	4	3	3	3	4	4	2	3	3	3	4	4	3	4	4	3	4
7	3	4	4	3	2	3	3	2	4	3	4	4	0	3	3	3	3	3	3	3
4	3	0	3	3	4	3	0	0	0	0	1	1	0	3	0	2	4	0	0	4
9	3	4	4	3	3	4	3	2	3	2	4	4	3	3	3	3	3	3	3	3
8	3	0	0	0	2	0	0	0	4	0	3	3	4	3	3	4	4	3	4	4

Tabela 39 – Valores de NDCG_{Relev} e NDCG_{Reprod} para cada abordagem.

	Lucene+Score+How-to	*Google*
NDCG_{Relev}	0,3583	0,4145
NDCG_{Reprod}	0,5243	0,5718

das recomendações feitas por cada abordagem.

Com base nesta suposição, selecionaram-se os casos em que a abordagem proposta neste trabalho (i.e. *Lucene+Score+How-to*) superou o *Google*, e vice-versa. As Tabelas 40 e 41 mostram estes casos. Nessas tabelas, os nomes das classes ou operadores utilizados para o *Cenário KAE* são mostrados em negrito. A abordagem proposta superou o *Google* em 8 casos e o *Google* superou a abordagem *Lucene+Score+How-to* em 14 casos.

A Tabela 40 mostra os casos em que a abordagem *Lucene+Score+How-to* superou o Google. Dos 8 casos em que *Lucene+Score+How-to* consegue lidar melhor que o *Google*, 5 (62,5%) são pertencentes ao cenário NKAE (Elemento de API não-conhecido). Estes resultados sugerem que a abordagem proposta lida bem com consultas genéricas que não possuem o nome do elemento da API. Entretanto, o *Google* produz uma recomendação ruim para este tipo de consulta. Observou-se que nestes casos, o Google recomenda *posts* do SO que não estão marcados (do inglês, *tagged*) com a API desejada. Portanto, a

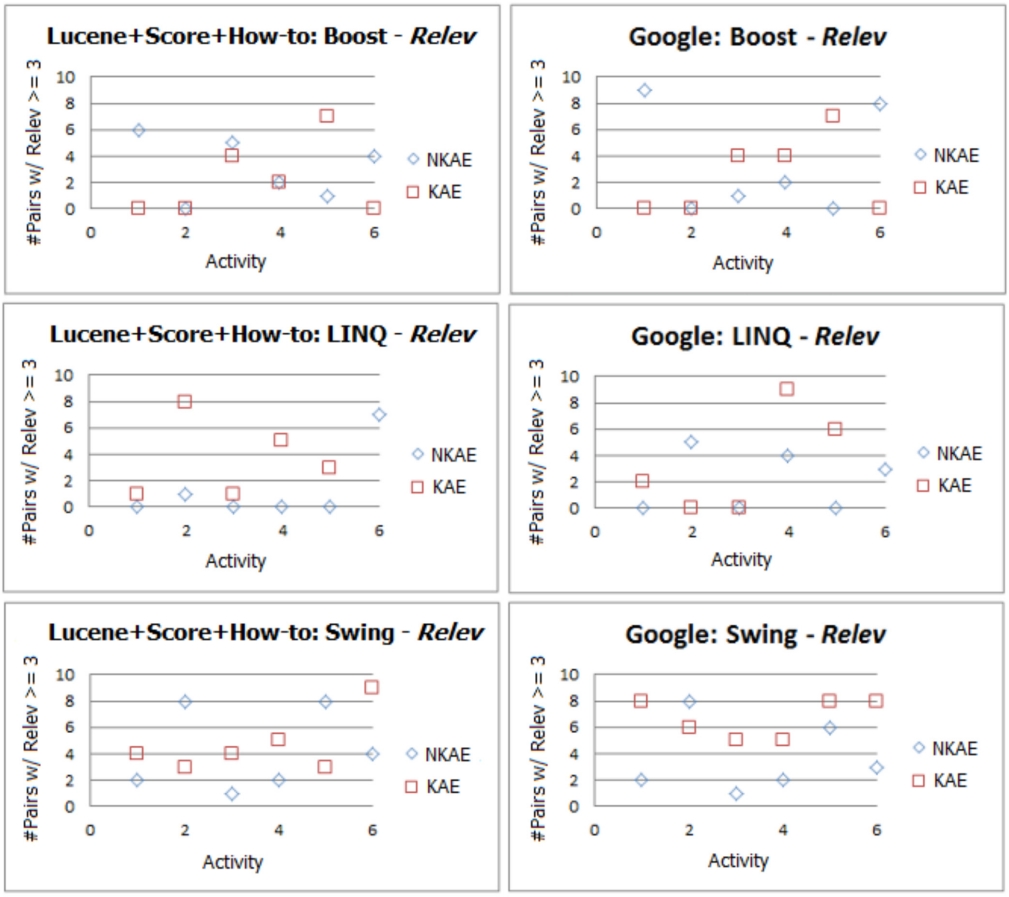

Figura 5 – Número de pares com *Relev* ≥ 3 (*Lucene+Score+How-to*: Esquerda, *Google*: Direita).

abordagem proposta tende a ser mais efetiva do que o *Google* no Cenário *NKAE*. Além disso, 4 dos 8 casos (50%) pertenciam ao tópico *Boost*. Estes resultados mostram que a abordagem *Lucene+Score+How-to* foi melhor que o Google no tópico *Boost*. A Tabela 42 mostra alguns tópicos recomendados pelo *Google* que não estavam relacionados com as APIs *Boost* e *LINQ*. Em 80% destes casos mostrados na Tabela 42, o Cenário era o *NKAE*.

A Tabela 41 mostra os casos em que o *Google* superou a abordagem *Lucene+Score+How-to*. Considerando estes 14 casos, 9 são relacionados com o Cenário KAE (*Known API Element* - Elemento da API conhecido), sendo 6 da API *Swing* e 3 da API *LINQ*. Estes resultados mostram que o Google tende a ser mais eficaz do que a abordagem proposta (i.e. utilizando expressões regulares para obter os nomes dos elementos da API a partir dos trechos de código-fonte) para o Cenário KAE.

Supõe-se que se uma API é bastante popular (i.e. possui um alto número de *posts* do SO marcados como a API *Swing*) no SO e o Cenário é o KAE, o *Google* tende a ser melhor do que a abordagem *Lucene+Score+How-to* devido ao seu sofisticado mecanismo de aprendizagem probabilística. Isso poderia explicar porque o *Google* foi melhor no tópico *Swing*, uma vez que esta API é bastante popular no SO e 6 dos 8 casos referentes ao tópico

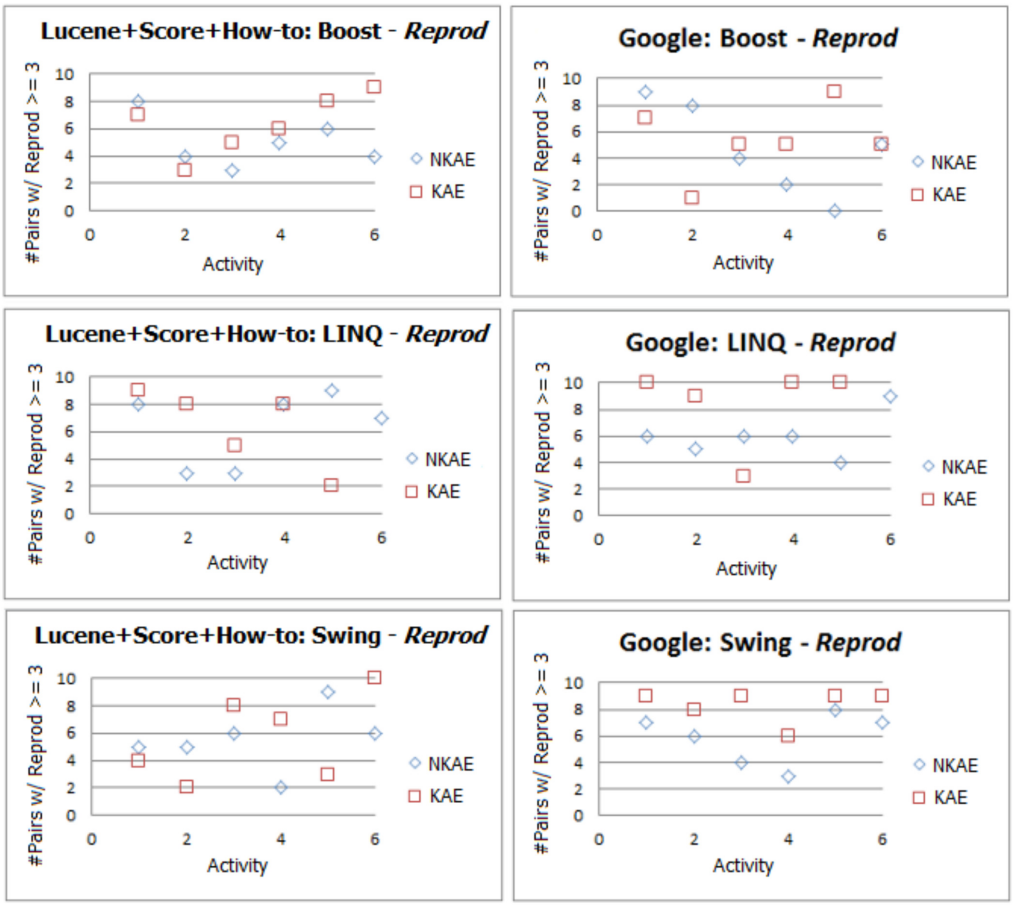

Figura 6 – Número de pares com *Reprod* ≥ 3 (*Lucene+Score+How-to*: Esquerda, *Google*: Direita).

Swing mostrados na Tabela 41 estavam relacionados com o Cenário KAE. O resultado sugere que a abordagem *Lucene+Score+How-to* tende a ser melhor em índices menores de documentos do que em índices maiores pois os mecanismos de filtragem adotados são úteis e o Google teria menos oportunidades de aprendizagem nestes casos. Se esta suposição for verdadeira, a abordagem de recomendação proposta tenderia a ser melhor para APIs menos populares no SO (ex. *Boost*, *Log4J*, *JFreeChart*) do que para APIs mais populares no SO (ex. *Swing*, *Hibernate*, *Spring*). Isso poderia explicar por que a abordagem proposta superou o Google no tópico *Boost*.

Para o tópico *LINQ* (apesar de ser um tópico popular no SO), nenhuma das duas abordagens obteve bons resultados. Talvez o SO não contenha soluções para essas tarefas. Além disso, algumas tarefas *LINQ* não possuem títulos suficientemente específicos (ex. o título da tarefa *"11. Desktop Search Statistics"* é muito genérico).

Para confirmar esta suposição, seria necessário realizar uma outra avaliação qualitativa envolvendo as novas APIs e os novos pares Q&A recomendados que exigiria um esforço significativo e, infelizmente, é deixado como trabalho futuro.

Tabela 40 – Casos em que o *Lucene+Score+How-to* superou o *Google*.

Tarefa	Tópico / Cenário / Critério	Título da Tarefa
13.11	*Swing / NKAE / Reprod*	*Choosing a Color from all the colors available on your computer*
12.5	*Boost / NKAE / Relev*	*Using portable math functions*
10.7	*Boost / NKAE / Reprod*	*The portable way to export and import functions and classes*
9.2	*Boost / KAE / Reprod*	*Using an unordered set and map* ***unordered_set***
8.8	*Boost / KAE / Reprod*	*Splitting a single tuple into two tuples* ***vector***
12	*LINQ / NKAE / Relev*	*Calculate the Standard Deviation*
11	*LINQ / NKAE / Reprod*	*Desktop Search Statistics*
7	*LINQ / KAE / Relev*	*Selecting Pages of Data from Northwind* ***Skip***
		Total: 8

Tabela 41 – Casos em que o *Google* superou o *Lucene+Score+How-to*.

Tarefa	Tópico / Cenário / Critério	Título da Tarefa
13.14	*Swing / NKAE / Reprod*	*Program: Custom Font Chooser*
13.13	*Swing / NKAE / Reprod*	*Changing a Swing Program's Look and Feel*
13.12	*Swing / KAE / Relev*	*Centering a Main Window* ***JFrame***
13.2	*Swing / KAE / Relev*	*Adding and Displaying GUI Components to a window* ***JFrame***
13.5	*Swing / KAE / Relev*	*Action Handling: Making Buttons Work* ***ActionListener***
13.12	*Swing / KAE / Reprod*	*Centering a Main Window* ***JFrame***
13.2	*Swing / KAE / Reprod*	*Adding and Displaying GUI Components to a window* ***JFrame***
13.5	*Swing / KAE / Reprod*	*Action Handling: Making Buttons Work* ***ActionListener***
3.5	*Boost / NKAE / Relev*	*Reference counting pointers to arrays used across methods*
3.1	*Boost / NKAE / Reprod*	*Doing something at scope exit*
1	*LINQ / NKAE / Reprod*	*Change the font for all labels on a windows form*
9	*LINQ / KAE / Relev*	*Dynamic Sort Order* ***OrderByDescending***
8	*LINQ / KAE / Relev*	*Querying XML Using LINQ* ***Contains***
8	*LINQ / KAE / Reprod*	*Querying XML Using LINQ* ***Contains***
		Total: 14

Tabela 42 – Casos de ruído na recomendação do *Google*.

Tarefa	Tópico / Cenário / Critério	Alguns tópicos que o Google recomendou
12.5	*Boost / NKAE / Relev*	*SIMD library, C language*
10.7	*Boost / NKAE / Relev*	*Python language, XML*
8.8	*Boost / KAE / Reprod*	*Scala language, Python language, SQLite*
11	*LINQ / NKAE / Reprod*	*SqlServer, Postgresql, Twitter*
12	*LINQ / NKAE / Relev*	*SQLite*

4.4 Discussão

É possível comparar parcialmente os resultados obtidos com a abordagem proposta (*Lucene+Score+How-to*) com os resultados apresentados por Ponzanelli *et al.* (PON-

ZANELLI; BACCHELLI; LANZA, 2013a). No trabalho deles, desenvolveu-se um *plugin* para Eclipse IDE denominado SEAHAWK com o objetivo de recomendar conteúdo proveniente do SO para auxiliar os desenvolvedores a resolverem problemas de programação. As principais diferenças entre os trabalhos são:

- A abordagem de recomendação *Lucene+Score+How-to* considera três aspectos para sugerir conteúdo: a similaridade textual que os pares Q&A possuem com a consulta, o *score* destes pares Q&A e a natureza "*How-to*" deles. Na abordagem deles, somente a relevância textual foi considerada. A etapa da classificação "*How-to*" é importante pois ela permite descartar pares que são mais teóricos do que práticos (ex. pares Q&A nas categorias *Conceptual* e *Seeking-something*);

- No experimento deles, somente tarefas Java foram consideradas. Neste capítulo, a abordagem *Lucene+Score+How-to* foi testada utilizando tarefas de programação de três diferentes tópicos (*Swing*, *Boost* e *LINQ*) que estão relacionados com diferentes linguagens de programação (Java, C++ e linguagens .NET, respectivamente);

- As tarefas utilizadas em ambos os trabalhos são diferentes: enquanto no presente trabalho as tarefas de programação foram selecionadas aleatoriamente de *cookbooks* para a realização do experimento, no trabalho deles as tarefas foram selecionadas de um curso de programação em Java. Quando se utiliza tarefas de programação de *cookbooks* ao invés de tarefas de um curso de programação, foca-se mais em tarefas práticas que um desenvolvedor pode realizar diariamente do que em itens didáticos utilizados para ensinar um tópico;

- No artigo deles, eles recomendam *posts* inteiros do SO. Já na abordagem proposta no presente trabalho, são recomendados pares Q&A individuais, uma vez que uma pergunta pode receber respostas bem votadas e respostas pouco votadas. O raciocínio por trás desta abordagem é recomendar somente as porções dos *posts* do SO que são bem avaliadas pela comunidade do SO;

- Foram definidos dois diferentes critérios que foram utilizados nos experimentos no presente trabalho: "Relevância" e "Reprodutibilidade". Ponzanelli *et al.* somente utilizou o critério "Relevância";

- No desenho experimental do presente trabalho, apresentou-se uma avaliação realizada por dois diferentes sujeitos humanos. Ponzanelli *et al.* não apresentou no artigo deles a maneira como os resultados deles foram avaliados (ex. por um ou dois autores).

