UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RECOMENDAÇÃO DE CONHECIMENTO DA MULTIDÃO PARA AUXÍLIO AO DESENVOLVIMENTO DE SOFTWARE

EDUARDO CUNHA CAMPOS

Uberlândia - Minas Gerais
2015

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EDUARDO CUNHA CAMPOS

RECOMENDAÇÃO DE CONHECIMENTO DA MULTIDÃO PARA AUXÍLIO AO DESENVOLVIMENTO DE SOFTWARE

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Prof. Dr. Marcelo de Almeida Maia

Co-orientador:
Prof. Dr. Martin Monperrus

Uberlândia, Minas Gerais
2015

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

C198r
2014

Campos, Eduardo Cunha, 1988-

Recomendação de conhecimento da multidão para auxílio ao desenvolvimento de software / Eduardo Cunha Campos. - 2014.

100 f. : il.

Orientador: Marcelo de Almeida Maia.

Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.

Inclui bibliografia.

1. Computação - Teses. 2. Software - Desenvolvimento - Teses. I. Maia, Marcelo de Almeida. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada "**Recomendação de conhecimento da multidão para auxílio ao desenvolvimento de software**" por **Eduardo Cunha Campos** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 16 de Janeiro de 2015

Orientador: _____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Co-orientador: _____
Prof. Dr. Martin Monperrus
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Stéphane Julia
Universidade Federal de Uberlândia

_____
Prof. Dr. Fernando Marques Figueira Filho
Universidade Federal do Rio Grande do Norte

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Janeiro de 2015

Autor:	**Eduardo Cunha Campos**
Título:	**Recomendação de conhecimento da multidão para auxílio ao desenvolvimento de software**
Faculdade:	**Faculdade de Ciência da Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Aos meus pais Olavo e Rose e as minhas irmãs Tatiana e Izabella. A minha namorada Lycea.

Agradecimentos

Agradeço...

A Deus, por minha vida.

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.

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.

Ao professor Rivalino Matias Júnior, pelo profissionalismo e pelos favores prestados fundamentais para a realização deste trabalho.

Aos meus amigos pelo incentivo e carinho:
Antônia Maria Maciel,
Batuira Filho,
Blayton Portela,
Carlos Eduardo Soares Sabino,
Fellipe Martins Lamoglia,
Flávio Marques,
Henrique De Villa Alves,
José Hudson Viana,
Lucas Batista Leite de Souza,
Luis Felipe Nogueira,
Raphael Ferreira Vieira,
Ricardo Luiz de Sousa Campos,
Roberto César Rodrigues.

A CAPES, CNPq pelo apoio financeiro.

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

"Nada é tão nosso quanto os nossos sonhos."
(Friedrich Nietzsche)

Resumo

O desenvolvimento de *software* dos dias atuais é inseparável do uso de Interfaces de Programação de Aplicações (APIs). Vários estudos mostraram que os desenvolvedores enfrentam problemas quando lidam com APIs não-familiares. São raros os casos em que a documentação e exemplos providos para um grande *framework* ou biblioteca são suficientes para o desenvolvedor utilizar a API efetivamente. Frequentemente, os desenvolvedores ficam perdidos quando tentam utilizar uma API e inseguros em como obter um progresso em uma tarefa de programação. Um comportamento comum destes desenvolvedores é postar perguntas em serviços de mídia social e obter respostas de outros desenvolvedores que pertencem a diferentes projetos.

Para ajudar os desenvolvedores a encontrar seu caminho, uma alternativa amplamente conhecida é o *Stack Overflow* (SOF), que é um *site* de pergunta e resposta (Q&A) que usa mídia social para facilitar a troca de conhecimento entre os desenvolvedores.

Apesar da sua utilidade, o conhecimento provido pelos serviços Q&A não pode ser aproveitado diretamente dentro do Ambiente de Desenvolvimento Integrado (IDE), no sentido de que os desenvolvedores devem mudar para o navegador *Web* para acessar estes serviços, interrompendo assim, o fluxo de programação e reduzindo o seu foco na tarefa de programação atual.

Outra atividade comum no cotidiano do desenvolvedor é a depuração. Esta atividade consiste em entender por que um pedaço de código não se comporta como esperado. Alguns *bugs* estão profundamente enraizados na lógica de domínio, mas outros são independentes da especificidade da aplicação que está sendo depurada. Esta última classe de *bugs* pode ser definida como *"bugs da multidão"*: *bugs* que causam uma saída ou comportamentos inesperados e incorretos resultantes do uso comum e intuitivo de uma API. Para esta classe de *bugs*, é provável que o mesmo já ocorreu diversas vezes em diferentes aplicações e por isso, a multidão já identificou o *bug* bem como sua correção.

Este trabalho propõe um sistema de recomendação na forma de um plug-in para Eclipse IDE que tem como objetivos principais: 1) Recomendar pares Q&A (i.e., um par Q&A é composto por uma pergunta e uma resposta para esta pergunta) da categoria *How-to-do* para auxiliar os desenvolvedores no processo de aprendizagem da API; 2) Recomendar correções de *"bugs da multidão"* para auxiliar os desenvolvedores durante as tarefas de depuração. Estas recomendações utilizam o *site* SOF como fonte de informação.

Foi conduzido um experimento considerando tarefas de programação de três diferentes APIs (*Swing*, *Boost* e *LINQ*) amplamente utilizados pela comunidade de desenvolvimento de *software* para avaliar a estratégia de recomendação de pares Q&A proposta. Os resultados são promissores: para 77,14% das 35 tarefas de programação avaliadas, pelo menos um par Q&A recomendado provou ser útil na resolução da tarefa de programação alvo. Foi realizado também um experimento com *"bugs da multidão"* pertencentes às linguagens de programação Java e JavaScript. Os resultados também são promissores: para 73,33% dos *"bugs da multidão"* Java, o sistema encontrou a correção no Top-15 e para 46,66% dos *"bugs da multidão"* JavaScript, o sistema encontrou a correção no Top-15.

Palavras chave: serviços q&a, "conhecimento da multidão", "bugs da multidão".

Abstract

Modern-day software development is inseparable from the use of the Application Programming Interfaces (APIs). Several studies have shown that developers face problems when dealing with unfamiliar APIs. It is seldom the case that the documentation and examples provided with a large framework or library are sufficient for a developer to use their API effectively. Frequently, developers become lost when trying to use an API, unsure of how to make a progress on a programming task. A common behavior of developers is to post questions on social media services and receive answers from other programmers that belong to different projects.

To help developers find their way, a widely-know alternative is Stack Overflow (SOF), which is a Question and Answer (Q&A) website which uses social media to facilitate knowledge exchange between programmers.

Despite its usefulness, the knowledge provided by Q&A services cannot be directly leveraged from within an Integrated Development Environment (IDE), in the sense, that developers must toggle to the Web browser to access those services, thus interrupting the programming flow and lowering their focus on the current task.

Another common activity in daily developer is Debugging. This activity consists of understanding why a piece of code does not behave as expected. Some bugs are deeply rooted in the domain logic but others are independent of the specificity of the application being debugged. This latter class of bugs are "crowd bugs": bugs that cause an unexpected and incorrect output or behavior resulting from a common and intuitive usage of an API. For this class of bugs, it is likely that the crowd bug has already occurred several times in different applications and the crowd has already identified the bug and its fix.

This work proposes a recommendation system in the form of a plugin for the Eclipse IDE that has as main objectives: 1) Recommend Q&A pairs (i.e., a Q&A pair is composed by a question and a answer for that question) of *How-to-do* category to assist developers in the API Learning Process; 2) Recommend "crowd bugs" fixes to help developers during debugging tasks. These recommendations use SOF website as an information source.

We conducted an experiment considering programming problems on three different topics (Swing, Boost and LINQ) widely used by the software development community to evaluate our recommendation strategy of Q&A pairs. The results are promising: for 77.14% of the 35 programming problems assessed, at least one recommended Q&A pair proved to be useful in the resolution of the target programming problem. It was also conducted an experiment with crowd bugs belonging to Java and JavaScript programming languages. The results are also promising: for 73.33% of Java crowd bugs, the system has found the fix in the Top-15 and for 46.66% of JavaScript crowd bugs, the system has found the fix in the Top-15.

Keywords: q&a services, "crowd knowledge", "crowd bugs".

Sumário

Lista de Figuras

Capítulo 1

Introdução

Desenvolvedores de *software* são continuamente introduzidos a novas tecnologias, componentes e ideias [42]. *Application Programming Interfaces* (APIs) são expostas para os desenvolvedores de forma a permitir o reuso de bibliotecas de *software* [34]. Tradicionalmente, muitos tipos de documentação de *software*, como documentação de API, são geradas por poucas pessoas e possuem um público-alvo muito maior. A documentação resultante, quando existe, geralmente possui baixa qualidade e carece de exemplos e explicações [37]. Devido ao fato dos desenvolvedores apresentarem dificuldades na utilização de algumas funcionalidades da biblioteca, eles frequentemente procuram por informação na *Web* que irá ajudá-los a resolver seus problemas de desenvolvimento de *software* [2].

Na última década, uma nova cultura e filosofia emergiram e estão mudando as características do desenvolvimento de *software*. Esta mudança é o resultado de uma ampla estrutura acessível da tecnologia de mídia social (wikis, blogs, sites de pergunta e resposta, fóruns.) [37]. Similar à maneira que o desenvolvimento *open source* mudou o processo tradicional de desenvolvimento de *software* [33], estas novas formas de colaboração e contribuição tem o potencial de redefinir como os desenvolvedores aprendem, preservam e compartilham o conhecimento sobre desenvolvimento de *software*.

Um importante exemplo de mídia social é o *Stack Overflow* (SOF), que é um serviço de pergunta e resposta orientado para suportar a colaboração entre os desenvolvedores de forma a ajudá-los a resolver seus problemas relacionados com desenvolvimento de *software*. No SOF, os desenvolvedores realizam perguntas relacionadas a um tópico de programação e outros membros do *site* podem prover respostas para ajudá-los. O *design* deste serviço estimula uma comunidade de desenvolvedores a praticar atividades de Engenharia de *Software*, que contribuem para a documentação composta por trechos de código úteis e de alta qualidade [2]. O conjunto de informações disponíveis neste serviço é denominado "conhecimento da multidão" e muitas vezes se torna um substituto para a documentação oficial do *software* [48].

Mamykina et al. conduziram um estudo estatístico com todo o *corpus* do SOF para descobrir o que está por trás do sucesso imediato do *site*. Suas descobertas mostraram que

a maioria das perguntas recebem uma ou mais respostas (mais de 90% muito rapidamente - com um tempo médio de resposta de 11 minutos) [30]. Em [49], Treude et al. apontaram que o SOF é particularmente efetivo para revisões de código, para questões conceituais e para novatos.

O SOF possui no seu *website*, um motor de busca que permite aos usuários realizar consultas informando um conteúdo textual (e.g., "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, 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* para o usuário que, apesar de ser relevante com a consulta, possui 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.

Ao realizar consultas baseadas em trechos de código utilizando o motor de busca do SOF, os desenvolvedores também enfrentam problemas, uma vez que o índice nativo do *site* não foi designado para pesquisas baseadas em código. Desta forma, é comum os desenvolvedores não encontrarem *threads* de discussão para consultas baseadas em código. Portanto, é necessário uma abordagem para melhorar este tipo de consulta.