Apesar de todas estas diferenças, o valor de NDCG obtido para o critério *Relev* no presente trabalho é muito superior ao respectivo valor obtido por Ponzanelli *et al.* (0,3583

e 0,0907, respectivamente), o que sugere que a abordagem proposta no presente trabalho supera a deles. Um pouco de cuidado é necessário para interpretar aqueles valores de NDCG pois a abordagem SEAHAWK avaliou os top-15 *posts* do SO recomendados, enquanto o presente trabalho avaliou os top-10 pares Q&A recomendados pela abordagem de recomendação *Lucene+Score+How-to*.

Analisando a Figura 5, é possível observar que para o critério *Relev*, o tópico *Swing* apresentou melhores resultados em relação aos tópicos *Boost* e *LINQ* pois para todas as tarefas de programação o tópico *Swing* obteve no mínimo um par Q&A com $Relev \geq 3$ para ambas as abordagens de recomendação (i.e. *Lucene+Score+How-to* e Google) e Cenários (i.e. *NKAE* e *KAE*).

Considerando a Figura 6, pode-se afirmar que para o critério *Reprod*, ambas as abordagens de recomendação possuem bons resultados na recomendação de pares Q&A que podem facilmente ser reproduzidos dentro da IDE, uma vez que para a abordagem *Lucene+Score+How-to*, todas as 35 tarefas tiveram no mínimo um par Q&A com $Reprod \geq 3$ e para o Google, somente uma das 35 tarefas (2,85%) não obteve pares Q&A com $Reprod \geq 3$.

Existe uma razão principal para explicar o baixo número de pares Q&A com $Relev \geq 3$ para algumas tarefas na abordagem de recomendação proposta (i.e. *Lucene+Score+How-to*). Primeiro, esta abordagem utiliza o título da tarefa como entrada para o motor de busca. Infelizmente algumas tarefas não possuem uma informação precisa no seus títulos. Por exemplo, a tarefa #5 *"11. Desktop Search Statistics"* do *LINQ* possui o objetivo de pesquisar no sistema de arquivos de um computador e contar o número de itens que são documentos, imagens ou mensagens eletrônicas. Ao utilizar somente a informação presente no título, não é possível saber esta informação. Em outras palavras, algumas tarefas não possuem títulos suficientemente claros. Para esta tarefa, o resultado foi igualmente ruim utilizando o motor de busca *Google*.

Todas as 35 tarefas tiveram no mínimo um par Q&A recomendado com $Reprod \geq 3$ e 34 tarefas tiveram no mínimo um par Q&A recomendado com $Reprod = 4$, indicando que a estratégia de recomendação proposta possui bom desempenho ao recomendar trechos de código que são reproduzíveis ou podem se tornar reproduzível com pequenos ajustes. As duas principais razões encontradas que explicam a dificuldade de reproduzir alguns trechos de código-fonte são:

- A utilização de uma variável que não foi declarada. Por exemplo, considere um par Q&A recomendado relacionado com a API *Swing*, cuja resposta possui um trecho de código que utiliza um objeto do tipo *Widget* (ex. um botão da interface gráfica), mas não mostra como criar este objeto. Embora criar um botão em *Swing* é bastante simples para a maioria das pessoas que possuem alguma experiência em programar GUIs, esta tarefa pode não ser trivial para alguém novo com programação de interface gráficas (GUIs);

- A omissão de algumas linhas de código (ex. algumas respostas usam "..." para indicar que algumas linhas são omitidas em um trecho de código). Novamente, esta falta de informação dificulta o uso do trecho de código em um ambiente de desenvolvimento como uma IDE.

Pode-se observar nas Figuras 5 e 6 que 26 das 35 tarefas da abordagem de recomendação *Lucene+Score+How-to* tiveram no mínimo um par Q&A recomendado que possui ambos os critérios *Relev*, *Reprod* ≥ 3.

Apesar dos valores de NDCG_{Relev} e NDCG_{Reprod} são menores na abordagem proposta do que na abordagem Google (veja a Tabela 39), foram identificados algumas fraquezas na recomendação do Google:

- Primeiro, o número de resultados ruins (i.e. pares Q&A com nota 0 em algum dos critérios *Relev* ou *Reprod*) para o *Google* foi maior nos tópicos *Boost* e *LINQ* do que para a abordagem *Lucene+Score+How-to*. A segunda e terceira linha de dados da Tabela 43 mostra que o Google obteve 46,66% e 40,90% de resultados ruins para os tópicos *Boost* e *LINQ*, respectivamente, enquanto a abordagem *Lucene+Score+How-to* obteve 32,08% e 31,81% de resultados ruins para os tópicos listados na mesma ordem. Além disso, o número de resultados ótimos (i.e. pares Q&A com nota 4 em algum dos critérios *Relev* ou *Reprod*) para a abordagem *Lucene+Score+How-to* foi maior no tópico *Boost* do que para a abordagem Google. A segunda linha de dados da Tabela 43 mostra que a abordagem *Lucene+Score+How-to* obteve 30% dos resultados ótimos para o tópico *Boost*, enquanto o *Google* obteve somente 25% neste tópico;

- Segundo, alguns *posts* do SO recomendados pelo Google não estavam relacionados com a respectiva API. Por exemplo, a tarefa 10.7 do *Boost* denominada *"The portable way to export and import functions and classes"*, o Google retornou alguns *posts* do SO relacionados com a linguagem de programação *Python*. Estes *posts* não continham nada sobre a biblioteca *Boost* da linguagem de programação C++. Existem outras tarefas de desenvolvimento em que o Google retorna alguns *posts* do SO que não estão relacionados com a respectiva API, como as tarefas 11 e 12 do *LINQ* - Cenário NKAE. Este comportamento inesperado desperta curiosidade pois definiu-se que a consulta deveria obrigatoriamente conter o nome da API (ex. *Boost*). A abordagem *Lucene+Score+How-to* não sofre deste problema pois foram filtrados somente *posts* do SO relacionados com a API de interesse usando o campo *tag*.

- Terceiro, alguns *posts* do SO recomendados pelo Google eram mais teóricos do que práticos. Por exemplo, para a tarefa 12.7 do *Boost "Combining multiple test cases in one test module"*, o Google retornou alguns *posts* do SO teóricos que explicam o

que é um teste unitário, um teste de regressão, etc. ao invés de um exemplo de uso utilizando a biblioteca *Boost* da linguagem de programação C++. Acredita-se que a ausência de um classificador *How-to* contribuiu para esse cenário indesejado para o Google.

Foram também identificadas algumas fraquezas na recomendação *Lucene+Score+How-to*:

- ❏ Primeiro, poderia ter sido um classificador binário personalizado pois a única distinção que interessa para este trabalho é entre pares pergunta-resposta *"how-to-do"* e *"not how-to-do"*. Indiscutivelmente, um classificador treinado apenas nessas duas categorias poderia obter maior precisão (i.e. taxa de acerto);
- ❏ Segundo, a escolha de *tags* para detectar tópicos parece ser pelo menos um pouco potencialmente problemática. As *tags* são fornecidas manualmente pelos participantes do SO, e portanto, não são confiáveis (ex. devido a problemas de sinonímia e polissemia, apesar dos melhores esforços da comunidade do SO). Essa realização levou a tentativas repetidas de prever as *tags* com base no texto da pergunta (ex. o trabalho de *Wang et al.* (WANG et al., 2014)).

Estes resultados sugerem que a construção de uma abordagem híbrida utilizando o motor de busca Google para combinar a semelhança textual e semântica melhorado com um classificador *How-to* proporcionaria os melhores resultados possíveis considerando as alternativas apresentadas.

Tabela 43 – Quantidade de Resultados Ruins e Ótimos por abordagem.

	#Resultados Ruins (Nota = 0)		#Resultados Ótimos (Nota = 4)	
Tópico	*Lucene+Score+How-to*	*Google*	*Lucene+Score+How-to*	*Google*
Swing	27,91%	21,66%	30,83%	48,75%
Boost	32,08%	46,66%	30%	25%
LINQ	31,81%	40,90%	24,09%	27,27%

4.4.1 Ameaças à Validade

Algumas ameaças devem ser consideradas na análise dos resultados apresentados.

A escolha das *tags* para detectar tópicos parece ser pelo menos um pouco potencialmente problemática. Por exemplo, se a *tag* não estiver correta (ex. a *tag* foi digitada incorretamente pelo usuário da ferramenta), a abordagem proposta não será capaz de retornar os *posts* presentes no banco de dados local do SO cujas perguntas estão marcadas com esta *tag* incorreta. Portanto, nenhum par Q&A será recomendado para o usuário final da abordagem proposta. *Tags* atribuídas a perguntas do SO tendem a conter ruídos e algumas perguntas do SO não estão bem marcadas (do inglês, *tagged*) (WANG et al.,

2014). A maioria dos *sites* de informações de *software* permitem que os usuários criem *tags* livremente. No entanto, essa liberdade tem um custo, pois as *tags* podem ser idiossincráticas devido à terminologia pessoal dos usuários (WANG et al., 2014) (GOLDER; HUBERMAN, 2006). Além disso, alguns *sites* de informações de *software* (ex. SO) exigem que os usuários adicionem no mínimo 3 *tags* no momento em que a pergunta é postada, mesmo que não conheçam as *tags* em circulação naquele momento. Como a marcação (i.e. *tagging*) é inerentemente um processo distribuído e não coordenado, muitas vezes questões semelhantes são marcadas de forma diferente (XIA et al., 2013). Por exemplo, no SO as *tags xmlparser*, *xml-parser* e *xmlparsing* são usadas para descrever um *parser* de um arquivo XML. Esta característica reduz a utilidade das *tags*, uma vez que os objetos relacionados não estão ligados entre si por uma *tag* comum e as informações relevantes tornam-se mais difíceis de serem recuperadas (WANG et al., 2014). Outro fenômeno em *sites* de informações de *software* são os sinônimos das *tags*. Este fenômeno refere-se às *tags* que são sintaticamente diferentes (i.e. são diferentes sequências de símbolos) mas são semanticamente a mesma coisa (i.e. possuem o mesmo significado) (XIA et al., 2013).

Uma maneira que o *website* pode facilitar a marcação precisa dos perguntas é recomendar *tags* para os usuários com base no conteúdo que eles geram. O desenvolvimento de um sistema de recomendação de *tags* para o conteúdo criado pelo usuário é um campo relativamente novo. Assim, a recomendação de *tags* é um campo no qual o estado da arte ainda está sendo desenvolvido ativamente, e os métodos mais precisos para recomendar *tags* ainda não foram estabelecidos. Foram descobertos dois artigos que detalham dois algoritmos, *TagCombine* (XIA et al., 2013) e *EnTagRec* (WANG et al., 2014) para recomendação de *tags* em *sites* como o SO. Esses dois algoritmos usam a aprendizagem da máquina para recomendação de *tags*.

Considerando a validade da construção, o processo de classificação de pares Q&A do SO é dependente do homem. Portanto, o desacordo entre as pessoas sobre a nota de um par Q&A é possível. Para mitigar essa ameaça, realizou-se um processo de avaliação em duas rodadas para melhorar a concordância entre os avaliadores medida através do *Kappa* Ponderado (COHEN, 1968). Outra ameaça nessa categoria ocorre ao selecionar uma classe para compor as consultas para as tarefas do Cenário KAE. Neste caso pode haver mais de uma classe em um trecho de código e diferentes escolhas podem levar a resultados diferentes. Essa ameaça é minimizada porque, mesmo quando há mais de uma classe no trecho de código-fonte, a classe escolhida é bastante óbvia porque os trechos de código tendem a ser pequenos, facilitando a identificação da classe dominante.

Uma outra ameaça à validade está relacionada ao processo de avaliação dos pares Q&A. Os mesmos foram avaliados sabendo se eles foram recomendados pelo Google ou pela abordagem *Lucene+Score+How-to*. Idealmente, a avaliação deveria ter sido cega, ou seja, sem saber qual abordagem recomendou o par Q&A.

Quanto à validade da conclusão, é possível identificar uma ameaça na definição de

medidas para o que significa um par Q&A útil no contexto de uma atividade de programação. Mesmo que a aderência semântica de um par Q&A (*Relev*) e a sua reprodutibilidade (*Reprod*) sejam representantes significativos da utilidade do mesmo, a adoção de outras medidas poderia refinar a avaliação destes pares Q&A.

Por fim, mas não menos importante, em relação à validade externa, os resultados apresentados neste trabalho referem-se à três APIs diferentes e pode-se observar uma variabilidade nos resultados dependendo da API. Existem vários fatores que podem afetar os resultados de uma API específica. Por exemplo, se a cobertura de uma API não é apropriada ou mesmo se a API não atrai o interesse da comunidade, não se espera que os resultados sejam promissores como mostrado neste trabalho. Além disso, para cada API, as tarefas escolhidas podem não representar a necessidade típica dos desenvolvedores durante o desenvolvimento de *software*. No entanto, a opção de selecionar essas tarefas a partir de documentos de terceiros foi uma conquista importante pois com isso elimina-se o viés de escolhas arbitrárias.

Capítulo 5

Trabalhos Relacionados

Este capítulo foi dividido em duas seções. A Seção 5.1 refere-se ao trabalho de utilizar o conhecimento presente no SO para auxiliar os desenvolvedores no processo de Depuração dos *API-usage-related bugs* (CAMPOS; MONPERRUS; MAIA, 2016). Já a Seção 5.2 refere-se ao trabalho de utilizar o conhecimento disponível no SO para auxiliar os desenvolvedores nas tarefas de desenvolvimento que envolvem utilização de APIs.

5.1 Utilização do conhecimento presente no SO para Depuração de *API-usage-related bugs*

O uso incorreto de APIs é uma causa prevalente de *bugs* de *software*, falhas e vulnerabilidades (MONPERRUS; MEZINI, 2013) (SUSHINE; HERBSLEB; ALDRICH, 2015) (AMANI et al., 2016) (FAHL et al., 2012) (EGELE et al., 2013) (NADI et al., 2016) (GEORGIEV et al., 2012) (CAMPOS; MONPERRUS; MAIA, 2016). Apesar das diversas tentativas de abordar o problema de uso incorreto de API, alguns estudos recentes mostraram que o problema ainda persiste. Um bom exemplo é o uso incorreto de APIs de criptografia Java que provocam muitas vulnerabilidades de segurança (EGELE et al., 2013) (NADI et al., 2016). Um uso incorreto de API é um uso de API que viola o contrato da API (AMANI et al., 2016).

Com o objetivo de endereçar este problema de pesquisa, várias técnicas de mineração de padrões de uso de API foram propostas. A Subseção 5.1.1 apresenta detalhadamente as principais técnicas, bem como suas características e estratégias. Além disso, diversos detectores de uso incorreto de API também foram propostos para lidar com este tópico de pesquisa, sendo algumas abordagens estáticas e outras dinâmicas. A Subseção 5.1.2 apresenta estes detectores, descrevendo as capacidades de cada detector com relação aos tipos de violações de API que cada um aborda bem como os respectivos elementos de uso de API. Estes, por sua vez, são elementos de programas que aparecem nos usos de API. São exemplos: chamadas de métodos, condições, iterações e tratamentos de exceções.

Além das Subseções 5.1.1 e 5.1.2, esta seção é composta por outras três subseções. A Subseção 5.1.3 revisa os trabalhos feitos para otimizar a eficácia da pesquisa em código. A Subseção 5.1.4 apresenta os trabalhos seminais em Depuração de Código envolvendo o uso do conhecimento disponível em serviços de mídia social como o SO. A Subseção 5.1.5 revisa alguns trabalhos feitos na área de Sistemas de Recomendação para Engenharia de *Software* dedicados para a recomendação de correção de *bugs*.