Um comportamento comum dos desenvolvedores é postar trechos de código-fonte que contém *bugs* em sites de pergunta-resposta (Q&A), com o intuito de obter uma revisão/correção para aquele determinado *bug*. Existe uma classe de *bugs* que está relacionada a um desentendimento comum sobre uma determinada funcionalidade da API. Estes *bugs* apresentam uma característica em comum: ocorrem repetidas vezes e são independentes do domínio da aplicação com *bugs*. Por exemplo, existe uma abundância de desenvolvedores que obtiveram o valor "0" ao invés do valor "8" para a função *parseInt*("08") do JavaScript. Este trecho de código não se comporta como o esperado. Esta classe de *bugs* foi denominada como *"crowd bugs"* por Monperrus et al. [35].

Para auxiliar o desenvolvedor durante as tarefas de desenvolvimento, a abordagem do presente trabalho tem como objetivos principais: 1) apresentar uma estratégia de recomendação que faz uso da informação disponível no SOF para sugerir pares Q&A (i.e., um par Q&A é composto por uma pergunta e uma resposta para esta pergunta) que podem ser úteis para a tarefa de programação que o desenvolvedor precisa solucionar; 2) Recomendar correções de *"crowd bugs"* [35] presentes no SOF para ajudar os desenvolvedores durante as tarefas de depuração.

A proposta é original em relação a outros trabalhos correlatos (e.g. Ponzanelli et al. [40], Cordeiro et al. [11], ČubraniĆ et al. [50], Takuya et al. [47]) pois classifica os pares Q&A de acordo com as preocupações que o usuário tem ao realizar uma pergunta

no *site* SOF. Ao realizar uma pergunta no site, o usuário pode estar procurando por livros, ferramentas ou tutoriais sobre uma determinada tecnologia. Uma outra possibilidade é que o usuário pode estar querendo uma explicação conceitual sobre alguma funcionalidade de uma tecnologia ou API. Ou ainda, o mesmo pode estar procurando por uma solução para o problema de desenvolvimento que precisa resolver.

A presente abordagem propõe um classificador automático que é capaz de distinguir entre 3 classes de categorias dos pares Q&A: *How-to-do*, *Conceptual* e *Seeking-Something*. Para recomendar pares Q&A para um desenvolvedor, o sistema de recomendação proposto por este trabalho consulta um índice de documentos contendo a informação destes pares Q&A (e.g., título, corpo da pergunta, corpo da resposta, etc.).

Para a composição deste índice de documentos, foi considerado apenas pares Q&A da categoria *How-to-do*, uma vez que esses pares possuem o formato de uma receita de um *Cookbook* (título da receita, cenário da tarefa, solução para a tarefa). A idéia é que pares Q&A da categoria *How-to-do* possuem um caráter mais "prático" do que "teórico" (e.g., pares das categorias *Conceptual* ou *Seeking-Something*). Além disso, são mais adequados para ajudar um desenvolvedor nas tarefas de desenvolvimento, visto que muitas respostas destes pares contém trechos de código-fonte de alta qualidade. Esta qualidade tem sido verificada pelos usuários da comunidade, que votam positivamente na resposta do *post* que possui trechos de código-fonte úteis na solução de um problema específico [46]. Portanto, os mesmos podem ser reutilizados por outros desenvolvedores durante as tarefas de desenvolvimento. A Figura 1.1 mostra a página inicial do SOF.

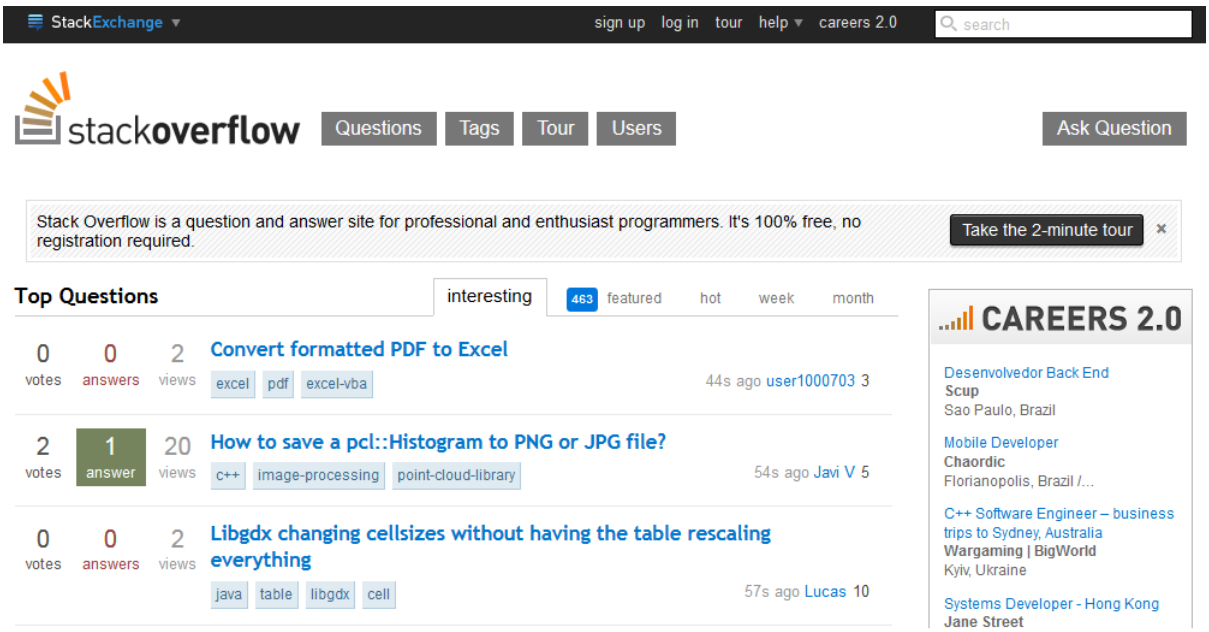

Figura 1.1: Página inicial do SOF. Acesso feito no dia 13/02/2014 11:00.

1.1 Contribuições

O presente trabalho possui as seguintes contribuições principais:

- Propor um sistema de recomendação na forma de um *plugin* para Eclipse IDE denominado *Nuggets Miner* [6] que utiliza o "conhecimento da multidão" disponível no SOF e possibilita ao desenvolvedor:

    1. Interagir diretamente com este conhecimento através da importação de trechos de código-fonte (e.g., exemplos de uso de uma API) para o editor de texto do programa via *drag and drop*;
    2. Aprender como resolver uma determinada tarefa de programação utilizando uma API particular;
    3. Obter correções de *"crowd bugs"* [35] através de *posts* do SOF durante as tarefas de depuração. Estes *posts* são recomendados quando o desenvolvedor formula consultas baseadas em trechos de código que supostamente possuem *"crowd bugs"*.

1.2 Organização da Dissertação

Esta dissertação está dividida em quatro capítulos.

O Capítulo 2 descreve os fundamentos teóricos do presente trabalho.

O Capítulo 3 descreve uma abordagem que faz uso do "conhecimento da multidão" disponível no SOF para recomendar informação que pode auxiliar os desenvolvedores em suas tarefas de desenvolvimento relacionadas a uma determinada API.

O Capítulo 4 descreve uma abordagem que faz uso do "conhecimento da multidão" disponível no SOF para recomendar correções de "bugs da multidão" para auxiliar os desenvolvedores durante as atividades de depuração do código-fonte.

Finalmente, no Capítulo 5 serão apresentadas as conclusões e as perspectivas para trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

A estratégia de recomendação de pares Q&A 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 ajudar os desenvolvedores durante as tarefas de programação com uma determinada API. Porém, como classificar automaticamente milhares de pares Q&A do SOF 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 (e.g., presença de determinadas palavras-chave) que possibilitam classificar um par Q&A do SOF dentro de um número de categorias ou classes. Para isso, existem vários algoritmos de classificação amplamente conhecidos nas Áreas de Aprendizado de Máquina e Reconhecimento de Padrões que viabilizam este tipo de classificação.

Estes 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 (e.g., *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 (e.g., redução de atributos do classificador, como será realizado o treinamento e teste do classificador). Finalmente, a Seção 2.3 apresenta o *software WEKA*, que foi utilizado no trabalho para estimar a taxa de acerto dos classi-

ficadores.

2.1 Métodos de Aprendizado de Máquina para Classificação

Esta seção apresenta os algoritmos de classificação conhecidos na área de Reconhecimento de Padrões 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 do 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, flexível e de fácil utilização. No LR, não são feitas suposições sobre a distribuição das variáveis explicativas [38].

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 [27]. O nome da classe que implementa este algoritmo no *WEKA* [20] é a *Logistic*.

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 Teorema de *Bayes*. As probabilidades *a priori* são estimadas com os dados de treinamento [28].

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 [28].

2.1.3 *Multilayer Perceptron*

Uma rede *Multilayer Perceptron* (MLP) consiste de 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 *Single-layer Perceptron* [21].

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*) [21].

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. SVM são classificadores binários, mas podem ser utilizados para classificação multi-classe [28].

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 [28].

2.1.5 *J4.8 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 [28].

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 [44].

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 [5].

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 (e.g., 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 [4].

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. O KNN também é conhecido no *WEKA* [20] como IBK [28].

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 Aprendizagem 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 [12]. A probabilidade de ocorrer *Overfitting* aumenta à medida que a dimensão do espaço de características aumenta [1].

2.2.2 *Principal Component Analysis*

Principal Component Analysis (PCA) é uma das técnicas mais bem sucedidas que têm sido utilizada em reconhecimento de imagem e compressão. PCA é um método estatístico cujo propósito é reduzir a grande dimensionalidade do espaço de dados (variáveis observadas) para a menor dimensionalidade instrínseca do espaço de características (variáveis independentes). Este é o caso quando existe uma forte correlação entre as variáveis observadas [21].

Os trabalhos que PCA pode fazer são previsão, remoção de redundância, extração de características, compressão de dados, entre outros [21].

2.2.3 *Cross-validation*

Cross-validation é um método estatístico utilizado para estimar a acurácia dos clas-

sificadores em um ambiente de aprendizagem supervisionado. A forma básica do *Cross-validation* é *k-fold cross-validation* [25].

Na *k-fold cross-validation*, os dados são primeiramente particionados em k *folds* de mesmo tamanho. Subsequentemente, k iterações de treinamento e validação são realizados de forma que dentro de cada iteração, um *fold* diferente é escolhido para validação enquanto que os k - 1 folds restantes são utilizados para treinamento [25].

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 [25].

A principal vantagem de utilizar *k-fold cross-validation* é a estimativa precisa de desempenho. 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) [25].

2.2.4 *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 [1].

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 perto quanto possível da distribuição original obtida utilizando todos os atributos [1].

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

- Seu uso pode ajudar a aumentar a acurácia em muitos problemas de aprendizagem de máquina [18];
- Seu uso pode ajudar a melhorar a eficiência do treinamento [18];
- É um meio poderoso para evitar o *Overfitting* [1].

Basicamente, os métodos de *Feature Selection* são divididos em três categorias [19]: *filter*, *wrapper* [26] e *embedded method* [5].

Um método do tipo *filter* calcula uma pontuação para cada característica e, em seguida, seleciona as características de acordo com a pontuação delas [32]. Yang et al. [51] and Forman [17] 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 *Software WEKA*

O *WEKA* é um *software* de mineração de dados que foi desenvolvido utilizando a linguagem JAVA pela Universidade de *Waikato* na Nova Zelândia. Possui uma coleção de algoritmos de aprendizagem de máquina para tarefas de mineração de dados. Ele implementa também algoritmos para pré-processamento de dados, classificação, regressão, *clustering* e regras de associação, além de incluir ferramentas de visualização.