5.1.1 Mineração de Padrões de Uso de API

Ao longo da última década, o problema do aprendizado de API atraiu considerável interesse da comunidade de pesquisa. Neste contexto, diversas técnicas foram propostas com o objetivo de extrair automaticamente *padrões de uso de API* (Robillard et al., 2013). Por exemplo, alguns trabalhos anteriores bem conhecidos como MAPO (ZHONG et al., 2009) e UPMiner (Wang et al., 2013) mostraram que recomendar *padrões de uso de API* aos desenvolvedores facilita o processo de aprendizado. Todavia, até o momento, as ferramentas de mineração de API ainda não obtiveram ampla adoção em ambientes de desenvolvimento, como o *Eclipse* e o *Visual Studio*. Uma das razões para isso é que a qualidade dos padrões extraídos ainda não é alta o suficiente: os padrões retornados pelos métodos atuais são numerosos e altamente redundantes.

O primeiro algoritmo para mineração de padrões de uso de API a partir do código-fonte foi o MAPO, proposto por Xie e Pei (XIE; PEI, 2006) e estendido por Zhong *et al.* (ZHONG et al., 2009). MAPO (ZHONG et al., 2009) minera as sequências de chamadas de método (que invocam uma API desejada) de trechos de código recuperados por mecanismos de pesquisa de código. Essas sequências de métodos de API invocados são primeiro extraídas dos trechos de código e, em seguida, agrupadas de acordo com uma métrica de distância, calculada como uma média da similaridade de nomes de métodos, nomes de classes e os próprios métodos de API invocados. Para cada grupo, a ferramenta MAPO analisa as chamadas de API mais frequentes usando SPAM (AYRES et al., 2002) e as alimenta a um recomendador de uso da API que as classifica com base em sua semelhança com o contexto de código do desenvolvedor.

O UP-Miner (Wang et al., 2013) estende o MAPO em uma tentativa de reduzir ainda mais a redundância das sequências de chamadas de API extraídas. Isto é conseguido através de três abordagens principais: usando o minerador de sequência frequente fechado denominado BIDE (Wang; Han, 2004) que retorna apenas as sequências frequentes que não possuem subsequências com a mesma frequência; usando uma métrica de distância de agrupamento com base no conjunto de todos os n-gramas da sequência de chamadas da API e em uma etapa de agrupamento adicional nas sequências de chamadas de cada grupo. Ao contrário do MAPO, as sequências de chamadas da API são apresentadas como grafos probabilísticos, ranqueados de acordo com a sua frequência.

Acharya *et al.* (ACHARYA et al., 2007) apresentaram um *framework* para extrair padrões da API como pedidos parciais do código do cliente. Para esse objetivo, os rastreamentos de API estáticos sensíveis ao fluxo de controle são extraídos do código-fonte e os padrões sequenciais são calculados. Embora essa abordagem proponha uma representação para padrões de API, sugestões sobre o uso da API ainda estão faltando.

Fowkes *et al.* (FOWKES; SUTTON, 2016) propuseram PAM (Minerador de API Probabilístico), uma abordagem probabilística livre de parâmetros para minerar padrões de uso de API. PAM usa o algoritmo estrutural **Maximização de Expectativas (EM)** (DEMPSTER; LAIRD; RUBIN, 1977) para inferir os padrões de API mais prováveis do código do cliente, que são então classificados de acordo com a sua probabilidade. A abordagem PAM supera tanto o MAPO quanto o UP-Miner (menor redundância e maior precisão).

5.1.2 Detectores de Uso Incorreto de API

Um uso incorreto de API pode ser visto como um desvio de um padrão de uso de API frequente. Com o intuito de descobrir diversos usos incorretos de APIs, vários detectores foram propostos na literatura nos últimos anos.

PR-Miner (LI; ZHOU, 2005) é um detector de uso incorreto de API para a linguagem C. Ele codifica os usos como um conjunto de todos os nomes de funções chamados dentro da mesma função e, em seguida, emprega mineração de itens frequentes para encontrar os padrões com um suporte mínimo de 15 usos. *PR-Miner* se concentra na detecção de chamadas de método faltantes. A avaliação aplicou o *PR-Miner* a três projetos-alvo individualmente, encontrando assim as violações dos padrões específicos do projeto. Os autores revisaram as principais 60 violações relatadas em todos os projetos e encontraram 18,1% verdadeiros positivos (26,7%, 10,0% e 14,3% nos projetos individuais).

Colibri/ML (LINDIG, 2007) é um detector de uso incorreto para a linguagem C. Ele reimplementa o detector *PR-Miner* utilizando Análise do Conceito Formal (GANTER; WILLE, 1997) para fortalecer a base teórica da abordagem. Consequentemente, as suas capacidades são as mesmas que a do detector *PR-Miner*. A avaliação aplicou o *Colibri/ML* a cinco projetos-alvo, encontrando assim as violações de padrões específicos do projeto. Embora algumas violações detectadas são apresentadas no artigo, não são relatadas estatísticas sobre a qualidade das descobertas do detector.

Jadet (WASYLKOWSKI; ZELLER; LINDIG, 2007) é um detector de uso incorreto para a linguagem Java. Ele usa o detector *Colibri/ML*, mas em vez de apenas nomes de métodos, ele codifica a ordem de chamada de métodos e os receptores de chamadas em usos. Para isso, o detector *Jadet* constrói um grafo direcionado cujos vértices representam chamadas de método em um determinado objeto e cujas arestas representam os fluxos de controle. A partir deste grafo, ele deriva um par de chamadas para cada relacionamento de ordem de chamada. A avaliação aplicou o *Jadet* a cinco projetos-alvo, encontrando

assim as violações de padrões específicos do projeto. Os autores revisaram as principais 10 violações reportadas por projeto e encontraram 0-13% verdadeiros positivos. As outras descobertas foram classificadas como *"code smells"* (TUFANO et al., 2015) (i.e. sintomas de *design* pobre e escolhas de implementação) ou sugestões.

RGJ07 (RAMANATHAN; GRAMA; JAGANNATHAN, 2007b) é um detector de uso incorreto para a linguagem C. Ele codifica usos como conjuntos de propriedades para cada variável. As propriedades são comparações para literais, posições de argumento em chamadas de função e atribuições. Para cada chamada, ele cria um grupo dos conjuntos de propriedades dos argumentos da chamada. Para todos os grupos de uma função particular, aplica-se mineração de sequência para aprender sequências comuns de propriedades de fluxo de controle e mineração de itens frequentes para identificar todos os conjuntos comuns de todos os outros tipos de propriedade. O detector *RGJ07* foi projetado para detectar as condições faltantes no código. A avaliação aplicou o *RGJ07* a um único projeto, encontrando violações de padrões específicos do projeto. Os autores discutem vários exemplos de erros reais que sua abordagem detecta, mas não relatam nenhuma estatística sobre o desempenho de detecção.

Chronicler (RAMANATHAN; GRAMA; JAGANNATHAN, 2007a) é um detector de uso incorreto para a linguagem C. Ele minera relações frequentes de precedência de chamada a partir de um grafo de fluxo de controle inter-procedimental. Uma relação é considerada frequente, se mantiver em pelo menos 80% de todos os caminhos de execução. Os caminhos onde essas relações não são realizadas são relatados como violações. O detector *Chronicler* detecta chamadas de método faltantes e mal colocadas. A avaliação aplicou o *Chronicler* a cinco projetos, encontrando assim as violações de padrões específicos do projeto. Os autores comparam os protocolos identificados com os protocolos documentados para uma API e discutem alguns exemplos de erros reais encontrados por sua ferramenta.

Grouminer (NGUYEN et al., 2009) é um detector de uso incorreto para a linguagem Java. Ele cria uma representação de uso de objeto baseada em grafo (i.e. um *GROUM*) para cada método alvo. Um *GROUM* é um grafo acíclico direcionado cujos vértices representam chamadas de método, ramificações, iterações e cujas arestas codificam controle e fluxo de dados. O detector *Grouminer* minera subgrafos frequentes em conjuntos de tais grafos para detectar padrões de uso recorrentes com suporte mínimo de 6. Ele detecta chamadas de método faltantes e mal colocadas. A avaliação aplicou o *Grouminer* a nove projetos individualmente, encontrando violações dos padrões específicos do projeto. Os autores revisaram as 10 principais violações por projeto e confirmaram 12% como *bugs*. As demais violações foram classificadas como *"code smells"* (TUFANO et al., 2015).

AX09 (ACHARYA; XIE, 2009) é um detector de uso incorreto para a linguagem C. Ele é especializado na detecção de manipulação incorreta de erros, realizada através de códigos de erro retornados e verificados. O *AX09* detecta tratamentos de erros faltantes, bem

como as chamadas de método faltantes e mal colocadas entre as funções de tratamento de erros. A avaliação aplicou o *AX09* a três projetos individualmente, encontrando violações de padrões específicos do projeto. Os autores revisaram manualmente todas as descobertas do detector e confirmaram 80-93% como *bugs*.

Car-Miner (THUMMALAPENTA; XIE, 2009b) é um detector de uso incorreto para as linguagens C++ e Java. Ele é especializado na detecção de manipulação incorreta de erros. Para cada método analisado em um determinado contexto de código, o detector *Car-Miner* consulta um motor de busca para código-fonte com o objetivo de encontrar exemplos de uso. A partir desses exemplos, ele constrói um Grafo de Fluxo de Exceção (GFE), i.e. um grafo de fluxo de controle com arestas adicionais para fluxo excepcional. A partir do GFE, ele gera as sequências de chamadas normais que levam à chamada atualmente analisada e as sequências de chamadas de exceção que levam da chamada ao longo das arestas excepcionais. Posteriormente, ele minera as regras de associação entre sequências normais e sequências de exceção, com um suporte mínimo de 40%. *Car-Miner* detecta tratamento de exceção faltante, bem como chamadas de método faltantes e mal colocadas entre as funções de tratamento de erros. A avaliação aplicou o *Car-Miner* a cinco projetos. Uma vez que ele consulta um motor de busca para código-fonte com o objetivo de obter os exemplos de uso, ele detecta violações de padrões que ocorrem em diferentes projetos. Os autores revisaram manualmente todas as violações das 10 principais regras de associação para cada projeto e confirmaram que 41-82% identificam tratamento incorreto de erros. As demais violações foram classificadas como dicas para melhorar a qualidade do código.

Alattin (THUMMALAPENTA; XIE, 2009a) é um detector de uso incorreto para a linguagem Java. Ele é especializado em padrões alternativos para verificações de condição. Para cada método alvo m, este detector consulta um motor de busca para código-fonte para encontrar exemplos de uso. Para cada exemplo, ele extrai um conjunto de regras sobre as verificações de pré e pós-condição no receptor, os argumentos e o valor de retorno de m. Em seguida, ele aplica mineração de itens frequentes nessas regras para obter padrões com um suporte mínimo de 40%. Para cada padrão, ele extrai o subconjunto de todos os grupos que não aderem ao padrão e repete a mineração nesse subconjunto para obter padrões infrequentes com um suporte mínimo de 20%. Finalmente, ele combina todos os padrões frequentes e infrequentes para o mesmo método por disjunção. A avaliação aplicou o *Alattin* a 6 projetos. Uma vez que ele consulta um motor de busca para código-fonte com o objetivo de obter os exemplos de uso, ele detecta violações de padrões que ocorrem em diferentes projetos. Os autores revisaram manualmente todas as violações dos 10 principais padrões por projeto e confirmaram que 12% e 52% identificam as verificações de condição faltantes.

Tikanga (WASYLKOWSKI; ZELLER, 2009) é um detector de uso incorreto para a linguagem Java que se baseia no *Jadet*. Ele estende as propriedades simples da ordem

de chamada para fórmulas gerais da Lógica de Árvore de Computação (LAC) (i.e. uma ramificação da Lógica Temporal) sobre usos de objetos. Especificamente, ele usa fórmulas que exigem uma certa chamada, fórmulas que exigem duas chamadas em ordem e fórmulas que exigem uma certa chamada para acontecer após a outra. O detector *Tikanga* usa a verificação do modelo para determinar todas essas fórmulas com um suporte mínimo de 20 na base de código. As capacidades do detector *Tikanga* são as mesmas do *Jadet*. A avaliação aplicou a abordagem a 6 projetos individualmente, encontrando assim violações de padrões específicos de projeto. Os autores revisaram manualmente os primeiros 25% das violações por projeto e confirmaram até 6% como *bugs*. As demais violações foram classificadas como *"code smells"* (TUFANO et al., 2015).

Dmmc (MONPERRUS; MEZINI, 2013) é um detector de uso incorreto para a linguagem Java, especializado em chamadas de método faltantes. A detecção é baseada em usos de tipo, ou seja, conjuntos de métodos chamados de uma instância de um determinado tipo em um determinado método. Dois usos são **exatamente semelhantes** se seus respectivos conjuntos coincidem e **quase semelhantes** se um deles contiver exatamente um método adicional. A detecção baseia-se no pressuposto de que as violações deveriam ter apenas alguns usos **exatamente semelhantes**, mas muitos usos **quase semelhantes**. O *Dmmc* detecta usos incorretos contendo exatamente uma chamada de método faltante. A avaliação aplicou o *Dmmc* a um único projeto, encontrando assim violações específicas do projeto. Os autores revisaram manualmente todas as 19 descobertas e confirmaram 84% como sendo *bugs*. As demais descobertas foram classificadas como soluções alternativas para *bugs* dentro de uma API utilizada.

DroidAssist (NGUYEN et al., 2015) é um detector de uso incorreto para Java *Bytecode* da plataforma *Android*. Ele gera sequências de chamadas de método a partir do código-fonte e aprende um modelo oculto de *Markov* (RABINER; JUANG, 1986) a partir delas, para calcular a probabilidade de uma determinada sequência de chamadas. Em seguida, o detector *DroidAssist* explora diferentes modificações da sequência (i.e. adição, substituição e remoção de chamadas) para encontrar uma sequência ligeiramente modificada e mais provável. Isso permite que ele detecte chamadas de método faltantes, mal colocadas e supérfluas. Além disso, ele é capaz de até mesmo sugerir soluções para elas. Uma avaliação desta abordagem não é fornecida no respectivo artigo.

5.1.3 Técnicas de Pesquisa de Código

O presente trabalho de doutorado também está associada aos campos de motores de busca e recuperação de amostra de código da *Web*. Diferentes técnicas de mineração de código e ferramentas foram propostas para retornar componentes de *software* relevantes a partir de diferentes repositórios, como é mostrado na Tabela 44. As abordagens que aparecem em negrito na Tabela 44 são mais similares à abordagem proposta na tese de doutorado pois utilizam a base de conhecimento do *Stack Overflow*.

Tabela 44 – Resumo de técnicas e ferramentas de mineração de código. A coluna Granularidade especifica como os usuários especificam as consultas (Chamada de **A**PI, **C**onceito ou Caso de **T**este) e como os resultados da pesquisa são retornados para cada abordagem (Repositório da **A**plicação, **F**ragmento de Código ou **M**ódulo composto por um conjunto de classes da aplicação). A coluna Corpora especifica o escopo da pesquisa (**C**ódigo ou **D**ocumentos), seguido pela coluna Expansão que especifica se uma determinada abordagem utiliza a técnica de expansão de consulta para melhorar a precisão da busca.