O *software* encontra-se licenciado ao abrigo da *General Public License* (GPL) sendo portanto possível estudar e alterar o respectivo código fonte [20].

O arquivo de dados normalmente utilizado pelo *WEKA* é o formato de arquivo ARFF (*Attribute-Relation File Format*), que consiste de *tags* especiais para indicar diferentes elementos no arquivo de dados (e.g. nomes de atributos, tipos de atributos, valores de atributos e os dados).

Capítulo 3

Recomendação Baseada em Termos para Auxílio à Compreensão de APIs

3.1 Introdução

Neste capítulo, será apresentada uma estratégia de recomendação que faz uso da informação disponível no SOF para sugerir pares de pergunta-resposta (Q&A) que podem ser úteis para a tarefa de programação que o desenvolvedor precisa resolver. Esta abordagem visa recomendar pares Q&A considerando quatro critérios. O primeiro refere-se à similaridade textual que o par Q&A possui com a tarefa que o desenvolvedor possui em mãos. A razão por trás deste critério é que outros desenvolvedores podem ter tido dúvidas semelhantes no passado e postaram as perguntas no SOF, então as respostas para estas perguntas passadas podem ser reusadas. O segundo critério considera o *score* das perguntas e respostas do SOF para tentar recomendar somente pares Q&A que foram bem avaliados pela multidão. O terceiro critério está relacionado ao mecanismo de filtragem dos *posts*: serão apenas recomendadas as soluções para tarefas de desenvolvimento que podem ser modeladas como *posts "How-to-do"* que serão filtrados por um algoritmo de classificação. O quarto critério está relacionado com a utilização de *tags* associadas às perguntas do SOF para filtrar apenas os *posts* do SOF associados a uma determinada API, uma vez que uma *tag* representa o nome da API (e.g., "swing", "boost", "linq", etc.) que será usada para realizar uma determinada tarefa de programação. O resultado do processo de recomendação é um conjunto de pares Q&A.

Para avaliar a estratégia de recomendação, faz-se necessário o desenho de um estudo experimental. O desenho do experimento foi baseado na escolha de tarefas de programação que pudessem ser respondidas com os pares de pergunta-resposta recomendados pelo sistema de recomendação proposto. As tarefas de programação utilizadas nos experimentos foram extraídas aleatoriamente de livros de receitas (*cookbooks*) sobre três APIs amplamente utilizadas pela comunidade de desenvolvimento de *software*: *Swing*, *Boost* e

LINQ.

O restante deste capítulo está organizado da seguinte maneira. A Seção 3.2 apresenta um classificador *Logistic Regression* utilizado neste estudo para classificar os pares Q&A em categorias de domínio. A Seção 3.4 apresenta a avaliação do ranqueamento dos pares Q&A recomendados pela abordagem proposta. Na Seção 3.5 são apresentados os resultados que são discutidos na Seção 3.6. Na Seção 3.7, são apresentados os trabalhos relacionados. Na Seção 3.9 são feitas as conclusões.

3.2 Classificação dos Pares Q&A

No SOF, os usuários perguntam muitos tipos de perguntas diferentes. Segundo Nasehi et al. [36], "os tipos de perguntas so SOF podem ser descritos com base em duas dimensões diferentes. A primeira dimensão lida com o tópico da pergunta: ela mostra a principal tecnologia ou conceito que a pergunta gira em torno e normalmente pode ser identificada a partir das *tags* da pergunta que o questionador pode adicionar na pergunta para ficar mais claro para outras pessoas (e.g., como os potenciais respondedores) a descobrir sobre qual assunto é a pergunta". Portanto, se o objetivo é recomendar pares Q&A para o tópico *Swing*, será apenas considerado na abordagem os pares Q&A pertencentes as *threads* de discussão em que a pergunta possui a *tag "swing"* entre suas *tags* (uma pergunta no SOF pode possuir até 5 *tags*). Ainda de acordo com Nasehi et al., "as perguntas do SOF podem também ser classificadas em uma segunda dimensão que refere-se às principais preocupações dos usuários e o que eles pretendem solucionar". Nesta dimensão, foram consideradas 5 categorias de pares Q&A:

- *How-to-do*: Provê um cenário e pergunta sobre como implementá-lo informando uma dada tecnologia ou API (e.g., como ordenar um vetor utilizando a linguagem Java). Pode ocorrer também quando o interrogador quer realizar uma tarefa mas não sabe como fazê-la (e.g., como instalar o *plugin* do GWT no Eclipse Juno). O interrogador espera obter receitas de qualidade para sua tarefa. O seguinte *post* do SOF é um exemplo desta categoria: [1];
- *Conceptual*: Questões/Dúvidas conceituais sobre determinado tópico (e.g., definição de conceitos, quais são as melhores práticas para uma dada tecnologia, diferença entre uma tecnologia e outra, etc.). O interrogador está querendo uma explicação sobre um determinado assunto ou justificação sobre determinado comportamento. O seguinte *post* do SOF é um exemplo desta categoria: [2];
- *Seeking-something*: O interrogador está procurando por algo mais "objetivo" (e.g., livro, tutorial, ferramenta, *framework*, biblioteca, aplicação) ou mais "subjetivo"

[1] http://stackoverflow.com/questions/527719/how-to-add-hyperlink-in-jlabel
[2] http://stackoverflow.com/questions/408820/what-is-the-difference-between-swing-and-awt

(e.g., conselho, opinião, sugestão, idéias, recomendação). O seguinte *post* do SOF é um exemplo desta categoria: [3];

- *Debug-Corrective*: Questões que lidam com problemas no código em desenvolvimento, como erros em tempo de execução, notificações ou comportamento inesperado do programa. O interrogador geralmente procura por correção no seu código. O seguinte *post* do SOF é um exemplo desta categoria: [4];
- *Miscellaneous*: Ocorre quando o interrogador tem vários interesses diferentes. Logo, ele faz várias perguntas. Isto geralmente conduz a uma mistura entre as outras categorias (e.g., o interrogador pode estar procurando por um livro e também por uma receita para a sua tarefa). O seguinte *post* do SOF é um exemplo desta categoria: [5].

No presente trabalho, foi decidido classificar o par Q&A todo ao invés de classificar somente a pergunta do par, uma vez que foi observado que, em alguns casos, a resposta do par continha informação relevante para ajudar na tomada de decisão sobre a categoria do par Q&A (e.g., diferenciar entre pares Q&A das categorias *How-to-do* e *Debug-Corrective*). Estas duas categorias são diferentes pois um par da categoria *Debug-Corrective* lida com correções de *bugs* no trecho de código, enquanto que um par da categoria *How-to-do* ensina como realizar uma tarefa de programação usando os elementos da API (i.e., classes e métodos) e não como corrigir um defeito em um trecho de código postado pelo usuário.

A categoria *How-to-do* é muito próxima do cenário em que um desenvolvedor possui uma tarefa de programação em mãos e precisa solucioná-la. Por esta razão, na abordagem proposta, foram considerados apenas os pares Q&A que foram classificados como *How-to-do*. Os pares Q&A desta categoria se assemelham às receitas de *cookbooks* (i.e., possuem o seguinte formato: título, tarefa de desenvolvimento e solução (passo-a-passo) para a tarefa).

Como o interesse é apenas nos pares Q&A da categoria *How-to-do* e a tarefa de categorização manual desses pares Q&A é tediosa e demorada, foi necessário realizar os seguintes passos para propiciar uma categorização automática:

- Construir um *dataset* para treinar/testar o classificador. A Subseção 3.2.1 detalha o processo utilizado;
- Definir os atributos do classificador e realizar experimentos com estes atributos para descobrir quais deles melhor classificam os pares Q&A. A Seção 3.3 apresenta quais são esses atributos bem como os experimentos realizados com estes atributos;

[3] http://stackoverflow.com/questions/15376/whats-the-best-uml-diagramming-tool
[4] http://stackoverflow.com/questions/761341/boxlayout-cant-be-shared-error
[5] http://stackoverflow.com/questions/7229226/should-i-avoid-the-use-of-setpreferredmaximumminimumsize-methods-in-java-swi

- Comparar o classificador com outros algoritmos de classificação com o intuito de encontrar o algoritmo que melhor classifica os pares Q&A em categorias de domínio (i.e., aquele que possui a maior taxa de acerto). A Seção 3.3 também apresenta tabelas comparando as taxas de acerto entre estes algoritmos de classificação.

3.2.1 Construção do *Dataset* para Treinamento e Teste do Classificador

Para classificar os pares pergunta-resposta do SOF, procedeu-se com o *download* de um *data dump* público do SOF e importou-se os dados em um banco de dados relacional. A tabela *posts* deste banco armazena todas as perguntas postadas por interrogadores no *site* até a data em que o *dump* foi realizado (a versão de Março de 2013). Esta tabela contém também as respostas que foram dadas a cada pergunta, caso existam.

Foram selecionados aleatoriamente deste banco de dados relacional 400 pares de pergunta-resposta (mais pares poderiam ter sido escolhidos, porém espera-se que esta amostra represente o comportamento geral das perguntas e respostas presentes no SOF), sendo que cada par foi manualmente classificado. No processo de classificação de cada par, foram consideradas as 5 categorias definidas na Seção 3.2.

A Tabela 3.1 mostra o resultado da classificação manual realizada nos 400 pares de pergunta-resposta selecionados.

Tabela 3.1: Classificação manual dos 400 pares Q&A.

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

Como o número de pares pergunta-resposta da categoria *Debug-Corrective* foi apenas 10, tal categoria não foi considerada no processo de construção do *dataset*. Até o presente momento, não foi encontrada nenhuma aplicação prática para a categoria *Miscellaneous*. Isto pode ser explicado pela sua natureza multi-focal, em que as respostas contém uma mistura de interesses. Portanto, esta categoria também não foi considerada no processo de construção do *dataset*.

No próximo passo, foi gerado um arquivo de extensão ARFF (*Attribute-Relation File Format*), contendo a informação dos atributos de cada um dos 336 pares de pergunta-resposta classificados (i.e., a soma dos pares Q&A classificados nas categorias *How-to-do*, *Conceptual* e *Seeking-something*). Para cada par classificado, o programa atribui o valor zero para todas as variáveis que representam os atributos. Depois disso, ele processa todas as palavras do título, pergunta e resposta de cada par para identificar as palavras-chave.

Quando a palavra-chave é identificada, o programa incrementa a variável que corresponde ao atributo associado com esta palavra-chave. Em relação aos atributos booleanos, o programa verifica a existência de código-fonte e/ou *links* na pergunta e resposta de cada par, atribuindo o valor 1 em caso afirmativo e 0, caso contrário. A Figura 3.1 mostra os passos do processo de construção do *dataset*.

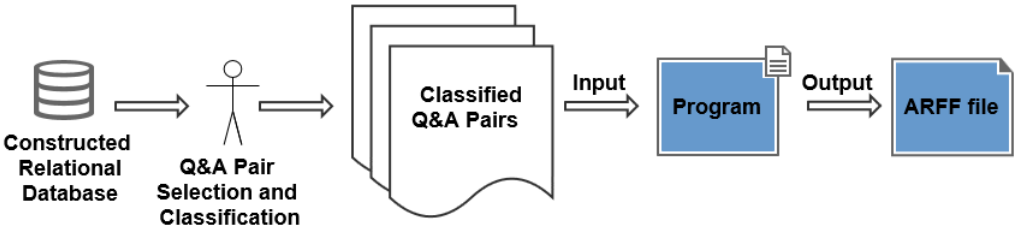

Figura 3.1: O processo de construção do *dataset* (arquivo ARFF).