Abordagem	Granularidade		Corpora	Expansão
	Entrada	Saída		
CodeFinder (HENNINGER, 1996)	C	M	D	Sim
CodeBroker (YE; FISCHER, 2005)	C	M	D	Sim
Mica (STYLOS; MYERS, 2006)	C	F	C	Sim
Prospector (MANDELIN et al., 2005)	A	F	C	Sim
Hipikat (ČUBRANIĆ, D. MURPHY, G., 2003)	C	A	C,D	Sim
xSnippet (SAHAVECHAPHAN; CLAYPOOL, 2006)	A	F	D	Sim
Strathcona (HOLMES; WALKER; MURPHY, 2006a)	C	F	C	Sim
AMC (HILL; RIDEOUT, 2004)	C	F	C	Não
Google Code	A,C	A,F,M	C,D	Não
Sourceforge	C	A	D	Não
SPARS-J (INOUE et al., 2003)	C	M	C	Não
Sourcerer (LINSTEAD et al., 2009)	C	A,F,M	C	Não
Sourcerer API Search (BAJRACHARYA; OSSHER; LOPES, 2010)	A,C	F	C	Não
CodeGenie (LEMOS et al., 2009)	T	F,M	C	Não
SpotWeb (THUMMALAPENTA; XIE, 2008)	C	M	C	Sim
ParseWeb (THUMMALAPENTA; XIE, 2007)	A	F	C	Sim
S^6 (REISS, 2009)	A,C,T	F	C	Manual
Krugle	A,C	A,F,M	C,D	Não
Koders	A,C	A,F,M,	C,D	Não
SNIFF (CHATTERJEE; JUVEKAR; SEN, 2009)	A,C	F,M	C,D	Sim
Blueprint (BRANDT et al., 2010)	A,C	F	C	Não
Portfolio (MCMILLAN et al., 2011)	A,C	A,F	C	Sim
Exemplar (MCMILLAN et al., 2012)	A,C	A,F,M	C,D	Não
CodeHow (LV et al., 2015)	A,C	F	C,D	Sim
RACK (RAHMAN; ROY; LO, 2016)	C	F	C,D	Sim
Prompter (PONZANELLI et al., 2016)	**A,C**	**F**	**C,D**	**Sim**
Lucene+Score+How-to (CAMPOS; SOUZA; MAIA, 2016)	**A,C**	**F**	**C,D**	**Sim**
DeepCS (GU; ZHANG; KIM, 2018)	C	F	C,D	Não
FaCoY (KIM et al., 2018)	**A**	**F**	**C,D**	**Sim**

Umarji *et al.* (UMARJI; SIM; LOPES, 2008) investigaram os hábitos dos desenvolvedores na busca de código na *Internet*. Sim *et al.* (SIM et al., 2011) investigaram como *sites* de recuperação de informações de propósito geral (ex. *Google*) supera *sites* personalizados para pesquisa de código (ex. *Krugle*) e reutilização de componentes (ex. *SourceForge*) na recuperação de amostras de código da *Web*. Na abordagem apresentada neste trabalho de doutorado, realizou-se recuperação de código no SO. Quando trechos de código são retornados pelos motores de busca, o desenvolvedor tem que avaliar sua validade. Como a abordagem proposta recomenda conteúdo do SO, os trechos de código já foram avaliados previamente pela comunidade.

McMillan *et al.* (MCMILLAN et al., 2012) criaram uma abordagem chamada de EXEMPLAR para encontrar projetos de *software* altamente relevantes a partir de grandes arquivos de aplicações. Após o programador digitar uma consulta em linguagem natural que contém conceitos de alto nível, EXEMPLAR retorna aplicações que implementam estes conceitos. EXEMPLAR lista as aplicações de forma ranqueada em três passos. Primeiro, é considerado a descrição das aplicações. Segundo, é examinado as chamadas de método de API (*Application Programming Interface*) utilizadas por estas aplicações. Terceiro, é analisado o fluxo de dados entre estas chamadas de API.

Henninger (HENNINGER, 1996) criou CODEFINDER, um sistema de recomendação que encontra código similar utilizando um método de busca denominado *Spreading Activation* para redes associativas, redes neurais ou redes semânticas. Este método pode ser aplicado na área de Recuperação da Informação por meio de uma rede de nós que representam documentos e os termos contidos nestes documentos (no caso do trabalho de Henninger, os termos que aparecem no código). Yunwen Ye e Gerhard Fischer (YE; FISCHER, 2002) criaram CODEBROKER, um sistema que utiliza código-fonte e comentários escritos por programadores para consultar repositórios de código para encontrar artefatos relevantes. Ao contrário do EXEMPLAR (MCMILLAN et al., 2012), CODEBROKER é dependente de descrições de documentos e nomes significativos de variáveis de programa e tipos, e esta dependência muitas vezes leva a uma menor precisão de projetos recomendados.

Jeffrey Stylos e Brad A. Myers (STYLOS; MYERS, 2006) propuseram MICA, uma ferramenta que retorna trechos de código ao invés de aplicações de *software* relevantes (no caso do Exemplar (MCMILLAN et al., 2012)). Mica utiliza documentação de ajuda para refinar os resultados da busca. Além disso, ela aumenta os resultados de busca *Web* padrão para ajudar os programadores a encontrar as classes de API certas e métodos dado a descrição da funcionalidade desejada. MICA também ajuda os programadores a encontrar exemplos de código quando eles já sabem quais métodos usar.

Chatterjee *et al.* (CHATTERJEE; JUVEKAR; SEN, 2009) propuseram uma nova técnica de pesquisa de código chamada de SNIFF, que estende a ideia de utilizar a documentação para chamadas de API para pesquisa de código-fonte. A técnica deles é

baseada na observação de que os métodos de biblioteca que o código do usuário invoca são frequentemente bem documentados. SNIFF utiliza a documentação dos métodos da biblioteca para adicionar significado na consultas em linguagem natural feitas utilizando o Inglês.

Sawadsky *et al.* apresentaram FISHTAIL (SAWADSKY; MURPHY, 2011), um *plug-in* do Eclipse que ajuda os desenvolvedores na descoberta de trechos de código-fonte na *Web* relevantes com a tarefa de desenvolvimento em mãos. FISHTAIL sugere trechos de código-fonte de acordo com o nome da entidade do programa que mais foi alterado. No trabalho do doutorado, será possível obter trechos de código-fonte a partir do *Stack Overflow*. Como esses trechos foram obtidos a partir do *site Stack Overflow*, tais trechos de código-fonte já foram avaliados pela comunidade. Assim, o desenvolvedor não precisa avaliar a sua validade.

Holmes *et al.* notaram que frequentemente, os desenvolvedores possuem dificuldades ao utilizar APIs não familiares. Estes desenvolvedores procuram por exemplos de uso correto da API para ajudá-los a completar suas tarefas. Infelizmente, estes exemplos nem sempre são fornecidos na documentação do projeto. Eles criaram um sistema de recomendação denominado STRATHCONA (HOLMES; WALKER; MURPHY, 2006b) para ajudar os desenvolvedores a localizarem exemplos de código-fonte relevantes a partir de um repositório de código-fonte. Isto é feito extraindo informação estrutural do ambiente de desenvolvimento integrado do desenvolvedor (IDE) e utilizando esta informação para localizar exemplos que possuem estrutura similar. Quatro heurísticas foram utilizadas para medir a similaridade estrutural entre o código do desenvolvedor e um dado exemplo do repositório.

Kim *et al.* apresentaram uma abordagem denominada *FaCoY* (KIM et al., 2018) para encontrar estaticamente fragmentos de código que podem ser semanticamente semelhantes ao código de entrada do usuário. Esta abordagem pode ser útil na recomendação de código e de correções para defeitos no *software*. *FaCoY* aproveita os dados do *Stack Overflow* para melhorar a detecção de clones. Kim *et al.* mostraram que a abordagem *FaCoY* alcança melhor precisão do que os motores de busca *online* que recebem código como entrada e retornam exemplos de código similares (ex. *Krugle* [1] (Julho de 2017)) e encontra mais clones de código semânticos no *benchmark* **BigCloneBench** (SVAJLENKO et al., 2014) do que os detectores de clone do estado da arte, como *Deckard* (JIANG et al., 2007), *NiCad* (CORDY; ROY, 2011), *CCFinderX* (KAMIYA; KUSUMOTO; INOUE, 2002) e *SourcererCC* (SAJNANI et al., 2016).

5.1.4 Depuração utilizando o conhecimento disponível no SO

Monperrus *et al.* (MONPERRUS; MAIA, 2014) propuseram um sistema de depuração para Javascript que foi designado como um sistema de recomendação. Eles realizaram um

[1] http://krugle.com/

estudo empírico para avaliar se o SO contém muitas correções de *API-usage-related bugs* (também chamado pelos autores de *"crowd bugs"*) com o intuito de motivar a proposta de um sistema de recomendação para *API-usage-related bugs* baseado na sabedoria da multidão. O estudo revelou que existe um número grande de Q&As que estão relacionados com correções de *API-usage-related bugs* para a linguagem de programação Javascript. Carzaniga *et al.* (CARZANIGA et al., 2010) também lidou com *bugs* em Javascript. Eles provaram que a *Web* contém algumas descrições desta classe de *bug*. Entretanto, eles trataram estes *bugs* a partir da perspectiva do usuário, que é provido com alternativas a fim de completar algumas requisições na *Web*. Ao contrário, no trabalho de Monperrus *et al.* (MONPERRUS; MAIA, 2014), a ideia é prover recomendações alternativas para o desenvolvedor que criou um ponto de interrupção (do inglês, *breakpoint*) em seu código.

Chen *et al.* (CHEN; KIM, 2015) propuseram uma nova técnica de depuração baseada na "sabedoria da multidão" com o objetivo de detectar fragmentos de código defeituosos no código fonte dos desenvolvedores e prover soluções sugeridas pela multidão contendo explicações para os desenvolvedores. Eles compararam os seus resultados com outras 3 ferramentas populares de análise estática de código-fonte (i.e. *FindBugs, JLint e PMD*). A abordagem de Chen *et al.* (CHEN; KIM, 2015) revelou 189 *warnings* e 171 (90.5%) deles foram confirmados pelos desenvolvedores de 8 projetos de alta qualidade e bem mantidos. Destes 171 *bugs* identificados pela abordagem deles, somente a ferramenta *FindBugs* detectou 6 deles enquanto as ferramentas *JLint* e *PMD* não detectaram nenhum *bug*.

5.1.5 Sistemas de Recomendação para Engenharia de *Software* dedicados para as Tarefas de Correção de *Bugs*

Watson *et al.* (WATSON; LI; GODWIN, 2012) propuseram BLUEFIX, uma ferramenta *online* designada para auxiliar os desenvolvedores com diagnósticos de erros e reparo. A abordagem deles está direcionada a prover o entendimento de erros de compilação em programas Java e o protótipo deles está integrado dentro da IDE *BlueJ*. Já o trabalho de Monperrus *et al.* (MONPERRUS; MAIA, 2014) trata outro tipo de erro, i.e. erros em tempo de execução em Javascript e o protótipo deles está integrado dentro do depurador *Firebug* (um depurador Javascript para o navegador *Web* Firefox).

Hartmann *et al.* (HARTMANN et al., 2010) inventaram um sistema de recomendação para corrigir erros de compilação. O sistema deles é alimentado com dados de monitoramento. Kallenbach (MUJUMDAR et al., 2011) ampliaram o sistema para erros em tempo de execução em *Ruby*. Ao contrário de construir um banco de dados com dados específicos, o presente trabalho de Doutorado tem o objetivo de construir um índice de documentos específico para a correção de *API-usage-related bugs* utilizando o motor de busca *Apache*

Solr [2]. Este índice será composto por *posts* do SO que devem respeitar as seguintes regras: 1) Possuir pelo menos um trecho de código na pergunta; 2) Ser da linguagem Java (ou seja, marcados com a *tag* "java"); 3) Conter respostas para as perguntas.

Ponzanelli *et al.* (PONZANELLI et al., 2014) propuseram uma abordagem de recomendação que, dado um contexto de código na IDE, automaticamente retorna discussões pertinentes a partir do SO, avalia a relevância delas, e se, um dado limiar de confiança for ultrapassado, notifica o desenvolvedor sobre a ajuda disponível. A abordagem deles foi implementada em um *plug-in* para Eclipse IDE denominado PROMPTER. A abordagem proposta neste trabalho de Doutorado também leva em conta o contexto de código que o desenvolvedor está trabalhando com o objetivo de sugerir discussões do SO relevantes. Entretanto, existem algumas diferenças entre os dois trabalhos: (i) a abordagem deles é mais geral e trata um escopo de problema diferente (i.e. eles não limitam o escopo deles para *"API-usage-related bugs"*); (ii) eles não utilizam o *Apache Solr* como motor de busca. Ao invés disso, eles usam outros motores de busca de propósito geral (i.e. *Google* e *Bing*); (iii) eles implementam uma técnica de busca de código diferente das técnicas de pré-processamento de código investigadas no presente trabalho de Doutorado (ex. *lsab-java* e *lsab-js*).

5.1.6 Mineração de Padrões de *Bugs*

Existe uma rica literatura para detecção de *bugs*. Uma área de pesquisa ativa consiste em encontrar padrões comuns de *bugs* e usar esses padrões para detectar *bugs* com antecedência (ex. *ESC/Java* (FLANAGAN et al., 2002), *FindBugs* (HOVEMEYER; PUGH, 2004),*PMD* (COPELAND, 2005)). Existem muitos verificadores estáticos que detectam erros no código-fonte do programa ou em binários executáveis usando código-fonte ou padrões binários. *FindBugs* (HOVEMEYER; PUGH, 2004) define 50 padrões de *bug* propensos a erros, enquanto o *ESC/Java* (FLANAGAN et al., 2002) utiliza verificação de tipo e especificação (um tipo de padrão) para detectar erros. O *PMD* (COPELAND, 2005) usa padrões problemáticos de estilo de código-fonte, como variáveis não utilizadas, blocos `catch` vazios e criação de objetos desnecessários, para detectar potenciais *bugs*. Esses sistemas baseiam seus padrões de *bug* em erros de programa bem conhecidos, e usam análise estática para detectar instâncias de padrão de *bug*.

Pan *et al.* (PAN; KIM; WHITEHEAD JR., 2009) identificaram manualmente 27 padrões de correção de *bugs* presentes na informação histórica de 7 projetos Java de código aberto. Eles utilizaram diferenciação de nível de linha para extrair e raciocinar sobre os padrões de correção de *bugs* (também conhecidos como padrões de reparo) para o Reparo Automático de Programas. Os padrões de reparo que eles descobriram possuem granularidade grossa e não identificam a causa raiz dos *bugs*.

[2] http://lucene.apache.org/solr/

Alguns trabalhos combinaram análise estática ou dinâmica e técnicas de mineração de repositórios de *software* para identificar *bugs*. Williams & Hollingsworth (WILLIAMS; HOLLINGSWORTH, 2005) apresentaram um método que utiliza o histórico de alterações de código-fonte de um projeto de *software* para orientar e refinar a busca por *bugs*. Com base nos dados recuperados do repositório de código-fonte, eles implementaram um verificador de código-fonte estático que procura por um *bug* comumente corrigido e usa informações extraídas automaticamente do repositório de código-fonte para refinar seus resultados. Livshits & Zimmermann (LIVSHITS; ZIMMERMANN, 2005) descobriram padrões de *bug* específicos de aplicação (i.e. métodos que devem ser reparados, mas não são) usando a mineração de regras de associação em dois projetos Java. Eles apresentaram DynaMine, uma ferramenta para aprender padrões comuns de uso a partir das histórias de revisão de grandes sistemas de *software*.

Hanam *et al.* (HANAM; BRITO; MESBAH, 2016) propuseram uma nova técnica semi-automática denominada BugAID, para descobrir os padrões de *bug* mais prevalentes e detectáveis na linguagem JavaScript. BugAID é baseado em Aprendizado de Máquina não-supervisionado (mais especificamente, a técnica BugAID utiliza o algoritmo de *Clustering* DBSCAN (ESTER et al., 1996)) e Diferenciação de Árvores Sintáticas Abstratas (ASTs) provenientes de correções de *bugs* no código JavaScript.

5.2 Utilização do Conhecimento Disponível em Sítios Q&A para Auxílio às Tarefas que Envolvem Uso de API

Esta seção foi dividida em duas subseções. A Subseção 5.2.1 revisa alguns trabalhos feitos na literatura para categorizar os tipos de perguntas realizadas no SO. Já a Subseção 5.2.2 revisa alguns trabalhos na área de Sistemas de Recomendação para Engenharia de *Software* dedicados a auxiliar os desenvolvedores nas tarefas de desenvolvimento que envolvem uso de API.