3.3 Algoritmo de Classificação

Foi realizada uma comparação entre diferentes algoritmos de classificação para encontrar aquele que melhor se adequa ao *dataset* do SOF. No processo de classificação, foram utilizados sete algoritmos de classificação amplamente conhecidos nas áreas de Mineração de Dados e Reconhecimento de Padrões: *Logistic Regression* (LR) [27,38], *Naïve Bayes* (NB) [28], *Multilayer Perceptron* (MLP) [21], *Support Vector Machine* (SVM) [28], *J4.8 Decision Tree* (J4.8) [28,44], *Random Forest* (RF) [4] e *K-Nearest Neighbors* (KNN) [28].

Para a construção do classificador, foi necessário definir alguns atributos. A Subseção 3.3.1 apresenta a definição de todos os atributos que foram definidos para este classificador.

3.3.1 Definição dos Atributos do Classificador

Foram definidos 10 atributos para caracterizar os pares de pergunta-resposta. Desses atributos, 6 estão relacionados com o número de vezes que um conjunto de palavras-chave aparecem no título, pergunta e resposta de um par. Os 4 atributos restantes são booleanos e estão relacionados com a presença/ausência de código-fonte ou *links* na pergunta e resposta de um determinado par.

Como existem palavras-chave que são mais determinantes e intuitivas do que outras, foi adotado um critério de peso. Dessa forma, cada palavra-chave possui seu peso (1 ou 5). Palavras-chave de peso 1 são contadas somente uma vez, enquanto que as de peso 5 são contadas 5 vezes quando aparecem no título, pergunta ou resposta de um par. Este critério ajuda a destacar quais são as palavras mais determinantes em um par pergunta-resposta. Esta é uma forma de deixar a abordagem proposta mais robusta para trabalhar com pares pergunta-resposta de tamanhos arbitrários, sejam eles sucintos ou extensos.

A Tabela 3.2 representa a relação entre os atributos e as suas respectivas palavras-chave. As palavras que aparecem em negrito possuem peso 5, enquanto que as demais possuem peso 1. Estas palavras foram extraídas a partir de uma análise qualitativa dos 400 pares Q&A utilizados no processo de classificação manual. Esta análise teve como objetivo selecionar as palavras que frequentemente apareciam nas perguntas e respostas destes pares para cada atributo definido (i.e., "howQtd", "debugQtd", "optimalQtd", "lookingForQtd" e "conceptualQtd").

A Tabela 3.3 descreve cada um dos atributos booleanos considerados.

Tabela 3.2: Relação entre os atributos e suas respectivas palavras-chave.

Atributo	Palavras-chave
howQtd	**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
debugQtd	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
optimalQtd	optimal, efficient, better, reliable, elegant, 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
lookingForQtd	**tool(s)**, **tutorial(s)**, **manual(s)**, **book(s)**, **looking for**, **looking forward**, **looking at**, **looking around**, **searching for**, **searching forward**, **searching at**, **searching around**, **books/tutorials**, **book/site/tutorial**, client(s), find, app, application(s), lib(s), library, libraries, framework(s), migrate, migration(s), migrating, migration/upgrade, 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, vmx, xen, application/framework, app/framework, framework/application, framework/app, tools/strategy, strategy/tools, strategy, getting, started, when, ide(s), scanning, googling, blog(s), debugger(s), interpreter(s), compiler(s), comments/suggestions, suggestions/comments, 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, procedures/guides, 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/open, direction(s), free, learning, material, beginner, possibilities, provider(s)
conceptualQtd	**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), overlook, mean, signify, signification, lesson(s), understand, explanatory, purpose(s), does, conceptual
questionQtd	Retorna a quantidade de perguntas realizadas dentro do par pergunta-resposta, com base no operador de interrogação (?)

Tabela 3.3: Definição dos 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

3.3.2 Avaliação da Acurácia do Classificador

1. *Cross-validation*: Utilizou-se este método estatístico pois ele 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 é indicado quando a quantidade de dados rotulados é relativamente pequena [25]. No caso, a amostra considerada no estudo contém 336 instâncias rotuladas (como dito na Subseção 3.2.1, foram considerados somente os pares pertencentes às categorias *How-to-do*, *Conceptual* e *Seeking-something*). A forma básica do *Cross-validation* é o *k-fold cross-validation*, a qual foi utilizada no presente trabalho (foi considerado *10-fold cross-validation*, i.e., 90% dos dados para construir o modelo e testar sua acurácia nos 10% restantes).
2. *Feature Selection*: Foi utilizado o *software WEKA* para aplicar *Information Gain Filter* nos atributos de forma a selecionar os atributos mais relevantes para a classificação da amostra. A Tabela 3.4 mostra os atributos mais relevantes ordenados pelo valor de *Information Gain*. Quanto maior for o valor do *Information Gain*, melhor o atributo classifica a amostra. Os atributos que apresentaram valor de *Information Gain* igual a 0 são menos relevantes para a classificação da amostra.

Tabela 3.4: Atributos ranqueados: *Information Gain*

Valor do *Information Gain*	Atributo
0,3309	conceptualQtd
0,2486	lookingForQtd
0,2211	howQtd
0,0757	answerHasCode
0,0389	questionHasCode
0	answerHasLink
0	debugQtd
0	questionHasLink
0	optimalQtd
0	questionQtd

3.3.3 Experimentos e Resultados

Para descobrir qual algoritmo de classificação possui a maior taxa de acerto, foram realizados experimentos, que foram divididos em duas partes:

- **Parte I - Experimentos com os Atributos Originais**: Foram realizados experimentos através da interface gráfica do *WEKA* [20] considerando *k-fold* igual a 10.
- **Parte II - Experimentos com os Atributos Modificados**: Foram realizados experimentos com o PCA para investigar se uma transformação no espaço de características (geração de novos atributos a partir da combinação dos atributos originais) implicaria em um aumento da taxa de acerto dos algoritmos de classificação considerados.

Parte I - Experimentos com os Atributos Originais

Inicialmente, foram utilizados todos os atributos como entrada para os algoritmos de classificação. É possível observar na Tabela 3.5 que o algoritmo LR obteve a maior taxa de acerto (73,8095%). A menor taxa de acerto foi obtida com o algoritmo KNN (59,2262%). O valor de k considerado foi 5. Este valor representa a quantidade de vizinhos mais próximos que foi considerada. Após realizar *Feature Selection* em todos os atributos, foi feito um novo teste considerando apenas os 5 atributos mais relevantes destacados pelo filtro *Information Gain*: "howQtd", "lookingForQtd", "conceptualQtd", "answerHasCode" e "questionHasCode". A Tabela 3.6 mostra que houve uma melhora na taxa de acerto de todos os algoritmos de classificação (com exceção do SVM que se manteve constante). Uma explicação para essa melhora é que a utilização de atributos menos relevantes estavam confundindo os classificadores durante o processo de classificação.

Tabela 3.5: Todos os 10 atributos originais.

Classificador	Taxa de Acerto	Corretos	Incorretos
LR	73,8095%	248	88
NB	70,8333%	238	98
MLP	70,5357 %	237	99
SVM	70,2381%	236	100
J4.8	64,5833%	217	119
RF	71,4286%	240	96
KNN (k = 5)	59,2262%	199	137

Tabela 3.6: Feature Selection: 5 atributos mais relevantes.

Classificador	Taxa de Acerto	Corretos	Incorretos
LR	76,1905%	256	80
NB	72,9167%	245	91
MLP	75,8929%	255	81
SVM	70,2381%	236	100
J4.8	69,6429%	234	102
RF	71,7262%	241	95
KNN (k = 5)	69,9405%	235	101

A Figura 3.2 mostra um gráfico de barras comparando dois cenários de teste distintos. O "Teste 1" representa o cenário considerando todos os atributos definidos neste trabalho. Já o "Teste 2" representa somente os atributos que foram selecionados com *Feature Selection*. Em especial, os algoritmos KNN, MLP e J4.8 aumentaram significativamente as suas taxas de acerto do "Teste 1" para o "Teste 2" (mais de 5%).

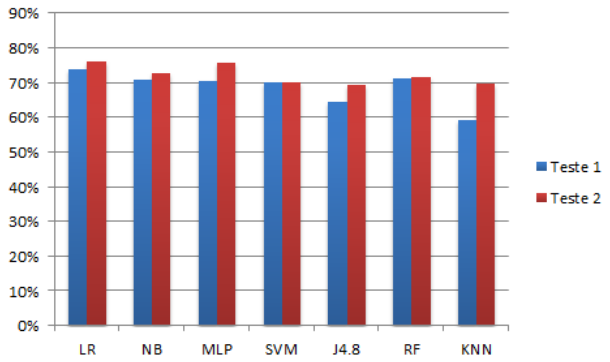

Figura 3.2: Comparação entre "Teste 1" e "Teste 2".

A Tabela 3.7 mostra a matriz de confusão do classificador LR no "Teste 2". A diagonal principal da matriz representa as instâncias classificadas corretamente pelo classificador. O classificador LR obteve melhor desempenho no "Teste 2" com pares de pergunta-resposta da categoria *How-to-do*. A matriz de confusão revela que dos 22 pares de pergunta-resposta da categoria *How-to-do* que foram classificados incorretamente, 13 são confundidos como sendo da categoria *Seeking-something*. Além disso, dos 30 pares de pergunta-resposta da categoria *Seeking-something* que foram classificados incorretamente, 16 são confundidos como sendo da categoria *How-to-do*. Isso revela uma confusão recíproca e clara do classificador LR entre estas duas categorias.

Tabela 3.7: Matriz de confusão do LR do "Teste 2"

a	b	c	<− classificado como
87	9	13	a = *How-to-do*
12	78	16	b = *Conceptual*
16	14	91	c = *Seeking-something*

A Figura 3.3 mostra um gráfico de barras comparando a taxa de acerto por categoria do classificador LR do "Teste 2". Este classificador obteve uma taxa de acerto de 79,81% em pares de pergunta-resposta da categoria *How-to-do*, 75,20% em pares da categoria *Seeking-something* e 73,58% nos pares da categoria *Conceptual*.

A Tabela 3.8 mostra a acurácia detalhada do classificador LR em cada uma das categorias. O *F-measure* [9] é uma média harmônica da *Precision* e *Recall*. Os valores variam entre 0 (pior) e 1 (melhor). Em todas as categorias, o valor do *F-measure* foi superior ou igual a 0,75.

Uma outra forma de medir a precisão é a área sob a curva de ROC [6]. Por exemplo,

[6] `http://gim.unmc.edu/dxtests/roc3.htm`

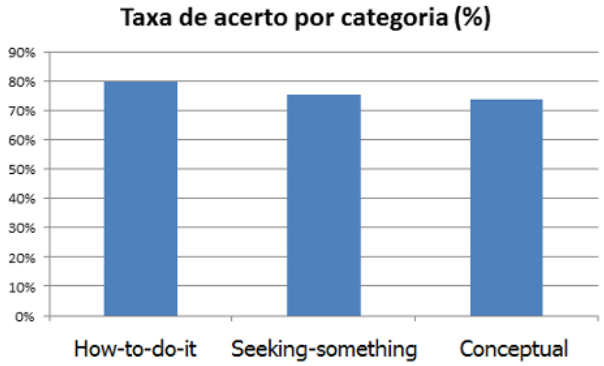

Figura 3.3: Taxa de acerto(%) do LR por categoria.

uma área de 1 representaria um teste perfeito, enquanto que uma área de 0,5 representaria um teste inútil. Nota-se que a acurácia da categoria *How-to-do* está excelente, enquanto que as categorias *Conceptual* e *Seeking-something* estão com uma acurácia boa.

Tabela 3.8: Acurácia detalhada por categoria

Categoria	F-Measure	Área ROC
How-to-do	0,757	0,919
Conceptual	0,772	0,891
Seeking-something	0,758	0,884

Tabela 3.9: Intervalos e Interpretação da área ROC

Intervalo	Interpretação
0,90 - 1,00	Excelente
0,80 - 0,90	Bom
0,70 - 0,80	Razoável
0,60 - 0,70	Pobre
0,50 - 0,60	Ruim

Parte II - Experimentos com os Atributos Modificados

Na tentativa de melhorar a taxa de acerto dos classificadores, foram realizados dois testes com o PCA. No primeiro teste (PCA1), aplicou-se o PCA em todos os 10 atributos originais, gerando um total de 10 componentes principais. A Tabela 3.10 mostra os resultados obtidos. Já no segundo teste (PCA2), aplicou-se o PCA apenas nos 5 atributos mais relevantes selecionados anteriormente com *Feature Selection*, gerando um total de 5 componentes principais. A Tabela 3.11 refre-se ao teste PCA2. Todos os componentes principais gerados em ambos os testes foram considerados na classificação.

Observa-se que todos os algoritmos de classificação aumentaram a sua taxa de acerto no PCA2 em relação ao PCA1. Outro aspecto notável é a baixa acurácia do algoritmo de classificação J4.8 no PCA1 (56,8452%), que melhorou significativamente no PCA2 (72,3214%).

Tabela 3.10: Aplicando o PCA nos 10 atributos.

Classificador	Taxa de Acerto	Corretos	Incorretos
LR	73,8095%	248	88
NB	70,8333%	238	98
MLP	70,8333%	238	98
SVM	71,4286%	240	96
J4.8	56,8452%	191	145
RF	64,2857%	216	120
KNN (k = 5)	63,3929%	213	123

Tabela 3.11: Aplicando o PCA nos 5 atributos mais relevantes.

Classificador	Taxa de Acerto	Corretos	Incorretos
LR	76,1905%	256	80
NB	74,1071%	249	87
MLP	73,8095%	248	88
SVM	72,619%	244	92
J4.8	72,3214%	243	93
RF	72,9167%	245	91
KNN (k = 5)	71,4286%	240	96

Um gráfico de barras comparando os testes efetuados com PCA é mostrado na Figura 3.4.

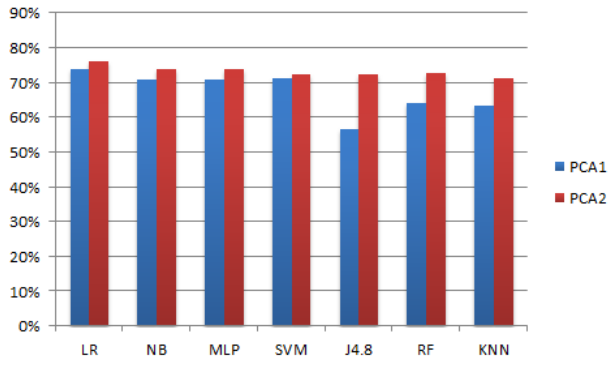

Figura 3.4: Comparação entre PCA1 e PCA2.

A Figura 3.5 mostra uma comparação entre as melhores configurações para este problema, ou seja, não utilizar o PCA nos 5 atributos mais relevantes ("Teste 2") e utilizar o PCA nos 5 atributos mais relevantes (PCA2). A conclusão é de que o PCA2 é a melhor configuração geral para o problema, uma vez que a maioria dos algoritmos de classificação obtiveram melhores taxas de acerto com o uso do PCA: NB, SVM, J4.8, RF e KNN. O algoritmo LR manteve-se constante nas duas abordagens (76,19%). Apenas no algoritmo MLP houve uma queda na taxa de acerto com o uso do PCA.

Após a realização destes experimentos, o classificador LR utilizando os 5 atributos mais relevantes foi o escolhido. Esta configuração foi a que obteve a maior taxa de acerto, ou seja, 76,1905%.

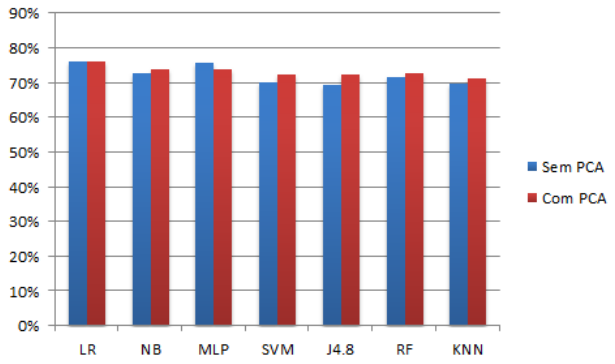

Figura 3.5: Comparação entre "Teste 2" (Sem PCA) e PCA2.

3.4 Avaliação do Ranqueamento de Pares Q&A

Esta seção aborda qual é o objetivo da recomendação de pares Q&A. Além disso, são apresentadas as APIs consideradas nos experimentos, além de apresentar detalhadamente como foi o processo de construção dos índices. Finalmente, é apresentado o desenho experimental e como foi realizada a avaliação do ranqueamento de pares Q&A.

3.4.1 Objetivo da Recomendação de Pares Q&A

O objetivo de recomendar pares Q&A é auxiliar os desenvolvedores nas suas tarefas de desenvolvimento, considerando que os pares Q&A recomendados devem ter similaridade textual com a tarefa de desenvolvimento e devem ter sido bem avaliados pela comunidade do SOF. A estratégia proposta visa recomendar pares Q&A ao invés de *threads* inteiras pois as respostas para uma mesma pergunta podem ter diferentes *scores*, i.e., algumas respostas podem ser melhores do que outras. O *score* de um par (i.e., o número de votos positivos menos o número de votos negativos) pode ser considerado como um indicativo da sua qualidade pois esta é a principal maneira que a comunidade do SOF possui para avaliar o conteúdo do *site*. Então, é desejável que os pares Q&A recomendados sejam altamente relevantes no contexto da tarefa do usuário.

3.4.2 APIs Consideradas

Foram realizados experimentos em três APIs pertencentes às seguintes linguagens de programação (Java, C++ e linguagens .NET) amplamente utilizadas na indústria de desenvolvimento de *software*: *Swing*, *Boost* e *Language Integrated Query (LINQ)*:

- **Swing**: É um conjunto de ferramentas criado para permitir o desenvolvimento corporativo em Java. Portanto, os desenvolvedores podem utilizar o *Swing* para criar aplicações Java de larga escala utilizando uma grande variedade de componentes poderosos [14, 16].

- **Boost**: É uma coleção de bibliotecas C++. Cada biblioteca foi revisada por muitos desenvolvedores profissionais antes de ser aceita no *Boost*. Estas bibliotecas foram testadas em várias plataformas utilizando muitos compiladores e as implementações da biblioteca padrão do C++ [39].
- **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 [22].

3.4.3 Construção dos Índices Lucene

O motor de busca Apache Lucene [3] foi usado para indexar os dados. Para uma dada API (e.g., *Swing*), foram obtidas todas as *threads* do banco de dados em que a pergunta possui uma *tag* específica (e.g., "swing"). Então, a partir deste conjunto de *threads*, todos os pares Q&A (e.g., se a *thread* possui uma pergunta e n respostas, n pares Q&A para aquela *thread*). A Tabela 3.12 mostra o número de pares Q&A obtidos para cada tópico considerado no trabalho.

Tabela 3.12: Total de pares Q&A por API.

API	Linguagem de Programação	Total de pares Q&A
Boost	C++	14,558
Swing	Java	45,239
LINQ	C#	60,035

O próximo passo foi classificar esses pares Q&A de forma a considerar somente os pares da categoria *How-to-do*. A Tabela 3.13 mostra os resultados da classificação dos pares Q&A para cada API.

Tabela 3.13: Resultados da classificação por API.

API	*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

Para cada par classificado como *How-to-do*, foram removidas as *tags* HTML utilizando a biblioteca Java denominada *HTML Cleaner* [7]. Além disso, foram removidas as palavras irrelevantes (do inglês, *stop words*) e foi realizado *stemming* no seu conteúdo (título da pergunta, corpo da pergunta e corpo da resposta, excluindo os trechos de código-fonte) utilizando o algoritmo de *Stemming* definido por *Porter* [41].

Como as perguntas e respostas do SOF 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.

[7] `http://htmlcleaner.sourceforge.net`

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/interfaces/métodos é devido ao fato de que o autor da pergunta pode estar querendo executar uma tarefa específica utilizando um elemento particular da API (e.g., 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 as classes/métodos/*structs* que estão sendo criados ou invocados.

Para o *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 (e.g., *"OrderByDescending"*, *"SelectMany"*) e um processo similar ao descrito para o *Swing* foi realizado no nome desses operadores. 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 3.4.4), é apresentada a utilização deste índice para pesquisar pares Q&A. A Tabela 3.14 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 3.14 também mostra o número de termos diferentes nos documentos para cada API.

Tabela 3.14: Informação dos índices por API.

API	Número de documentos	Número de termos
Boost	7,125	55,383
Swing	26,374	187,914
LINQ	39,592	263,502

3.4.4 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 EANC - Elemento da API não-conhecido* e *Cenário EAC - Elemento da API conhecido*.

O *Cenário EANC* corresponde à situação em que o desenvolvedor possui uma tarefa de desenvolvimento em mãos, que deve ser solucionada utilizando uma API particular (e.g., *"Boost"*), mas ele não sabe qual elemento da API (e.g., 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 EAC* corresponde à situação em que o desenvolvedor precisa resolver uma tarefa de desenvolvimento utilizando um elemento particular de uma determinada API (e.g., uma classe). 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 EAC* da mesma maneira do que foi descrito para o *Cenário EANC*. 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, foi adicionado 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*.

3.4.5 Ranqueamento dos Pares Q&A pelo *Score* do SOF

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 SOF. O *score* de um *post* (pergunta ou resposta) tem sido usado como um *proxy* para avaliar sua qualidade pois o mecanismo de votação no SOF é a principal forma que os membros possuem de avaliar o conteúdo do *site* ([29], [52], [36]).

Devido ao fato de que cada *post* individual no SOF 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, foi necessário definir 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 a 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. Foi definido, arbitrariamente, os valores 0,7 e 0,3 para os pesos da resposta e pergunta de um par. Desta forma, é possível calcular o *score* do SOF 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:

$$ScoreParQ\&A = (0,7 * ScoreResposta) + (0,3 * ScorePergunta); \tag{3.1}$$

Na próxima Subseção (ver Subseção 3.4.6), o *score* do SOF 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.

3.4.6 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 que o *score* do SOF indica o quanto o par foi bem votado pela comunidade do *site*. Ambos 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, foi necessário realizar uma etapa de normalização. A técnica de normalização consiste basicamente em colocar o valor do *score Lucene* e do *score* do SOF no intervalo de [0,1]. A razão para essa etapa de normalização é que geralmente o *score* do SOF é muito maior que o *score Lucene*, i.e., estas métricas possuem naturezas diferentes. Depois desta etapa de normalização, foi necessário calcular 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.

3.4.7 Critérios de Avaliação

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

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. 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 ou esses trechos não podem ser compilados. 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, estes trechos podem ser compilados depois de alguns pequenos ajustes (e.g., 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, foi definido o valor 3. Nos casos em que o trecho de código possui apenas uma linha de código solta 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.