5.2.1 Categorização das perguntas do SO

Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) analisaram dados provenientes do SO para categorizar os tipos de perguntas que são realizadas e para explorar quais perguntas são bem respondidas pela comunidade do SO e quais permanecem sem resposta. Os resultados preliminares deles indicaram que *sites* Q&A são particularmente efetivos para revisões de código e perguntas conceituais. Eles analisaram os textos dos títulos e conteúdo de 385 perguntas do SO e encontraram as seguintes categorias ordenadas por sua frequência: *how-to*, *discrepancy*, *environment*, *error*, *decision help*, *conceptual*, *review*,

non-functional, *novice* e *noise*. Eles também fizeram perguntas sobre o impacto das mídias sociais no conhecimento de desenvolvimento de *software*, e como isso poderia influenciar os hábitos dos desenvolvedores.

Nasehi *et al.* (NASEHI et al., 2012b) mostraram que os tipos de perguntas do SO podem ser descritas baseados em duas diferentes dimensões. A primeira dimensão trata do tópico da pergunta: ela mostra a tecnologia principal que a questão gira em torno e normalmente pode ser identificada a partir das *tags* da pergunta. A segunda dimensão é sobre as principais preocupações dos questionadores e o que eles querem resolver. Eles identificaram quatro tipos nesta última dimensão: *Debug/Corrective*, *Need-to-Know*, *How-To-Do-It*, and *Seeking-Different-Solution*. Ao contrário de Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) e Nasehi *et al.* (NASEHI et al., 2012b), neste trabalho de doutorado foi realizada uma categorização de pares Q&A do SO ao invés de perguntas do SO.

5.2.2 Sistemas de Recomendação para Engenharia de *Software* dedicados para as Tarefas de Desenvolvimento

Holmes *et al.* (HOLMES; WALKER; MURPHY, 2006a) discutiram no estudo deles que frequentemente os desenvolvedores ficam perdidos quando tentam usar uma API, inseguros de como fazer progressos em uma tarefa de programação. Eles apresentaram STRATHCONA, um *plug-in* para Eclipse IDE que utiliza a estrutura do código-fonte sob desenvolvimento para encontrar exemplos de código relevantes em um repositório. Similarmente, a abordagem apresentada neste trabalho aborda o mesmo problema, porém utiliza outra estratégia: aproveitar a "sabedoria da multidão" disponível no SO para recomendar informação que pode auxiliar os desenvolvedores a resolver tarefas de programação usando uma determinada API.

Ponzanelli *et al.* (PONZANELLI; BACCHELLI; LANZA, 2013b) apresentaram uma abordagem integrada e amplamente automatizada para auxiliar os desenvolvedores a aproveitarem a "sabedoria da multidão" dos serviços Q&A. Eles implementaram SEAHAWK, um sistema de recomendação na forma de *plug-in* para Eclipse IDE designado para aproveitar esta "sabedoria da multidão" presente no SO dentro do IDE. Este *plug-in* automaticamente formula consultas a partir do contexto atual na IDE e apresenta uma lista ranqueada e interativa de resultados. SEAHAWK permite aos usuários identificar pedaços de discussão individuais e importar trechos de código através de um simples *drag & drop*.

Sawadsky *et al.* apresentaram FISHTAIL (SAWADSKY; MURPHY, 2011), um *plug-in* do Eclipse que ajuda os desenvolvedores na descoberta de trechos de código-fonte na *Web* relevantes com a tarefa de desenvolvimento em mãos. FISHTAIL sugere trechos de código-fonte de acordo com o nome da entidade do programa que mais foi alterada.

Neste trabalho de doutorado, algumas atividades (Cenário KAE) utilizadas para avaliar a abordagem, focou-se também em nomes de entidades (ex. nomes de classes) utilizando as mesmas como parte da consulta feita na estratégia de recomendação proposta.

Cordeiro *et al.* (CORDEIRO; ANTUNES; GOMES, 2012) apresentaram um *plug-in* para Eclipse que auxilia os desenvolvedores nas tarefas de desenvolvimento. Baseado nos rastros de exceção da pilha de execução de chamadas de métodos obtidos a partir do console do IDE, eles sugerem documentos relacionados a partir do SO. Ao invés de focar em rastros de pilha, este trabalho focou no título da tarefa de desenvolvimento para consultar um índice relacionado com a API que o desenvolvedor está encontrando dificuldades e recomenda pares Q&A ao desenvolvedor. Este índice composto por pares *How-to-do* referente a uma determinada API é construído previamente na abordagem proposta.

Takuya et al. apresentaram SELENE (TAKUYA; MASUHARA, 2011), uma ferramenta de recomendação de código-fonte baseada em um motor de busca associativo. Esta ferramenta espontaneamente procura e mostra trechos de código-fonte de programas enquanto o desenvolvedor está editando um texto do programa. Através do uso deste motor de busca associativo, a ferramenta consegue pesquisar em um repositório contendo 2 milhões de trechos de código-fonte de programas dentro de poucos segundos. Este trabalho também está relacionado com a área de motores de busca (*search engines*). Todavia, é sugerido pares Q&A retirados do SO de forma a enriquecer a informação provida pelos trechos de código-fonte.

Holmes *et al.* apresentaram DEEPINTELLISENCE (HOLMES; BEGEL, 2008b), um *plug-in* para *Visual Studio* IDE que estabelece *links* entre as entidades do código-fonte e relatórios de *bugs*, *emails*. Similarmente, ČubraniĆ et al. criaram o HIPIKAT (ČUBRANIĆ, DAVOR AND MURPHY, GAIL C. AND SINGER, JANICE AND BOOTH, KELLOGG S., 2004), que é um sistema de recomendação desenvolvido para auxiliar os novatos em um projeto de *software*. Este sistema recomenda itens a partir de relatórios de problemas, grupos de notícias e artigos. Alternativamente, este trabalho, ao invés de fornecer os recursos a partir do conhecimento que está dentro do projeto, foca na recomendação de conteúdo a partir do SO.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho de doutorado apresenta duas abordagens de recomendação diferentes que utilizam em comum o conhecimento disponível no SO, uma vez que foram estudados dois problemas distintos que acometem os desenvolvedores de *software* nas suas atividades diárias. O **primeiro problema** está associado aos *bugs* que são introduzidos pelos desenvolvedores oriundos de uma utilização incorreta de elementos da API (ex. chamadas de método). Já o **segundo problema** consiste na dificuldade que os desenvolvedores possuem em realizar certas tarefas de programação com alguma API não-familiar.

Para tratar o **primeiro problema**, foi proposta uma abordagem de recomendação com o objetivo de recomendar correções para *bugs* relacionados com o uso incorreto de API (i.e. *API-usage-related bugs*). Assim, esta abordagem foi designada para receber como entrada trechos de código suspeitos de conter este tipo de *bug*. Trata-se de uma abordagem inédita pois os motores de busca de propósito geral (ex. *Google*, motor de busca do SO) e de código-fonte (ex. *Exemplar* (MCMILLAN et al., 2012)) não foram projetados para lidar com esse tipo de consulta. Desta forma, eles não são capazes de lidar com esta classe particular de *bugs*.

Para a condução do experimento, foram consideradas funções de pré-processamento para trechos de código-fonte escritos nas linguagens de programação Java e JavaScript. Foi construído também um *data set* composto por 30 trechos de código-fonte com potenciais *API-usage-related bugs* extraídos a partir do *site OpenHub Code Search* com o objetivo de avaliar a abordagem de recomendação proposta. Os resultados mostraram uma vantagem clara de aproximadamente 40% da abordagem que **utiliza o filtro de palavras-chave** em relação à abordagem em que **este filtro não é utilizado na consulta** do *Apache Solr*. Além disso, os resultados são encorajadores: para 66,67% dos trechos de código-fonte Java com potenciais *API-usage-related bugs* extraídos do *site OpenHub*, foram encontradas as correções dos mesmos no top-10 resultados recomendados pela abordagem de recomendação proposta. Com relação aos trechos de código-fonte JavaScript presentes neste *data set*, para 40% deles, foram encontradas as correções dos mesmos no top-10. Estes resultados sugerem que as abordagens (i.e. *lsab-java* e *lsab-js* combinadas com o

filtro de palavras-chave) superam o *Google* e o SO na busca por correções para potenciais *API-usage-related bugs* presentes em projetos reais de *software*.

Para abordar o **segundo problema**, foi proposta uma estratégia de recomendação com o objetivo de recomendar uma lista ranqueada de pares Q&A do SO para auxiliar os desenvolvedores durante as suas tarefas de programação com APIs. O critério de ranqueamento leva em conta a semelhança textual dos pares Q&A em relação ao problema do desenvolvedor, a qualidade destes pares e a caracterização *How-to* dos mesmos. Foram conduzidos experimentos considerando 35 tarefas de programação distribuídas em três diferentes tópicos (*Swing*, *Boost* e *LINQ*) amplamente utilizados pela comunidade de desenvolvimento de *software*.

Uma análise manual qualitativa foi realizada nos pares Q&A recomendados considerando dois critérios de avaliação: **Relevância** e **Reprodutibilidade**. Obteve-se um valor de NDCG igual a 0,3583 para o primeiro critério e 0,5243 para o segundo critério. Pode-se afirmar que o valor do NDCG_{Relev} para o *Google* foi 5,62% maior do que a abordagem *Lucene+Score+How-to* e o valor do NDCG_{Reprod} para o *Google* foi 4,75% maior do que a abordagem *Lucene+Score+How-to*. Esses resultados mostram que a qualidade geral dos pares Q&A recomendados pelo *Google* foi um pouco melhor do que a abordagem *Lucene+Score+How-to*. Esta, por sua vez, obteve os seguintes resultados: para 27 das 35 tarefas de programação (77,14%), no mínimo um par Q&A recomendado provou ser útil para o problema de programação alvo. Além disso, para todas as 35 tarefas de programação, no mínimo um par Q&A recomendado tinha um fragmento de código-fonte reproduzível ou quase reproduzível. Esses resultados sugerem que a abordagem proposta supera os resultados obtidos em um trabalho relacionado (i.e. *Seahawk* (PONZANELLI; BACCHELLI; LANZA, 2013b)).

Para avaliar a acurácia da recomendação realizada pela abordagem *Lucene+Score+How-to*, realizou-se também uma comparação com o motor de busca *Google*. A abordagem *Lucene+Score+How-to* alcançou melhor desempenho do que o *Google* no tópico *Boost*. Além disso, a abordagem proposta neste trabalho de doutorado obteve menos resultados ruins nos tópicos *Boost* e *LINQ* (32,08% e 31,81%, respectivamente) do que o *Google* (46,66% e 40,90%, respectivamente). Por outro lado, o *Google* apresentou um desempenho melhor do que a abordagem *Lucene+Score+How-to* no tópico *Swing*.

O mecanismo de pesquisa do *Google* possui dois recursos importantes que o ajudam a produzir resultados de alta precisão. Primeiro, ele faz uso da estrutura de *links* da *Web* para calcular um ranqueamento de qualidade para cada página da *Web*. Este ranqueamento é denominado *PageRank* (PAGE et al., 1999). Segundo, o *Google* faz uso do texto de âncora (i.e. texto visível em um *hyperlink*) para melhorar os resultados de busca (BRIN; PAGE, 1998). Isso tem várias vantagens. Primeiro, as âncoras geralmente fornecem descrições mais precisas das páginas *Web* do que as próprias páginas. Segundo, as âncoras podem existir para documentos que não podem ser indexados por um motor

de busca baseado em texto, como as imagens, programas e bancos de dados. Esses dois recursos não estão presentes no motor de busca *Lucene*. Além disso, o *Lucene* não é eficiente em processar documentos com campos de texto grandes devido ao custo envolvido com o número de iterações através do fluxo de entrada, que deve ser feito para produzir *tokens* válidos (SU, 2002).

Assim, sugere-se que a abordagem ideal para recomendação deste primeiro problema é usar o motor de busca do *Google* juntamente com o classificador *How-to*. Observou-se que alguns *posts* do SO recomendados pelo *Google* eram mais teóricos (i.e. *posts* conceituais) do que práticos (i.e. *posts* que mostram passo a passo como resolver uma tarefa de programação desejada com uma determinada API). Acredita-se que a ausência de um classificador *How-to* contribuiu para este cenário indesejado para o *Google*.

Outro problema com a recomendação do *Google* refere-se ao fato de que foi definido que a consulta de entrada para o *Google* deveria conter obrigatoriamente o nome da API. Apesar disso, o *Google* retornou resultados que não estavam relacionados com a respectiva API de pesquisa. A abordagem de recomendação *Lucene+Score+How-to* não sofre deste problema pois foram filtrados apenas *posts* do SO relacionados com a API de interesse usando o campo de *tag*. Portanto, sugere-se que esta "abordagem híbrida" poderia usar o motor de busca *Google* ao invés do motor de busca *Lucene* e depois usar o classificador *How-to* para filtrar os *posts How-to* pela *tag* (i.e. o nome da *tag* corresponde ao nome da API de pesquisa).

Como trabalho futuro, outros classificadores e outros atributos podem ser investigados com o objetivo de melhorar a taxa de acerto e assim, o ranqueamento geral da abordagem de recomendação proposta. Além disso, novas formas de geração da consulta e a respectiva indexação podem ser projetados. Por exemplo, a consulta poderia ser extraída a partir do código-fonte contextual onde o desenvolvedor está trabalhando. Além disso, novos mecanismos de busca além do *Lucene* poderiam ser investigados com o intuito de melhorar o ranqueamento.

Um outro trabalho futuro poderia ser conduzido com tópicos menos populares para investigar o impacto da popularidade do tópico no resultado. Para tópicos bastante populares, a chance de ter boas respostas no SO é maior. Mesmo os assuntos que estão nos *cookbooks* podem apresentar variabilidade em sua popularidade.

Em suma, como dito anteriormente na **Declaração de tese** (Capítulo 1), esta tese propõe soluções customizadas de busca que levam em consideração os aspectos intrínsecos do desenvolvimento e depuração de *software*. O motivo de defender esta tese está ligado ao fato de que é viável construir estas soluções na busca do conhecimento presente em serviços Q&A.

A Seção 6.1 lista a produção bibliográfica resultante do presente trabalho de doutorado, bem como alguns trabalhos anteriores publicados previamente ao doutorado.

6.1 Contribuições em Produção Bibliográfica

Como resultado deste trabalho de doutorado, foram publicados os seguintes artigos:

- Eduardo Campos, Martin Monperrus, Marcelo Maia. *Searching Stack Overflow for API-usage-related Bug Fixes Using Snippet-based Queries*. Proc. of 26th International Conference on Computer Science and Software Engineering (CASCON), Toronto, Canada, p. 232–242, 2016 (Capítulo 3);
- Eduardo Campos, Lucas Souza, Marcelo Maia. *Searching Crowd Knowledge to Recommend Solutions for API Usage Tasks*. Journal of Software: Evolution and Process (JSEP) (Volume: 28, Issue: 10, Pages: 863–892). Wiley, 2016 (Capítulo 4);

Outros artigos foram publicados durante o doutorado, mas não fazem parte desta tese:

- Eduardo Campos, Marcelo Maia. *Common Bug-fix Patterns: A Large-Scale Observational Study*. Proc. of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, Canada, pp. 1-10, 2017.
- Eduardo Campos, Marcelo Maia. *Mining Historical Information to Study Bug Fixes*. Proc. of 14th International Conference on Information Technology: New Generations (ITNG), Las Vegas, Nevada, USA, p. 535–543, 2018;

Outros artigos anteriores foram publicados previamente ao doutorado, mas contribuiram na fundamentação desta tese:

- Lucas Souza, Eduardo Campos, Marcelo Maia. *Ranking Crowd Knowledge to Assist Software Development*. Proc. of 22nd International Conference on Program Comprehension (ICPC), Hyderabad, India, p. 72–82, ACM, 2014. **ACM SIGSOFT Distinguished Paper Award** (Capítulo 4).
- Eduardo Campos, and Marcelo Maia. *Automatic Categorization of Questions from Q&A Sites*. Proc. of the 29th Annual ACM Symposium on Applied Computing (SAC), Gyeongju, Korea, p. 641-643, ACM, 2014 (Capítulo 4).
- Lucas Souza, Eduardo Campos, Marcelo Maia. *On the Extraction of Cookbooks for APIs from the Crowd Knowledge*. Proc. of the 28th Brazilian Symposium on Software Engineering (SBES), Maceió, Alagoas, Brasil, p. 21-30, IEEE Xplore, 2014.
- Eduardo Campos, Lucas Souza, Marcelo Maia. *Nuggets Miner: Assisting Developers by Harnessing the StackOverflow Crowd Knowledge and the GitHub Traceability*. Proc. of the Brazilian Conference on Software: Theory and Practice (CBSoft 2014) - Tool Session. 8pp. Maceió, Alagoas, Brasil. **Second-place Best Tool Award**.