3.4.8 Desenho 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* [14], que contém somente tarefas relacionadas com GUI (*Graphical User Interfaces*). Existem 14 tarefas neste capítulo e foram selecionadas aleatoriamente 12 delas.

As tarefas do *Boost* foram extraídas do livro *Boost Cookbook* [39]. Foram selecionadas aleatoriamente 12 das 91 tarefas disponíveis neste *cookbook*.

As tarefas do *LINQ* foram retiradas de um *blog* [8] 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*.

A idéia 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 idéia 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.

As atividades de desenvolvimento a serem retiradas desses *Cookbooks* foram sorteadas, evitando assim, a possibilidade de serem selecionadas algumas atividades de desenvolvimento mais prováveis de serem encontradas no SOF.

Uma análise manual qualitativa da recomendação para as 35 tarefas foi realizada. A Figura 3.6 mostra o desenho experimental do experimento proposto. Para cada API (*Swing*, *Boost* e *LINQ*), foram realizados experimentos para testar dois cenários: *Cenário EANC* e *Cenário EAC*. Para as 12 tarefas previamente selecionadas para *Swing*, foram selecionadas aleatoriamente 6 para o *Cenário EANC* e 6 para o *Cenário EAC*. O mesmo foi feito para o *Boost*. Para o *LINQ*, como são 11 tarefas, foram selecionadas aleatoriamente 6 e 5 tarefas para os *Cenários EANC e EAC*, respectivamente. As consultas de entrada

[8] `http://blogs.msdn.com/b/vbteam/archive/tags/linq+cookbook/`

para o *Cenário EANC* foram os títulos das tarefas (após realizar o processo de *stemming* e remoção das *stop words*). A consulta para o *Cenário EAC* foi a *string* formada pelo título da tarefa concatenada com o nome da classe (no caso de *Swing* ou *Boost*) ou operador (no caso do *LINQ*) que era importante na solução apresentada nos *cookbooks* originais, a partir dos quais as tarefas foram extraídas.

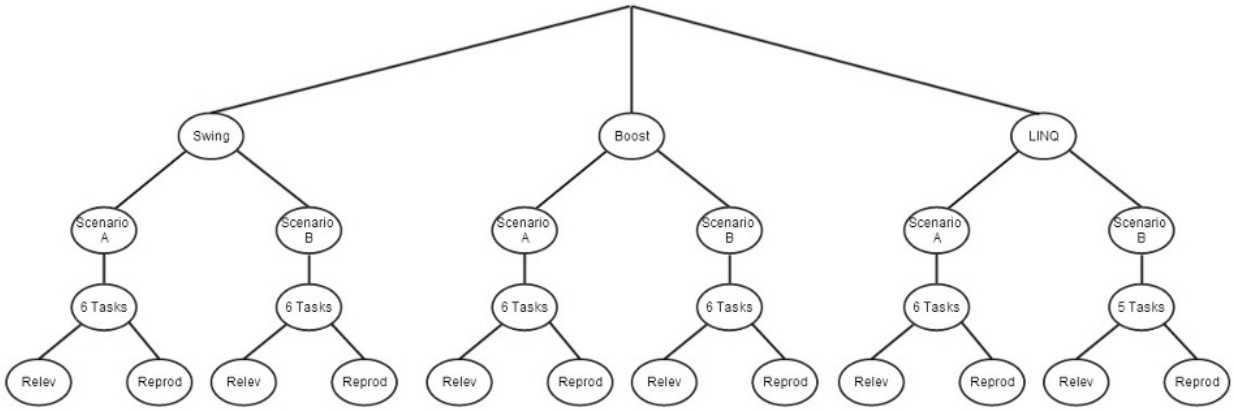

Figura 3.6: Desenho Experimental com a distribuição das tarefas por API.

A Tabela 3.16 mostra para o *Swing*, as 12 tarefas selecionadas para o experimento. As primeiras 6 foram incluídas no *Cenário EANC* e as 6 restantes no *Cenário EAC*. As tarefas para o *Cenário EAC* são mostradas com seu título modificado. Por exemplo, o título da tarefa 13.5 é originalmente *"Action Handling: Making Buttons Work"*. Na Tabela 3.16, o título desta tarefa foi apresentado como *"Action Handling: Making Buttons Work* ***ActionListener"***, pois *"ActionListener"* foi o nome da classe escolhido para ser concatenado com o título, uma vez que esta é uma classe chave utilizada na solução deste problema. Após a remoção das *stop words* (caso existam) e aplicação do processo de *stemming*, a *string* resultante foi utilizada como consulta para retornar os pares Q&A. Tabelas similares são mostradas para *Boost* e *LINQ* (Tabelas 3.21 e 3.26, respectivamente). Nestas tabelas, o nome das classes ou operadores utilizados para o *Cenário EAC* são mostrados em negrito.

Dois sujeitos humanos estudantes de mestrado em Engenharia de *Software* e com experiência de 3 anos na indústria de *software* (Autor A e Autor B) individualmente avaliaram para cada uma das 35 tarefas, os primeiros 10 pares recomendados. Para cada par, eles deram nota para os dois critérios previamente descritos na Subseção 3.4.7. Na Tabela 3.15, a coluna *"Kappa* Antes" mostra o *Kappa* Ponderado [10] calculado para medir a concordância entre os dois avaliadores. Nesta tabela, cada linha representa a tríplice "API/Cenário/Critério". Portanto, na primeira linha, o *Kappa* Ponderado foi calculado para comparar as notas para o critério *Relev* dada pelos dois avaliadores para os pares Q&A retornados para as 6 tarefas selecionadas para o *Cenário EANC* da API *Swing* (então, com exceção do *LINQ Cenário EAC*, cada linha representa uma comparação entre 60 valores, uma vez que os avaliadores analisaram os primeiros 10 pares Q&A

recomendados para cada tarefa).

Tabela 3.15: *Kappa* Ponderado - Comparação da Concordância

API/Cenário/Critério	*Kappa* Antes	*Kappa* Depois
Swing/EANC/Relev	0,6	0,89
Swing/EANC/Reprod	0,84	0,95
Swing/EAC/Relev	0,58	0,92
Swing/EAC/Reprod	0,86	0,98
Boost/EANC/Relev	0,54	0,95
Boost/EANC/Reprod	0,94	0,95
Boost/EAC/Relev	0,81	0,94
Boost/EAC/Reprod	0,67	0,98
LINQ/EANC/Relev	0,95	0,97
LINQ/EANC/Reprod	0,81	0,93
LINQ/EAC/Relev	0,68	0,92
LINQ/EAC/Reprod	0,87	0,94

No próximo passo, as avaliações dos avaliadores foram comparadas. Os pares Q&A em que a diferença das notas dadas pelos autores foram maior ou igual a 2 foram marcados para uma discussão posterior (e.g., considere que o Autor A deu nota 4 para o critério *Relev* de um par Q&A e o Autor B deu nota 1 para o critério *Relev* para o mesmo par Q&A). A razão por considerar somente as diferenças maiores ou iguais a 2 é porque uma diferença de 1 não foi considerada como uma discordância significativa entre os avaliadores (e.g., o Autor A deu nota 3 para o critério *Relev* de um par Q&A e o Autor B avaliou o critério *Relev* deste mesmo par Q&A como 4).

Após marcar estes pares Q&A com maior divergência, os avaliadores discutiram cada um até chegar a um acordo sobre eles. Após modificarem as notas nesta etapa de discussão, o *Kappa* Ponderado foi novamente calculado e os valores estão mostrados na coluna *"Kappa* Depois" da Tabela 3.15. Comparando os valores antes e depois desta etapa de discussão, observa-se que a concordância entre eles melhorou (um *Kappa* Ponderado com valor de "1" significa uma concordância perfeita). Nos resultados são consideradas as avaliações do Autor A, uma vez que a concordância geral foi alta.

3.5 Resultados

Nesta seção, são apresentados os resultados do experimento proposto. As Tabelas 3.17, 3.18, 3.19, 3.20, 3.22, 3.23, 3.24, 3.25, 3.27, 3.28, 3.29, 3.30, possuem a mesma estrutura de colunas: identificador da tarefa e as primeiras 10 posições do *ranking* (P1 até P10). Cada posição no *ranking*, corresponde a um par Q&A recomendado. As tabelas mostram o ranqueamento obtido para cada tarefa, considerando a API (*Swing*, *Boost* ou *LINQ*), o cenário (*EANC* ou *EAC*) e o critério (*Relev* ou *Reprod*).

Por exemplo, na Tabela 3.17, na primeira linha de dados, apresenta-se os resultados para a tarefa 13.14. Observa-se que o melhor par Q&A ranqueado (coluna P1) recebeu

Tabela 3.16: Tarefas do *Swing*

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

Tabela 3.17: *Swing - Cenário EANC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.14	2	2	4	2	2	4	2	2	2	2
13.13	0	4	4	4	3	3	1	3	3	4
13.11	1	2	2	0	2	2	3	0	1	2
13.3	3	0	0	2	0	0	2	3	0	0
13.1	4	4	0	3	3	4	4	4	2	3
13.8	3	1	2	3	4	2	2	2	1	4

Tabela 3.18: *Swing - Cenário EANC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.14	0	0	4	3	0	3	4	3	2	0
13.13	3	0	0	4	4	0	0	0	4	3
13.11	0	0	0	3	3	3	4	4	3	0
13.3	0	4	0	0	0	0	4	0	0	0
13.1	4	4	4	4	4	4	4	0	3	3
13.8	0	0	0	3	4	3	0	4	4	4

nota 2 (neutro). Além disso, observa-se pares Q&A altamente relevantes nas posições P3 e P6.

A métrica denominada *Normalized Discounted Cumulative Gain* (NDCG) foi utilizada para obter uma avaliação numérica do ranqueamento dos pares Q&A recomendados nos experimentos. Esta métrica é geralmente utilizada para avaliar os resultados retornados pelos motores de busca e utiliza uma noção multivalorada de relevância [31]:

$$NDCG(Q,k) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{M(j,m)}-1}{log_2(1+m)} \tag{3.2}$$

onde k é a quantidade de resultados retornados no *ranking*. Foi estabelecido k = 10, uma vez que são recomendados em cada consulta, 10 pares Q&A para o usuário do

Tabela 3.19: *Swing* - *Cenário EAC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.12	4	2	3	3	4	0	2	0	2	2
13.2	3	1	2	1	4	4	1	0	0	2
13.9	4	4	4	1	1	0	0	4	0	0
13.4	4	2	2	2	3	2	1	3	3	3
13.5	4	2	4	2	2	2	2	2	2	3
13.6	2	4	4	3	3	3	3	3	4	4

Tabela 3.20: *Swing* - *Cenário EAC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
13.12	0	0	0	4	3	3	4	0	2	0
13.2	4	0	0	0	0	0	4	0	2	2
13.9	4	4	4	3	4	4	4	4	0	0
13.4	4	0	0	3	4	4	0	4	4	4
13.5	0	4	0	0	0	4	0	0	0	3
13.6	3	4	3	3	4	4	4	4	4	4

Tabela 3.21: Tarefas do *Boost*

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

Tabela 3.22: *Boost* - *Cenário EANC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2.8	3	2	3	3	3	2	2	2	3	3
3.1	1	2	0	0	0	0	0	0	0	1
12.5	1	0	0	0	4	0	4	4	4	4
12.7	2	4	0	0	0	2	2	2	4	0
10.7	3	2	0	1	2	0	0	0	0	0
3.5	3	2	2	2	3	2	2	3	2	3

Tabela 3.23: *Boost - Cenário EANC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2.8	2	4	4	4	4	0	4	4	4	4
3.1	4	4	4	0	4	1	0	2	0	0
12.5	0	0	0	0	4	0	4	4	0	0
12.7	0	0	0	3	4	0	0	4	3	4
10.7	4	0	4	0	4	0	3	4	4	0
3.5	4	0	4	0	0	4	0	0	0	4

Tabela 3.24: *Boost - Cenário EAC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
7.7	2	0	0	0	0	0	1	1	0	0
10.2	2	2	1	0	0	0	0	0	0	1
1.11	3	1	3	1	1	3	3	2	2	1
9.2	2	1	1	1	3	1	3	1	1	0
7.2	4	4	4	4	3	4	0	2	2	4
8.8	1	1	1	1	1	1	1	1	1	1

Tabela 3.25: *Boost - Cenário EAC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
7.7	4	0	4	1	4	2	4	4	4	4
10.2	4	0	0	2	4	4	0	0	0	0
1.11	0	3	0	4	2	0	4	4	4	2
9.2	0	4	0	3	4	4	3	4	2	2
7.2	0	4	4	4	4	4	4	4	0	4
8.8	4	4	4	4	2	4	3	3	3	4

Tabela 3.26: Tarefas do *LINQ*

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

Tabela 3.27: *LINQ - Cenário EANC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2	0	0	0	2	2	0	2	0	0	0
10	1	2	4	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
5	2	0	2	1	2	2	2	2	0	2
11	0	0	0	0	0	0	2	0	0	0
12	4	4	4	4	1	4	3	1	3	0

Tabela 3.28: *LINQ - Cenário EANC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
2	3	0	3	4	4	4	4	0	3	3
10	0	0	3	3	2	0	3	0	0	0
1	4	1	0	0	2	2	2	4	4	0
5	3	2	4	0	3	4	3	3	3	3
11	3	4	4	4	4	4	4	4	3	0
12	3	3	3	3	0	3	4	2	4	0

Tabela 3.29: *LINQ - Cenário EAC* (0 = Não Relevante, 4 = Altamente Relevante)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
3	1	1	2	1	4	1	0	2	1	1
7	4	2	2	4	4	4	4	4	4	4
4	0	1	2	0	3	0	1	0	0	0
9	2	4	4	2	2	4	4	1	4	2
8	0	2	1	2	1	0	0	3	4	3

Tabela 3.30: *LINQ - Cenário EAC* (0 = Não Reprodutível, 4 = Altamente Reprodutível)

Task	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
3	4	4	4	4	3	3	3	4	4	2
7	3	4	4	3	2	3	3	2	4	3
4	3	0	3	3	4	3	0	0	0	0
9	3	4	4	3	3	4	3	2	3	2
8	3	0	0	0	2	0	0	0	4	0

sistema de recomendação. *M(j,m)* é o valor da métrica dada para um documento m (no caso, um documento é um par Q&A) para a *query* j (no caso, a *query* corresponde ao título da tarefa de programação retirada do *Cookbook*). Como foram considerados dois critérios nos experimentos *M(j,m)* pode ser a nota dada para o critério *Relev* ou *Reprod*. Foi calculado o valor do NDCG para ambos os critérios. Z_{kj} é o fator de normalização calculado de modo que o ranqueamento seja perfeito (i.e., NDCG igual a 1). Utilizou-se a mesma abordagem de um trabalho relacionado [40] para calcular este fator, onde ele é calculado considerando o cenário em que todos os documentos retornados possuem a nota máxima (i.e., o valor 4). Uma avaliação com esta métrica possibilita comparar parcialmente os resultados de ambos os trabalhos (não totalmente porque foi considerado

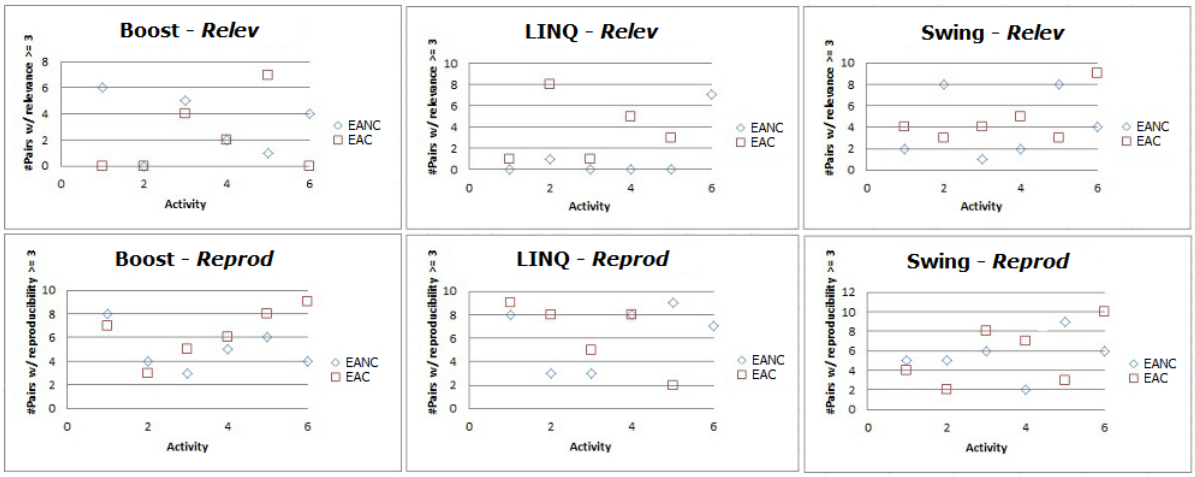

Figura 3.7: Número de pares que possuem *Relev* ≥ 3 (Primeira Linha) e *Reprod* ≥ 3 (Segunda Linha).

os top-10, enquanto que eles consideraram os top-15). Entretanto, utilizar o NDCG para esta avaliação parece ser inadequado pois somente as consultas que tiverem quase todos os 10 pares Q&A com notas de relevância próximas de 4 irão ter um alto valor de NDCG e isto parece ser extremamente rigoroso. O interesse é que existam alguns pares relevantes melhor ranqueados, i.e., não necessariamente todos os 10 pares Q&A precisam ser altamente relevantes porque se o primeiro par Q&A altamente relevante recomendado resolve o problema, então este par satisfaz a consulta. O fator de normalização que foi calculado utilizando esta abordagem é $Z_{kj} \approx 0{,}01$. $Q = 35$, pois foram consideradas 35 tarefas de desenvolvimento nos experimentos realizados.

O NDCG calculado foi 0,3583 and 0,5243 para os critérios *Relev* e *Reprod*, respectivamente.

Considerando que é desejável ter pares Q&A com notas altas entre os 10 primeiros (top-10), foi analisado o número de pares Q&A recomendados pela abordagem proposta que possuem os critérios *Relev* ou *Reprod* maior ou igual a 3. Para esta análise, os gráficos mostrados na Figura 3.7 foram utilizados. Os três gráficos na primeira linha da figura consideram o critério *Relev*, enquanto que os outros três gráficos da segunda linha da figura consideram o critério *Reprod*.

Cada gráfico considera as atividades selecionadas para o *Cenário EANC* (símbolo de diamante) e as atividades do *Cenário EAC* (símbolo de quadrado). As atividades aparecem nos gráficos na ordem mostrada nas Tabelas 3.21,3.26 e 3.16. Portanto, no primeiro gráfico da Figura 3.7 (i.e., gráfico *"Boost - Relev"*), a primeira atividade corresponde a *"Parsing date-time input"* (*Cenário EANC*) e *"Using a reference to string type* ***string_ ref****"* (*Cenário EAC*). Além de considerar o valor 4 para o critério *Relev*, foi considerado também o valor 3 pois, embora não sendo o melhor caso, os pares Q&A avaliados com nota 3 para o critério *Relev* poderiam ser utilizados para resolver quase todo o problema representado pela consulta no sistema de recomendação. Similarmente, pares Q&A

com *Reprod* igual a 3, possuem trechos de código-fonte quase completos. Como é possível visualizar nos gráficos, todas as atividades para *Swing* possuem pares Q&A com *Relev* ≥ 3. Somente 8 das 35 tarefas não tinham pares Q&A com *Relev* ≥ 3 entre os top-10 recomendados. Todas as 35 atividades tinham pelo menos um par Q&A com *Reprod* ≥ 3.

3.6 Discussão dos Resultados

Os resultados obtidos no presente trabalho foram parcialmente comparados com os resultados de Ponzanelli et al. [40] com o intuito de avaliar a estratégia de recomendação proposta. No trabalho de Ponzanelli et al., foi apresentado um *plugin* para Eclipse IDE denominado SEAHAWK com o objetivo de recomendar conteúdo proveniente do SOF para ajudar os desenvolvedores a solucionar problemas de programação. As principais diferenças entre os trabalhos são:

- No presente trabalho quatro critérios foram considerados no processo de recomendação de pares Q&A: a similaridade textual que os pares Q&A possuem com a consulta, o *score* dos pares Q&A, a caracterização *How-to-do* destes pares e a *tag* associada ao par, que corresponde ao nome da API. Já na abordagem deles, somente a relevância textual foi considerada;
- As tarefas em ambos os trabalhos são diferentes: enquanto que no presente trabalho foram selecionadas aleatoriamente as tarefas consideradas no experimento de *cookbooks*, no artigo de Ponzanelli et al., as atividades foram selecionadas de um curso de programação em Java. As tarefas extraídas de *cookbooks* possuem um caráter mais prático do que as tarefas retiradas de um curso de programação, uma vez que as tarefas presentes nos *cookbooks* estão mais relacionadas com os problemas de programação que um desenvolvedor pode enfrentar diariamente. Já as tarefas retiradas de um curso de programação geralmente estão mais relacionadas a itens didáticos utilizados para ensinar um determinado assunto;
- Foi realizado um passo de classificação para obter os pares da categoria *How-to-do*. Apenas estes pares Q&A foram considerados no processo de recomendação. Ponzanelli et al. consideraram todos os tipos de perguntas na abordagem de recomendação deles. A etapa de classificação é importante, uma vez que nesta fase são descartados os pares Q&A que são mais "teóricos" do que "práticos" (e.g., pares das categorias *Conceptual* ou *Seeking-something*);
- Nos experimentos de Ponzanelli et al., somente as tarefas Java foram consideradas, ao contrário do trabalho proposto nesta dissertação, em que a abordagem foi testada utilizando tarefas de programação de três diferentes APIs (*Swing*, *Boost* e

LINQ), que estão relacionadas a diferentes linguagens de programação (Java, C++ e linguagens .NET, respectivamente);

- No artigo de Ponzanelli et al., as *threads* inteiras do SOF são recomendadas, ao contrário do trabalho proposto nesta dissertação, em que pares Q&A individuais são recomendados, uma vez que para uma pergunta podem existir respostas bem votadas e respostas mal votadas. O raciocínio por trás desta abordagem é recomendar somente as partes das *threads* Q&A que foram bem avaliadas pela multidão do SOF;
- Foram definidos dois diferentes critérios para serem utilizados nos experimentos deste trabalho de dissertação: Relevância e Reprodutibilidade. Ponzanelli et al. somente utilizou o critério Relevância;
- No desenho experimental do presente trabalho, uma avaliação feita por dois diferentes sujeitos foi apresentada. Ponzanelli et al. não apresentaram a maneira que os resultados foram avaliados (e.g., por um ou dois autores);
- SEAHAWK recomenda 15 Q&A *threads*. Nos experimentos apresentados no presente trabalho, somente os 10 primeiros pares Q&A foram recomendados, embora este número pode ser configurado conforme a necessidade do usuário.