Referências

ABDALKAREEM, R.; SHIHAB, E.; RILLING, J. What Do Developers Use the Crowd For? A Study Using Stack Overflow. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 2, p. 53–60, mar. 2017. ISSN 0740-7459. Disponível em: <https://doi.org/10.1109/MS.2017.31>.

ACAR, Y. et al. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In: **2016 IEEE Symposium on Security and Privacy (SP)**. [S.l.: s.n.], 2016. p. 289–305.

ACHARYA, M.; XIE, T. Mining API Error-Handling Specifications from Source Code. In: **Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009**. Berlin, Heidelberg: Springer-Verlag, 2009. (FASE '09), p. 370–384. ISBN 978-3-642-00592-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-00593-0_25>.

ACHARYA, M. et al. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In: **Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 25–34. ISBN 978-1-59593-811-4.

AMANI, S. et al. MUBench: A Benchmark for API-misuse Detectors. In: **Proceedings of the 13th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2016. (MSR '16), p. 464–467. ISBN 978-1-4503-4186-8.

AMANN, S. et al. A Systematic Evaluation of API-Misuse Detectors. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, 2018. ISSN 0098-5589.

ANDERSON, A. et al. Discovering Value from Community Activity on Focused Question Answering Sites: A Case Study of Stack Overflow. In: **Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: ACM, 2012. (KDD '12), p. 850–858. ISBN 978-1-4503-1462-6. Disponível em: <http://doi.acm.org/10.1145/2339530.2339665>.

ANTIN, J.; CHURCHILL, E. F. Badges in Social Media: A Social Psychological Perspective. In: **Proc. CHI**. [S.l.]: ACM, 2011.

Avizienis, Algirdas and Laprie, Jean-Claude and Randell, Brian and Landwehr, Carl. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Trans. Dependable Secur. Comput.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Disponível em: <http://dx.doi.org/10.1109/TDSC.2004.2>.

AYRES, J. et al. Sequential PAttern Mining Using a Bitmap Representation. In: **Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: ACM, 2002. (KDD '02), p. 429–435. ISBN 1-58113-567-X.

AZHAGUSUNDARI, B.; THANAMANI, A. S. Feature selection based on information gain. **International Journal of Innovative Technology and Exploring Engineering (IJITEE)**, p. 18–19, 2013.

BACCHELLI, A.; PONZANELLI, L.; LANZA, M. Harnessing Stack Overflow for the IDE. In: **Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2012. (RSSE '12), p. 26–30. ISBN 978-1-4673-1759-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2666719.2666725>.

BAJRACHARYA, S.; OSSHER, J.; LOPES, C. Searching API Usage Examples in Code Repositories with Sourcerer API Search. In: **Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation**. New York, NY, USA: ACM, 2010. (SUITE '10), p. 5–8. ISBN 978-1-60558-962-6. Disponível em: <http://doi.acm.org/10.1145/1809175.1809177>.

BALL, T.; NAIK, M.; RAJAMANI, S. K. From symptom to cause: Localizing errors in counterexample traces. In: **Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2003. (POPL '03), p. 97–105. ISBN 1-58113-628-5.

BAO, L. et al. Tracking and Analyzing Cross-Cutting Activities in Developers' Daily Work (N). In: **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 277–282.

BIALECKI, A.; MUIR, R.; INGERSOLL, G. Apache lucene 4. In: **SIGIR 2012 Workshop on Open Source Information Retrieval**. [S.l.: s.n.], 2012. p. 1–8.

BOSTANCI, B.; BOSTANCI, E. An Evaluation of Classification Algorithms Using Mc Nemar's Test. In: **Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2012)**. [S.l.]: Springer India, 2013. v. 201, p. 15–26. ISBN 978-81-322-1037-5.

BRANDT, J. et al. Example-centric Programming: Integrating Web Search into the Development Environment. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2010. p. 513–522. ISBN 978-1-60558-929-9. Disponível em: <http://doi.acm.org/10.1145/1753326.1753402>.

BREIMAN, L. Random forests. **Machine Learning**, Kluwer Academic Publishers, Hingham, MA, USA, v. 45, n. 1, p. 5–32, out. 2001. ISSN 0885-6125. Disponível em: <http://dx.doi.org/10.1023/A:1010933404324>.

BREIMAN, L. et al. Classification and regression trees. Wadsworth, 1984.

BRIN, S.; PAGE, L. The Anatomy of a Large-scale Hypertextual Web Search Engine. In: **Proceedings of the Seventh International Conference on World Wide Web 7**. [S.l.]: Elsevier Science Publishers B. V., 1998. (WWW7), p. 107–117.

CAMPOS, E. C.; MAIA, M. de A. Automatic Categorization of Questions from Q&A Sites. In: **Proceedings of the 29th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2014. (SAC '14), p. 641–643. ISBN 978-1-4503-2469-4. Disponível em: <http://doi.acm.org/10.1145/2554850.2555117>.

CAMPOS, E. C.; MONPERRUS, M.; MAIA, M. A. Searching Stack Overflow for API-usage-related Bug Fixes Using Snippet-based Queries. In: **Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering**. Riverton, NJ, USA: IBM Corp., 2016. (CASCON '16), p. 232–242. Disponível em: <http://dl.acm.org/citation.cfm?id=3049877.3049902>.

CAMPOS, E. C.; SOUZA, L. B. L. de; MAIA, M. d. A. Searching Crowd Knowledge to Recommend Solutions for API Usage Tasks. **Journal of Software: Evolution and Process**, 2016. ISSN 2047-7481. Wiley. Disponível em: <http://dx.doi.org/10.1002/smr.1800>.

CAPILUPPI, A.; SEREBRENIK, A.; SINGER, L. Assessing Technical Candidates on the Social Web. **IEEE Software**, v. 30, n. 1, p. 45–51, Jan 2013. ISSN 0740-7459.

CARZANIGA, A. et al. Automatic Workarounds for Web Applications. In: **Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2010. p. 237–246. ISBN 978-1-60558-791-2.

CAVUSOGLU, H.; LI, Z.; HUANG, K.-W. Can Gamification Motivate Voluntary Contributions?: The Case of StackOverflow Q&A Community. In: **Proceedings of the 18th ACM Conference Companion on Computer Supported Cooperative Work & Social Computing**. New York, NY, USA: ACM, 2015. (CSCW'15 Companion), p. 171–174. ISBN 978-1-4503-2946-0. Disponível em: <http://doi.acm.org/10.1145/2685553.2698999>.

CESSIE, S. le; HOUWELINGEN, J. van. Ridge Estimators in Logistic Regression. **Applied Statistics**, v. 41, n. 1, p. 191–201, 1992.

CHATTERJEE, S.; JUVEKAR, S.; SEN, K. SNIFF: A Search Engine for Java Using Free-Form Queries. In: **Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009**. Berlin, Heidelberg: Springer-Verlag, 2009. p. 385–400. ISBN 978-3-642-00592-3.

CHEN, F.; KIM, S. Crowd Debugging. In: **Proceedings of the 10th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. [S.l.]: ACM, 2015.

CLEVE, H.; ZELLER, A. Finding failure causes through automated testing. In: **4th International Workshop on Automated Debugging**. Munich, Germany: [s.n.], 2000.

COHEN, J. **Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit.** 1968.

COPELAND, T. **PMD Applied: An Easy-to-use Guide for Developers**. Centennial Books, 2005. (An easy-to-use guide for developers). ISBN 9780976221418. Disponível em: <https://books.google.com.br/books?id=Pj_8AAAACAAJ>.

CORDEIRO, J.; ANTUNES, B.; GOMES, P. Context-based Recommendation to Support Problem Solving in Software Development. In: **Proceedings of 3rd Int. Workshop on Recommendation Systems for Software Engineering**. [S.l.]: IEEE, 2012. p. 85–89.

CORDY, J. R.; ROY, C. K. The NiCad Clone Detector. In: **Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension**. Washington, DC, USA: IEEE Computer Society, 2011. (ICPC '11), p. 219–220. ISBN 978-0-7695-4398-7. Disponível em: <https://doi.org/10.1109/ICPC.2011.26>.

ČUBRANIĆ, DAVOR AND MURPHY, GAIL C. AND SINGER, JANICE AND BOOTH, KELLOGG S. Learning from Project History: A Case Study for Software Development. In: **Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work**. New York, NY, USA: ACM, 2004. p. 82–91.

ČUBRANIĆ, D. MURPHY, G. Hipikat: recommending pertinent software development artifacts. In: **Proceedings of the 25th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2003. (ICSE '03), p. 408–418. ISBN 0-7695-1877-X. Disponível em: <http://dl.acm.org/citation.cfm?id=776816.776866>.

CUNNINGHAM, P. **Overfitting and Diversity in Classification Ensembles based on Feature Selection**. [S.l.], 2000. 1-3 p.

CZERWINSKI, M.; HORVITZ, E.; WILHITE, S. A Diary Study of Task Switching and Interruptions. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2004. (CHI '04), p. 175–182. ISBN 1-58113-702-8. Disponível em: <http://doi.acm.org/10.1145/985692.985715>.

DAGENAIS, B.; ROBILLARD, M. P. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In: **Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2010. (FSE '10), p. 127–136. ISBN 978-1-60558-791-2. Disponível em: <http://doi.acm.org/10.1145/1882291.1882312>.

_____. Recovering Traceability Links Between an API and Its Learning Resources. In: **Proceedings of the 34th International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 47–57. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337230>.

DARWIN, I. F. **Java Cookbook**. Sebastopol, CA, USA: O'Reilly Media, 2004. ISBN: 978-0-596-00701-0.

DEKEL, U.; HERBSLEB, J. D. Improving API Documentation Usability with Knowledge Pushing. In: **Proceedings of the 31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society,

2009. (ICSE '09), p. 320–330. ISBN 978-1-4244-3453-4. Disponível em: <http://dx.doi.org/10.1109/ICSE.2009.5070532>.

DEMPSTER, A. P.; LAIRD, N. M.; RUBIN, D. B. Maximum likelihood from incomplete data via the EM algorithm. **Journal of the Royal Statistical Society: Series B**, v. 39, p. 1–38, 1977. Disponível em: <http://web.mit.edu/6.435/www/Dempster77.pdf>.

DETERDING, S. Gamification: Designing for Motivation. **Interactions**, ACM, New York, NY, USA, v. 19, n. 4, p. 14–17, jul. 2012. ISSN 1072-5520. Disponível em: <http://doi.acm.org/10.1145/2212877.2212883>.

DETERDING, S. et al. Gamification. using game-design elements in non-gaming contexts. In: **CHI '11 Extended Abstracts on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2011. (CHI EA '11), p. 2425–2428. ISBN 978-1-4503-0268-5. Disponível em: <http://doi.acm.org/10.1145/1979742.1979575>.

DIETTERICH, T. Overfitting and Undercomputing in Machine Learning. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 27, n. 3, p. 326–327, set. 1995. ISSN 0360-0300.

DUALA-EKOKO, E.; ROBILLARD, M. P. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In: **Proceedings of the 34th International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 266–276. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337255>.

ECKSTEIN, R.; LOY, M.; WOOD, D. **Java Swing**. Sebastopol, CA, USA: O'Reilly Media, 1998. ISBN: 1-56592-455-X.

EGELE, M. et al. An Empirical Study of Cryptographic Misuse in Android Applications. In: **Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security**. New York, NY, USA: ACM, 2013. (CCS '13), p. 73–84. ISBN 978-1-4503-2477-9.

ERK, K.; PADó, S. A Structured Vector Space Model for Word Meaning in Context. In: **Proceedings of the Conference on Empirical Methods in Natural Language Processing**. Stroudsburg, PA, USA: Association for Computational Linguistics, 2008. (EMNLP '08), p. 897–906.

ESTER, M. et al. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: **Proceedings of the Second International Conference on Knowledge Discovery and Data Mining**. AAAI Press, 1996. (KDD'96), p. 226–231. Disponível em: <http://dl.acm.org/citation.cfm?id=3001460.3001507>.

FAHL, S. et al. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In: **Proceedings of the 2012 ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2012. (CCS '12), p. 50–61. ISBN 978-1-4503-1651-4. Disponível em: <http://doi.acm.org/10.1145/2382196.2382205>.

FLANAGAN, C. et al. Extended Static Checking for Java. In: **Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2002. (PLDI '02), p. 234–245. ISBN 1-58113-463-0. Disponível em: <http://doi.acm.org/10.1145/512529.512558>.

FORMAN, G. An extensive empirical study of feature selection metrics for text classification. **Journal of Machine Learning Research**, 2003.

FOWKES, J.; SUTTON, C. Parameter-free Probabilistic API Mining Across GitHub. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2016. (FSE 2016), p. 254–265. ISBN 978-1-4503-4218-6.

GALLABA, K. et al. Refactoring Asynchrony in JavaScript. In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2017. p. 353–363.

GALLABA, K.; MESBAH, A.; BESCHASTNIKH, I. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In: **2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2015. p. 1–10. ISSN 1949-3770.

GANTER, B.; WILLE, R. **Formal Concept Analysis: Mathematical Foundations**. 1st. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997. ISBN 3540627715.

GENG, X. et al. Feature Selection for Ranking. In: **Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval**. New York, NY, USA: ACM, 2007. (SIGIR '07), p. 407–414. ISBN 978-1-59593-597-7. Disponível em: <http://doi.acm.org/10.1145/1277741.1277811>.

GEORGIEV, M. et al. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In: **Proceedings of the 2012 ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2012. (CCS '12), p. 38–49. ISBN 978-1-4503-1651-4. Disponível em: <http://doi.acm.org/10.1145/2382196.2382204>.

GLASSMAN, E. L. et al. Visualizing API Usage Examples at Scale. In: **Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2018. (CHI '18), p. 580:1–580:12. ISBN 978-1-4503-5620-6.

GOLDER, S. A.; HUBERMAN, B. A. Usage Patterns of Collaborative Tagging Systems. **Journal of Information Science**, Sage Publications, Inc., v. 32, n. 2, p. 198–208, abr. 2006. ISSN 0165-5515.

GROCE, A.; KROENING, D.; LERDA, F. Understanding counterexamples with explain. In: **In Computer-Aided Verification**. [S.l.: s.n.], 2004. p. 453–456.

GU, X.; ZHANG, H.; KIM, S. Deep Code Search. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 933–944. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180167>.

GUYON, I.; ELISSEEFF, A. An introduction to variable and feature selection. **Journal of Machine Learning Research**, 2003.

HALL, M. et al. The weka data mining software : An update. **SIGKDD Explorations**, p. 10–18, 2009.

HANAM, Q.; BRITO, F. S. d. M.; MESBAH, A. Discovering Bug Patterns in JavaScript. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2016. (FSE 2016), p. 144–156. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950308>.

HAPPEL, H.-J.; MAALEJ, W. Potentials and Challenges of Recommendation Systems for Software Development. In: **Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering**. New York, NY, USA: ACM, 2008. (RSSE '08), p. 11–15. ISBN 978-1-60558-228-3.

HARTMANN, B. et al. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2010. (CHI '10), p. 1019–1028. ISBN 978-1-60558-929-9.

HAYKIN, S. **Neural Networks: A Comprehensive Foundation**. PTR Upper Saddle River, NJ, USA: Prentice Hall, 1998.

HENNINGER, S. Supporting the Construction and Evolution of Component Repositories. In: **Proceedings of the 18th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 1996. p. 279–288. ISBN 0-8186-7246-3.

HENß, S.; MONPERRUS, M.; MEZINI, M. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In: **Proceedings of the 34th International Conference on Software Engineering (ICSE' 12)**. [S.l.: s.n.], 2012. p. 793–803. ISSN 0270-5257.

HILL, R.; RIDEOUT, J. Automatic Method Completion. In: **Proceedings. 19th International Conference on Automated Software Engineering**. [S.l.: s.n.], 2004. p. 228–235. ISSN 1938-4300.

HILYARD, J.; TEILHET, S. **C# 3.0 Cookbook, 3rd Edition**. Third. [S.l.]: O'Reilly, 2007. ISBN: 9780596516109.

HOLMES, R.; BEGEL, A. Deep Intellisense: A Tool for Rehydrating Evaporated Information. In: **Proceedings of the 2008 International Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2008. (MSR '08), p. 23–26. ISBN 978-1-60558-024-1. Disponível em: <http://doi.acm.org/10.1145/1370750.1370755>.

_____. Deep Intellisense: A tool for rehydrating evaporated information. In: **Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)**. [S.l.]: ACM, 2008. p. 23–26.

HOLMES, R. et al. The end-to-end use of source code examples: An exploratory study. In: **Proceedings of the IEEE International Conference on Software Maintenance**. [S.l.: s.n.], 2009.

HOLMES, R.; MURPHY, G. C. Using Structural Context to Recommend Source Code Examples. In: **Proceedings of the 27th International Conference on Software Engineering**. New York, NY, USA: ACM, 2005. p. 117–125. ISBN 1-58113-963-2.

HOLMES, R.; WALKER, R. J.; MURPHY, G. C. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 32, n. 12, p. 952–970, dez. 2006. ISSN 0098-5589.

_____. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. **IEEE Transactions on Software Engineering**, IEEE Press, p. 952–970, 2006.

HOVEMEYER, D.; PUGH, W. Finding Bugs is Easy. **ACM SIGPLAN Notices**, ACM, New York, NY, USA, v. 39, n. 12, p. 92–106, dez. 2004. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1052883.1052895>.

INOUE, K. et al. Component rank: relative significance rank for software component search. In: **25th International Conference on Software Engineering, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 14–24. ISSN 0270-5257.

IQBAL, S. T.; HORVITZ, E. Disruption and Recovery of Computing Tasks: Field Study, Analysis, and Directions. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2007. (CHI '07), p. 677–686. ISBN 978-1-59593-593-9. Disponível em: <http://doi.acm.org/10.1145/1240624.1240730>.

JENSEN, S. H.; MADSEN, M.; MØLLER, A. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 59–69. ISBN 978-1-4503-0443-6. Disponível em: <http://doi.acm.org/10.1145/2025113.2025125>.

JENSEN, S. H.; MØLLER, A.; THIEMANN, P. Type Analysis for JavaScript. In: **Proceedings of the 16th International Symposium on Static Analysis**. Berlin, Heidelberg: Springer-Verlag, 2009. (SAS '09), p. 238–255. ISBN 978-3-642-03236-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-03237-0_17>.

JIANG, L. et al. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 96–105. ISBN 0-7695-2828-7. Disponível em: <https://doi.org/10.1109/ICSE.2007.30>.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: **Proceedings of the 24th International Conference on Software Engineering**. New York, NY, USA: ACM, 2002. (ICSE '02), p. 467–477. ISBN 1-58113-472-X.

KAMIYA, T.; KUSUMOTO, S.; INOUE, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. **IEEE Transactions on Software Engineering**, v. 28, n. 7, p. 654–670, July 2002. ISSN 0098-5589.

KARAA, W.; GRIBâA, N. Information retrieval with Porter stemmer: A new version for English. In: **Advances in Computational Science, Engineering and Information Technology**. [S.l.]: Springer International Publishing, 2013, (Advances in Intelligent Systems and Computing, v. 225). p. 243–254. ISBN 978-3-319-00950-6.

KIM, K. et al. FaCoY: A Code-to-code Search Engine. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 946–957. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180187>.

KIM, S.; WHITEHEAD JR., E. J. How Long Did It Take to Fix Bugs? In: **Proceedings of the International Workshop on Mining Software Repositories (MSR '06)**. New York, NY, USA: ACM, 2006. p. 173–174. ISBN 1-59593-397-2.

KOHAVI, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In: **Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. (IJCAI'95), p. 1137–1143. ISBN 1-55860-363-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1643031.1643047>.

KOHAVI, R.; JOHN, G. H. Wrappers for Feature Subset Selection. **Artificial Intelligence**, Elsevier Science Publishers Ltd., Essex, UK, v. 97, n. 1-2, p. 273–324, dez. 1997. ISSN 0004-3702.

KRUEGER, C. W. Software Reuse. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/130844.130856>.

LATOZA, T. D.; VENOLIA, G.; DELINE, R. Maintaining Mental Models: A Study of Developer Work Habits. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: ACM, 2006. (ICSE '06), p. 492–501. ISBN 1-59593-375-1. Disponível em: <http://doi.acm.org/10.1145/1134285.1134355>.

LEGUNSEN, O. et al. How Good Are the Specs? A Study of the Bug-finding Effectiveness of Existing Java API Specifications. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2016. (ASE 2016), p. 602–613. ISBN 978-1-4503-3845-5. Disponível em: <http://doi.acm.org/10.1145/2970276.2970356>.

LEMOS, O. A. L. et al. Applying Test-driven Code Search to the Reuse of Auxiliary Functionality. In: **Proceedings of the 2009 ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2009. (SAC '09), p. 476–482. ISBN 978-1-60558-166-8. Disponível em: <http://doi.acm.org/10.1145/1529282.1529384>.

LEMPITSKY, V. et al. Random Forest Classification for Automatic Delineation of Myocardium in Real-Time 3D. In: **Functional Imaging and Modeling of the Heart**. [S.l.]: Springer Berlin Heidelberg, 2009. p. 447–456.

LI, Z.; ZHOU, Y. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In: **Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2005. (ESEC/FSE-13), p. 306–315. ISBN 1-59593-014-0. Disponível em: <http://doi.acm.org/10.1145/1081706.1081755>.

LIDDELL, D. Practical tests of 2x2 contingency tables. **Journal of the Royal Statistical Society**, v. 25, p. 295—-304, 1976.

LINARES-VáSQUEZ, M.; DIT, B.; POSHYVANYK, D. An Exploratory Analysis of Mobile Development Issues Using Stack Overflow. In: **Proceedings of the 10th Working Conference on Mining Software Repositories**. [S.l.]: IEEE Press, 2013. p. 93–96. ISBN 978-1-4673-2936-1.

LINARES-VáSQUEZ, M. et al. On Using Machine Learning to Automatically Classify Software Applications into Domain Categories. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 3, p. 582–618, jun. 2014. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-012-9230-z>.

LINDIG, C. **Mining Patterns and Violations using Concept Analysis**. [S.l.], 2007.

LINSTEAD, E. et al. Sourcerer: Mining and Searching Internet-scale Software Repositories. **Data Mining and Knowledge Discovery**, Kluwer Academic Publishers, Hingham, MA, USA, v. 18, n. 2, p. 300–336, abr. 2009. ISSN 1384-5810. Disponível em: <http://dx.doi.org/10.1007/s10618-008-0118-x>.

LIVSHITS, B.; ZIMMERMANN, T. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In: **Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.]: ACM, 2005. (ESEC/FSE-13), p. 296–305. ISBN 1-59593-014-0.

LV, F. et al. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In: **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 260–270.

Maalej, W.; Robillard, M. P. Patterns of Knowledge in API Reference Documentation. **IEEE Transactions on Software Engineering**, v. 39, n. 9, p. 1264–1282, Sep. 2013. ISSN 0098-5589.

MAMYKINA, L. et al. Design lessons from the fastest Q&A site in the west. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2011. p. 2857–2866.

MANDELIN, D. et al. Jungloid Mining: Helping to Navigate the API Jungle. In: **Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2005. (PLDI '05), p. 48–61. ISBN 1-59593-056-6. Disponível em: <http://doi.acm.org/10.1145/1065010.1065018>.

MANNING, C. D.; RAGHAVAN, P.; SCHüTZE, H. **Introduction to Information Retrieval**. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719 9780521865715.

MARTIE, L.; HOEK, A. v. d.; KWAK, T. Understanding the Impact of Support for Iteration on Code Search. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 774–785. ISBN 978-1-4503-5105-8.

MCDONNELL, T.; RAY, B.; KIM, M. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: **Proceedings of the 2013 IEEE International Conference on Software Maintenance**. Washington, DC, USA: IEEE Computer Society, 2013. (ICSM '13), p. 70–79. ISBN 978-0-7695-4981-1.

MCMILLAN, C. et al. Portfolio: Finding Relevant Functions and Their Usage. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, USA: ACM, 2011. (ICSE '11), p. 111–120. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985809>.

_____. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. **IEEE Transactions on Software Engineering**, v. 38, n. 5, p. 1069–1087, Sept 2012. ISSN 0098-5589.

MCNEMAR, Q. Note on the sampling error of the difference between correlated proportions or percentages. **Psychometrika**, v. 12, p. 153–157, 1947.

MEYER, A. N. et al. The Work Life of Developers: Activities, Switches and Perceived Productivity. **IEEE Transactions on Software Engineering**, v. 43, n. 12, p. 1178–1193, Dec 2017. ISSN 0098-5589.

MINELLI, R.; MOCCI, A.; LANZA, M. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In: **2015 IEEE 23rd International Conference on Program Comprehension**. [S.l.: s.n.], 2015. p. 25–35. ISSN 1092-8138.

MISHNE, A.; SHOHAM, S.; YAHAV, E. Typestate-based Semantic Code Search over Partial Programs. In: **Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2012. p. 997–1016. ISBN 978-1-4503-1561-6. Disponível em: <http://doi.acm.org/10.1145/2384616.2384689>.

MLADENIC, D.; GROBELNIK, M. Feature selection for unbalanced class distribution and naive bayes. **International Conference on Machine Learning (ICML)**, 1999.

MONPERRUS, M. et al. What should developers be aware of? an empirical study on the directives of api documentation. **Empirical Software Engineering**, v. 17, n. 6, p. 703–737, 2012. ISSN 1573-7616. Disponível em: <http://dx.doi.org/10.1007/s10664-011-9186-4>.

MONPERRUS, M.; MAIA, A. **Debugging with the Crowd: a Debug Recommendation System based on StackOverflow**. [S.l.], 2014. Disponível em: <http://hal.archives-ouvertes.fr/hal-00987395/PDF/article.pdf>.

MONPERRUS, M.; MEZINI, M. Detecting Missing Method Calls As Violations of the Majority Rule. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 22, n. 1, p. 7:1–7:25, mar. 2013. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/2430536.2430541>.

MORENO, L. et al. How Can I Use This Method? In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 880–890. ISBN 978-1-4799-1934-5.

MUJUMDAR, D. et al. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. In: **CHI '11 Extended Abstracts on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2011. (CHI EA '11), p. 1525–1530. ISBN 978-1-4503-0268-5.

NADI, S. et al. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 935–946. ISBN 978-1-4503-3900-1.

NASEHI, S. et al. What Makes a Good Code Example? A Study of Programming Q&A in Stack Overflow. In: **Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.: s.n.], 2012. p. 25–34.

_____. What Makes a Good Code Example? A Study of Programming Q&A in Stack Overflow. In: **Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)**. [S.l.: s.n.], 2012. p. 25–34.

NGUYEN, T. T. et al. Graph-based Mining of Multiple Object Usage Patterns. In: **Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2009. (ESEC/FSE '09), p. 383–392. ISBN 978-1-60558-001-2. Disponível em: <http://doi.acm.org/10.1145/1595696.1595767>.

_____. Recommending API Usages for Mobile Apps with Hidden Markov Model. In: **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 795–800.

OCARIZA, F. S. et al. A Study of Causes and Consequences of Client-Side JavaScript Bugs. **IEEE Transactions on Software Engineering**, IEEE Press, v. 43, n. 2, p. 128–144, 2017. ISSN 0098-5589.

OKUR, S.; DIG, D. How Do Developers Use Parallel Libraries? In: **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**. New York, NY, USA: ACM, 2012. (FSE '12), p. 54:1–54:11. ISBN 978-1-4503-1614-9. Disponível em: <http://doi.acm.org/10.1145/2393596.2393660>.

PAGE, L. et al. **The PageRank Citation Ranking: Bringing Order to the Web**. [S.l.], 1999. Previous number = SIDL-WP-1999-0120.

PAN, K.; KIM, S.; WHITEHEAD JR., E. J. Toward an Understanding of Bug Fix Patterns. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 14, n. 3, p. 286–315, 2009. ISSN 1382-3256.

PARNAS, D. L. On the Criteria to Be Used in Decomposing Systems into Modules. **Communications of the ACM**, ACM, New York, NY, USA, v. 15, n. 12, p. 1053–1058, dez. 1972. ISSN 0001-0782.

PARNIN, C.; ORSO, A. Are Automated Debugging Techniques Actually Helping Programmers? In: **Proceedings of the 2011 International Symposium on Software Testing and Analysis**. [S.l.]: ACM, 2011. p. 199–209. ISBN 978-1-4503-0562-4.

PARNIN, C.; RUGABER, S. Resumption strategies for interrupted programming tasks. **Software Quality Journal**, v. 19, n. 1, p. 5–34, Mar 2011. ISSN 1573-1367. Disponível em: <https://doi.org/10.1007/s11219-010-9104-9>.

PARNIN, C.; TREUDE, C. Measuring API Documentation on the Web. In: **Proceedings of the 2Nd International Workshop on Web 2.0 for Software Engineering**. New York, NY, USA: ACM, 2011. (Web2SE '11), p. 25–30. ISBN 978-1-4503-0595-2. Disponível em: <http://doi.acm.org/10.1145/1984701.1984706>.

PENG, C.-Y. J.; LEE, K. L.; INGERSOLL, G. M. An Introduction to Logistic Regression Analysis and Reporting. **The Journal of Educational Research**, v. 96, n. 1, p. 3–14, 2002. Disponível em: <http://dx.doi.org/10.1080/00220670209598786>.

PICCIONI, M.; FURIA, C. A.; MEYER, B. An empirical study of API usability. In: **2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013**. IEEE Computer Society, 2013. p. 5–14. ISBN 978-0-7695-5056-5. Disponível em: <https://doi.org/10.1109/ESEM.2013.14>.

POHAR, M.; BLAS, M.; TURK, S. Comparison of Logistic Regression and Linear Discriminant Analysis: A Simulation Study. **Metodološki Zvezki**, v. 1, n. 1, p. 143–144, 2004.

POLUKHIN, A. **Boost C++ Application Development Cookbook**. Birmingham: Packt Publ., 2013. ISBN: 1849514887 9781849514880.

PONZANELLI, L.; BACCHELLI, A.; LANZA, M. Leveraging Crowd Knowledge for Software Comprehension and Development. In: **Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE Computer Society, 2013. (CSMR '13), p. 57–66. ISBN 978-0-7695-4948-4.

_____. Leveraging Crowd Knowledge for Software Comprehension and Development. In: **Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering**. [S.l.]: IEEE Computer Society, 2013. (CSMR '13), p. 57–66. ISBN 978-0-7695-4948-4.

_____. Seahawk: Stack Overflow in the IDE. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA:

IEEE Press, 2013. p. 1295–1298. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486988>.

PONZANELLI, L. et al. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. p. 102–111. ISBN 978-1-4503-2863-0. Disponível em: <http://doi.acm.org/10.1145/2597073.2597077>.

_____. Prompter. **Empirical Software Engineering**, v. 21, n. 5, p. 2190–2231, Oct 2016. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-015-9397-1>.

_____. Improving Low Quality Stack Overflow Post Detection. In: **Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution**. Washington, DC, USA: IEEE Computer Society, 2014. (ICSME '14), p. 541–544. ISBN 978-1-4799-6146-7. Disponível em: <http://dx.doi.org/10.1109/ICSME.2014.90>.

PORTER, M. F. Readings in Information Retrieval. In: JONES, K. S.; WILLETT, P. (Ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. cap. An algorithm for suffix stripping, p. 313–316.

PRADEL, M.; SCHUH, P.; SEN, K. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [S.l.: s.n.], 2015. v. 1, p. 314–324. ISSN 0270-5257.

RABINER, L. R.; JUANG, B. H. An introduction to hidden Markov models. **IEEE ASSp Magazine**, 1986.

RAHMAN, M. M.; ROY, C. K.; LO, D. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2016. v. 1, p. 349–359.

RAMANATHAN, M. K.; GRAMA, A.; JAGANNATHAN, S. Path-Sensitive Inference of Function Precedence Protocols. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 240–250. ISBN 0-7695-2828-7. Disponível em: <http://dx.doi.org/10.1109/ICSE.2007.63>.

_____. Static Specification Inference Using Predicate Mining. In: **Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2007. (PLDI '07), p. 123–134. ISBN 978-1-59593-633-2. Disponível em: <http://doi.acm.org/10.1145/1250734.1250749>.

REISS, S. P. Semantics-based Code Search. In: **Proceedings of the 31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 243–253. ISBN 978-1-4244-3453-4. Disponível em: <http://dx.doi.org/10.1109/ICSE.2009.5070525>.

RENIERIS, M.; REISS, S. P. Fault Localization With Nearest Neighbor Queries. In: **18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada**. IEEE Computer Society, 2003. p. 30–39. ISBN 0-7695-2035-9. Disponível em: <http://dx.doi.org/10.1109/ASE.2003.1240292>.

RICHARDS, G. et al. An Analysis of the Dynamic Behavior of JavaScript Programs. In: **Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2010. (PLDI '10), p. 1–12. ISBN 978-1-4503-0019-3. Disponível em: <http://doi.acm.org/10.1145/1806596.1806598>.

RIGBY, P. C.; ROBILLARD, M. P. Discovering Essential Code Elements in Informal Documentation. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 832–841. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486897>.

ROBILLARD, M.; WALKER, R.; ZIMMERMANN, T. Recommendation Systems for Software Engineering. **IEEE Software**, v. 27, n. 4, p. 80–86, July 2010. ISSN 0740-7459.

_____. Recommendation Systems for Software Engineering. **IEEE Software**, v. 27, n. 4, p. 80–86, July 2010. ISSN 0740-7459.

ROBILLARD, M. P. What Makes APIs Hard to Learn? Answers from Developers. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 6, p. 27–34, nov. 2009. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/MS.2009.193>.

Robillard, M. P. What Makes APIs Hard to Learn? Answers from Developers. **IEEE Software**, v. 26, n. 6, p. 27–34, Nov 2009. ISSN 0740-7459.

Robillard, M. P. et al. Automated API Property Inference Techniques. **IEEE Transactions on Software Engineering**, v. 39, n. 5, p. 613–637, May 2013. ISSN 0098-5589.

ROBILLARD, M. P.; DELINE, R. A Field Study of API Learning Obstacles. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 16, n. 6, p. 703–732, dez. 2011.

ROBILLARD, M. P. et al. **An Introduction to Recommendation Systems in Software Engineering**. Springer-Verlag Berlin Heidelberg, 2014. (Recommendation Systems in Software Engineering). ISBN 978-3-642-45135-5. Disponível em: <http://www.springer.com/us/book/9783642451348>.

_____. On-demand Developer Documentation. In: **33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2017. p. 479–483.

Sadowski, Caitlin and Stolee, Kathryn T. and Elbaum, Sebastian. How Developers Search for Code: A Case Study. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2015. (ESEC/FSE 2015), p. 191–201. ISBN 978-1-4503-3675-8. Disponível em: <http://doi.acm.org/10.1145/2786805.2786855>.

SAHAVECHAPHAN, N.; CLAYPOOL, K. XSnippet: Mining For Sample Code. In: **Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications**. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 413–430. ISBN 1-59593-348-4. Disponível em: <http://doi.acm.org/10.1145/1167473.1167508>.

SAJNANI, H. et al. SourcererCC: Scaling Code Clone Detection to Big-code. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 1157–1168. ISBN 978-1-4503-3900-1. Disponível em: <http://doi.acm.org/10.1145/2884781.2884877>.

SALTON, G.; FOX, E. A.; WU, H. Extended Boolean Information Retrieval. **Commun. ACM**, ACM, New York, NY, USA, v. 26, n. 11, p. 1022–1036, nov. 1983. ISSN 0001-0782.

SALTON, G.; WONG, A.; YANG, C. S. A Vector Space Model for Automatic Indexing. **Commun. ACM**, ACM, New York, NY, USA, v. 18, n. 11, p. 613–620, nov. 1975. ISSN 0001-0782.

SAWADSKY, N.; MURPHY, G. C. Fishtail: From Task Context to Source Code Examples. In: **Proceedings of the 1st Workshop on Developing Tools As Plug-ins**. New York, NY, USA: ACM, 2011. p. 48–51.

SEHGAL, L.; MOHAN, N.; SANDHU, P. S. Quality prediction of function based software using decision tree approach. p. 43–44, September 2012.

SIM, S. E. et al. How Well Do Search Engines Support Code Retrieval on the Web? **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 21, n. 1, p. 4:1–4:25, dez. 2011. ISSN 1049-331X.

SOLINGEN, R. van; BERGHOUT, E.; LATUM, F. van. Interrupts: just a minute never is. **IEEE Software**, v. 15, n. 5, p. 97–103, Sep 1998. ISSN 0740-7459.

SOUZA, L. B. L. d.; CAMPOS, E. C.; MAIA, M. d. A. On the Extraction of Cookbooks for APIs from the Crowd Knowledge. In: **Brazilian Symposium on Software Engineering (SBES' 2014)**. [S.l.: s.n.], 2014. p. 21–30.

SOUZA, L. B. L. de; CAMPOS, E. C.; MAIA, M. d. A. Ranking Crowd Knowledge to Assist Software Development. In: **Proceedings of the 22Nd International Conference on Program Comprehension**. New York, NY, USA: ACM, 2014. p. 72–82. ISBN 978-1-4503-2879-1. Disponível em: <http://doi.acm.org/10.1145/2597008.2597146>.

STOLEE, K. T.; ELBAUM, S.; DOBOS, D. Solving the Search for Source Code. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 3, p. 26:1–26:45, jun. 2014. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/2581377>.

STOLEE, K. T.; ELBAUM, S.; DWYER, M. B. Code Search with Input/Output Queries. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 116, n. C, p. 35–48, jun. 2016. ISSN 0164-1212. Disponível em: <https://doi.org/10.1016/j.jss.2015.04.081>.

STYLOS, J.; MYERS, B. A. Mica: A Web-Search Tool for Finding API Components and Examples. In: **Visual Languages and Human-Centric Computing (VL/HCC'06)**. [S.l.: s.n.], 2006. p. 195–202. ISSN 1943-6092.

SU, D. C.-C. **Performance Analysis and Optimization on Lucene**. [S.l.], 2002.

SUBRAMANIAN, S.; HOLMES, R. Making Sense of Online Code Snippets. In: **Proceedings of the 10th Working Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 85–88. ISBN 978-1-4673-2936-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2487085.2487106>.

SUBRAMANIAN, S.; INOZEMTSEVA, L.; HOLMES, R. Live API Documentation. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. p. 643–652. ISBN 978-1-4503-2756-5.

SUROWIECKI, J. **The Wisdom of Crowds**. [S.l.]: Anchor, 2005. ISBN 0385721706.

SUSHINE, J.; HERBSLEB, J. D.; ALDRICH, J. Searching the State Space: A Qualitative Study of API Protocol Usability. In: **Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension**. Piscataway, NJ, USA: IEEE Press, 2015. (ICPC '15), p. 82–93. Disponível em: <http://dl.acm.org/citation.cfm?id=2820282.2820295>.

SVAJLENKO, J. et al. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In: **2014 IEEE International Conference on Software Maintenance and Evolution**. [S.l.: s.n.], 2014. p. 476–480. ISSN 1063-6773.

TAKUYA, W.; MASUHARA, H. A Spontaneous Code Recommendation Tool Based on Associative Search. In: **Proceedings of the 3rd International Workshop on Search-Driven Development**. [S.l.]: ACM, 2011. p. 17–20. ISBN: 978-1-4503-0597-6.

TERRAGNI, V.; LIU, Y.; CHEUNG, S.-C. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In: **roceedings of the 25th International Symposium on Software Testing and Analysis**. [S.l.]: ACM, 2016. (ISSTA 2016), p. 118–129. ISBN 978-1-4503-4390-9.

THUMMALAPENTA, S.; XIE, T. ParseWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In: **Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2007. (ASE '07), p. 204–213. ISBN 978-1-59593-882-4. Disponível em: <http://doi.acm.org/10.1145/1321631.1321663>.

_____. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In: **Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2008. (ASE '08), p. 327–336. ISBN 978-1-4244-2187-9. Disponível em: <http://dx.doi.org/10.1109/ASE.2008.43>.

_____. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In: **2009 IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2009. p. 283–294. ISSN 1938-4300.

_____. Mining Exception-handling Rules As Sequence Association Rules. In: **Proceedings of the 31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 496–506. ISBN 978-1-4244-3453-4. Disponível em: <http://dx.doi.org/10.1109/ICSE.2009.5070548>.

TREUDE, C.; BARZILAY, O.; STOREY, M.-A. How Do Programmers Ask and Answer Questions on the Web? (NIER Track). In: **Proceedings of the 33rd International Conference on Software Engineering**. [S.l.]: ACM, 2011. p. 804–807.

TREUDE, C.; ROBILLARD, M. P. Augmenting API Documentation with Insights from Stack Overflow. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016.

TUFANO, M. et al. When and Why Your Code Starts to Smell Bad. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 403–414. ISBN 978-1-4799-1934-5. Disponível em: <http://dl.acm.org/citation.cfm?id=2818754.2818805>.

UDDIN, G.; ROBILLARD, M. P. How API Documentation Fails. **IEEE Software**, v. 32, n. 4, p. 68–75, July 2015. ISSN 0740-7459.

UMARJI, M.; SIM, S. E.; LOPES, C. V. Archetypal Internet-Scale Source Code Searching. In: . Milan, Italy: [s.n.], 2008. p. 7.

VASILESCU, B. et al. How Social Q&A Sites Are Changing Knowledge Sharing in Open Source Software Communities. In: **Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work; Social Computing**. New York, NY, USA: ACM, 2014. (CSCW '14), p. 342–354. ISBN 978-1-4503-2540-0. Disponível em: <http://doi.acm.org/10.1145/2531602.2531659>.

VELASCO, R. **Apache Solr: For Starters**. USA: CreateSpace Independent Publishing Platform, 2016. ISBN 1540551903, 9781540551900.

Wang, J. et al. Mining succinct and high-coverage API usage patterns from source code. In: **2013 10th Working Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2013. p. 319–328. ISSN 2160-1860.

Wang, J.; Han, J. Bide: efficient mining of frequent closed sequences. In: **Proceedings. 20th International Conference on Data Engineering**. [S.l.: s.n.], 2004. p. 79–90. ISSN 1063-6382.

WANG, S. et al. EnTagRec: An enhanced tag recommendation system for software information sites. In: **Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.]: IEEE, 2014.

WASYLKOWSKI, A.; ZELLER, A. Mining Temporal Specifications from Object Usage. In: **Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 295–306. ISBN 978-0-7695-3891-4. Disponível em: <http://dx.doi.org/10.1109/ASE.2009.30>.

WASYLKOWSKI, A.; ZELLER, A.; LINDIG, C. Detecting Object Usage Anomalies. In: **Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 35–44. ISBN 978-1-59593-811-4. Disponível em: <http://doi.acm.org/10.1145/1287624.1287632>.

WATSON, C.; LI, F. W. B.; GODWIN, J. L. BlueFix: Using Crowd-sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In: **Proceedings of the 11th International Conference on Advances in Web-Based Learning**. Berlin, Heidelberg: Springer-Verlag, 2012. p. 228–239. ISBN 978-3-642-33641-6.

WEI, L.; LIU, Y.; CHEUNG, S.-C. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.]: ACM, 2016. p. 226–237. ISBN 978-1-4503-3845-5.

WILLIAMS, C. C.; HOLLINGSWORTH, J. K. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 31, n. 6, p. 466–480, jun. 2005. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2005.63>.

XIA, X. et al. What do developers search for on the web? **Empirical Software Engineering**, v. 22, n. 6, p. 3149–3185, Dec 2017. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-017-9514-4>.

_____. Tag Recommendation in Software Information Sites. In: **Proceedings of the 10th Working Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 287–296. ISBN 978-1-4673-2936-1.

XIE, T.; PEI, J. MAPO: Mining API Usages from Open Source Repositories. In: **Proceedings of the 2006 International Workshop on Mining Software Repositories**. New York, NY, USA: ACM, 2006. p. 54–57. ISBN 1-59593-397-2.

YANG, D.; HUSSAIN, A.; LOPES, C. V. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In: **Proceedings of the 13th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2016. (MSR '16), p. 391–402. ISBN 978-1-4503-4186-8.

YANG, Y.; PEDERSEN, J. O. A comparative study on feature selection in text categorization. **International Conference on Machine Learning (ICML)**, 1997.

YE, D.; XING, Z.; KAPRE, N. The Structure and Dynamics of Knowledge Network in Domain-specific Q&A Sites: A Case Study of Stack Overflow. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 1, p. 375–406, fev. 2017. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-016-9430-z>.

YE, Y.; FISCHER, G. Supporting Reuse by Delivering Task-relevant and Personalized Information. In: **Proceedings of the 24th International Conference on Software Engineering**. New York, NY, USA: ACM, 2002. p. 513–523. ISBN 1-58113-472-X.

_____. Reuse-conducive development environments. **Automated Software Engineering**, v. 12, n. 2, p. 199–235, Apr 2005. ISSN 1573-7535. Disponível em: <https://doi.org/10.1007/s10515-005-6206-x>.

ZELLER, A.; HILDEBRANDT, R. Simplifying and Isolating Failure-Inducing Input. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 2, p. 183–200, fev. 2002. ISSN 0098-5589.

ZHANG, T. et al. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 886–896. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180260>.

ZHANG, X.; GUPTA, N.; GUPTA, R. Pruning Dynamic Slices with Confidence. In: **Proceedings of the Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2006. p. 169–180. ISBN 1-59593-320-4.

ZHANG, X.; GUPTA, R.; ZHANG, Y. Precise Dynamic Slicing Algorithms. In: **Proceedings of the 25th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2003. p. 319–329. ISBN 0-7695-1877-X.

ZHONG, H.; SU, Z. Detecting API Documentation Errors. In: **Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications**. New York, NY, USA: ACM, 2013. (OOPSLA '13), p. 803–816. ISBN 978-1-4503-2374-1. Disponível em: <http://doi.acm.org/10.1145/2509136.2509523>.

ZHONG, H. et al. Mapo: Mining and recommending api usage patterns. In: DROSSOPOULOU, S. (Ed.). **ECOOP 2009 – Object-Oriented Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 318–343. ISBN 978-3-642-03013-0.

ZIMMERMANN, T. et al. Mining Version Histories to Guide Software Changes. In: **Proceedings of the 26th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2004. (ICSE '04), p. 563–572. ISBN 0-7695-2163-0. Disponível em: <http://dl.acm.org/citation.cfm?id=998675.999460>.