Embora existam muitas diferenças entre os trabalhos, ambos utilizaram a métrica NDCG para avaliação dos experimentos. O valor de NDCG obtido neste trabalho para o critério *Relev* é 3,95 vezes maior do que o valor obtido por Ponzanelli et al. (0,3583 e 0,0907 respectivamente), o que sugere que a abordagem proposta supera a deles.

Analisando a Figura 3.7, observa-se que para o critério *Relev*, a API *Swing* obteve melhores resultados em relação às outras APIs *Boost* e *LINQ*, uma vez que para todas as tarefas, este critério tinha no mínimo um par Q&A com $Relev \geq 3$ para ambos os *Cenários EANC e EAC*. Para a API *LINQ*, 2 das 6 tarefas possuíam pares Q&A recomendados com $Relev \geq 3$ no caso do *Cenário EANC* e para o *Cenário EAC*, todas as 5 tarefas tinham no mínimo um par Q&A com $Relev \geq 3$. Para a API *Boost*, 5 tarefas do *Cenário EANC* e 3 do *Cenário EAC* possuíam pelo menos um par Q&A recomendado com $Relev \geq 3$.

Existem duas principais razões para explicar o baixo número de pares com $Relev \geq 3$ para algumas tarefas. Primeiro, a abordagem proposta usa o título da tarefa como entrada para o motor de busca. Algumas tarefas não possuem uma informação precisa no seu título. Por exemplo, a tarefa *"Desktop Search Statistics"* do *LINQ* tem o objetivo de pesquisar no sistema de arquivos do computador e contar o número de itens que são documentos, imagens ou e-mails. Usando somente a informação do título, não é possível obter um resultado de busca satisfatório. Em outras palavras, algumas tarefas não possuem títulos suficientemente específicos. Outra razão que poderia justificar que algumas tarefas não recomendaram pares Q&A com $Relev \geq 3$ é a falta de informação

relacionada com a tarefa no conhecimento da multidão do SOF. Para as 35 tarefas, 27 possuíam no mínimo um par Q&A recomendado com $Relev \geq 3$ e 19 possuíam no mínimo um par Q&A recomendado com $Relev = 4$. É possível que no futuro, o *dataset* contenha um maior número de *posts* e estas tarefas tenham sido cobertas pela multidão do SOF.

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

- O uso de 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-fonte que utiliza um objeto *widget* (e.g., um botão), mas não mostra como criar este objeto. Embora criar um botão em *Swing* seja muito simples para a maioria das pessoas que possuem alguma experiência em programação com GUIs, essa tarefa pode não ser trivial para alguém novato com programação com GUI;
- A omissão de algumas linhas de código (e.g., algumas respostas usam "..." para indicar que algumas linhas foram omitidas em um trecho de código-fonte). Novamente, esta falta de informação dificulta o uso de um trecho de código-fonte em um ambiente de programação como uma IDE.

Embora não esteja mostrado na Figura 3.7, 26 das 35 tarefas possuíam no mínimo um par Q&A recomendado que possuía ambos os critérios *Relev*, $Reprod \geq 3$. Esta é uma importante métrica pois não há nenhum valor em pares Q&A que são altamente reprodutíveis, mas não são relevantes com o problema do desenvolvedor.

Algumas características da abordagem proposta podem explicar o baixo desempenho da estratégia de recomendação em alguns pontos:

- Os pares Q&A das categorias *Miscellaneous* ou *Debug-corrective* serão classificados incorretamente pelo classificador em uma das outras 3 categorias: *How-to-do*, *Conceptual* ou *Seeking-something*;
- Os valores dos pesos usados para diferenciar entre os termos menos relevantes (peso igual a 1) e os termos mais relevantes (peso igual a 5) são arbitrários e valores diferentes podem produzir resultados de classificação diferentes;
- Os valores dos pesos (0,7 e 0,3) utilizados para calcular o *score* de um par Q&A do SOF foram arbitrariamente definidos. Entretanto, resultados razoáveis com esta escolha foram atingidos.

3.6.1 Ameaças à Validade

O processo de classificação dos pares Q&A do SOF em categorias, para a construção do *dataset* de treinamento e testes, é sujeito a variações, pois existe a possibilidade de discordância entre as pessoas sobre a categoria do par Q&A. Esta ameaça foi mitigada com uma estratégia de alinhamento de avaliações auxiliada pela análise de concordância *Kappa*.

A escolha das palavras-chave para cada atributo do classificador também é sujeita a variações, pois cada pessoa pode definir um diferente conjunto de palavras-chave.

Uma análise qualitativa envolvendo uma inspeção manual das recomendações foi necessária para obter uma interpretação rica. Todavia, esta interpretação detalhada é sujeita a variações. Apesar da variabilidade inerente do processo, muitos fatores contribuíram para mitigar esta variabilidade melhorando a robustez da avaliação. A avaliação dos pares Q&A recomendados foi feita em duas fases. Primeiro, cada um dos dois avaliadores realizou uma avaliação individual. Depois, uma avaliação consensual foi realizada para diminuir o viés (i.e., os valores do *Kappa* Ponderado melhoraram após a etapa de discussão).

Outra ameaça à validade diz respeito às classes selecionadas para compor as consultas para as tarefas do *Cenário EAC*, uma vez que esta escolha está sujeita a variações e escolhas diferentes poderiam levar a resultados diferentes. Esta ameaça foi mitigada com uma restrição de que os nomes de elementos da API (i.e., nomes de classes ou nomes de operadores, no caso do *LINQ*) deveriam obrigatoriamente ser retirados das receitas dos *cookbooks* considerados. Além disso, o elemento da API que mais aparecia dentro da receita, era o escolhido.

3.7 Trabalhos Relacionados

A revisão da literatura correlata foi dividida em duas subseções: Subseção 3.7.1 e Subseção 3.7.2. A Subseção 3.7.1 revisa os trabalhos que foram feitos no *site* SOF com o intuito de classificar as perguntas do *site* de acordo com as preocupações do usuário ao postar a pergunta. Já a Subseção 3.7.2 revisa os trabalhos sobre Sistemas de Recomendação para Engenharia de *Software*.

3.7.1 Classificação das Perguntas do SOF

Várias abordagens na literatura conduziram estudos sobre o *site* Q&A SOF. Nasehi et al. [36] realizou uma análise qualitativa das perguntas e respostas postadas no *site* SOF. Eles consideraram quatro categorias de questões de acordo com as principais preocupações dos questionadores ao realizar uma pergunta no *site*: *Debug/Corrective*, *Need-To-Know*, *How-To-Do-It* e *Seeking-Different-Solution*. Além disso, eles analisaram respostas bem recebidas deste *site* e identificaram as características de trechos de código-fonte efetivos.

Eles descobriram que as explicações acompanhadas de trechos de código-fonte são tão importante quanto os próprios trechos de código-fonte. Ao contrário de Nasehi et al., o trabalho proposto classificou os pares Q&A ao invés de classificar somente a pergunta da *thread* de discussão. Foram consideradas inicialmente as seguintes categorias de pares Q&A: *How-to-do*, *Debug-corrective*, *Conceptual*, *Seeking-something* e *Miscellaneous*.

Em Treude et al. [48], eles analisaram dados do SOF para categorizar os tipos de perguntas que são respondidas e para explorar quais perguntas são bem respondidas pela comunidade e quais aquelas que permanecem sem ser respondidas. Suas descobertas preliminares indicaram que *sites* Q&A são particularmente efetivos para revisões de código e questões conceituais. Eles analisaram os textos dos títulos e corpo de 385 perguntas do SOF e encontraram as seguintes categorias, ordenadas pela sua frequência: *how-to*, *discrepancy*, *environment*, *error*, *decision help*, *conceptual*, *review*, *non-functional*, *novice* e *noise*. Enquanto o trabalho de Treude et al. propõe uma classificação manual, o presente trabalho propõe uma classificação automática. Neste quesito, um complementa o outro. Neste trabalho, foram analisados os textos do título, corpo da pergunta e corpo da resposta de 400 pares Q&A do SOF, sendo considerado inicialmente as seguintes categorias de pares Q&A: *How-to-do*, *Conceptual*, *Seeking-something*, *Debug-Corrective* e *Miscellaneous*.

3.7.2 Sistemas de Recomendação para Engenharia de *Software*

Vários trabalhos na literatura focaram no desenvolvimento de Sistemas de Recomendação para Engenharia de *Software* com o intuito de ajudar os desenvolvedores nas tarefas de desenvolvimento e recomendar trechos de código-fonte.

Ponzanelli et al. [40] apresentaram uma abordagem integrada e automatizada para ajudar os desenvolvedores que desejam aproveitar o "conhecimento da multidão" dos serviços Q&A. Eles implementaram SEAHAWK, um sistema de recomendação na forma de *plug-in* para Eclipse IDE que leva parte do conhecimento do SOF para dentro do IDE. Este *plug-in* automaticamente formula consultas a partir do contexto atual no IDE e apresenta uma lista ordenada e interativa dos resultados. SEAHAWK permite aos usuários identificar pedaços de discussão individuais e importar trechos de código-fonte através de um simples drag & drop. O *software* SEAHAWK [9] foi utilizado no desenvolvimento do *plugin* denominado *Nuggets Miner* [6]. A Seção 3.8 apresenta a arquitetura do *plugin* *Nuggets Miner*.

O trabalho proposto também recai na área de recuperação de trechos de código-fonte a partir da *Web*. Sawadsky et al. apresentaram FISHTAIL [43], 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. Este trabalho

[9] `http://seahawk.inf.usi.ch/download.html`

mostra como é possível obter trechos de código-fonte a partir do SOF. Como esses trechos foram obtidos a partir do *site* SOF, tais trechos de código-fonte já foram avaliados pela comunidade. Assim, o desenvolvedor não precisaria avaliar a sua validade.

Cordeiro et al. [11] 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 SOF. 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 [47], 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 SOF de forma a enriquecer a informação provida pelos trechos de código-fonte.

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 [24] 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. O trabalho proposto também está relacionado com o problema de Aprendizagem de API enfrentado diariamente pelos desenvolvedores. Ambos os trabalhos visam recomendar exemplos de uso da API para auxiliar os desenvolvedores durante as tarefas de desenvolvimento. Entretanto, o trabalho proposto não consulta repositórios de código-fonte. A recomendação do trabalho proposto é realizada com base nos pares Q&A do SOF referentes a uma determinada API.

Holmes et al. apresentaram DEEPINTELLISENCE [23], 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 HIPKAT [50], 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 SOF.

3.8 *Plugin Nuggets Miner*

Esta seção apresenta a arquitetura do *plugin* para Eclipse IDE. Este *plugin* foi desenvolvido como parte deste trabalho para ajudar os desenvolvedores durante as tarefas de desenvolvimento e na depuração de *"crowd bugs"*.

A Figura 3.8 mostra a arquitetura do *plugin Nuggets Miner*. A parte esquerda desta figura representa a parte servidora do *plugin*, enquanto a parte direita representa a parte cliente, ou seja, a interface gráfica e funcionalidades do *plugin*. O nome do *plugin Nuggets Miner* ("minerador de pepitas de ouro") é uma alusão à "Corrida do Ouro" na Califórnia (1848-1855). As pepitas representam a informação preciosa escondida e espalhada no "conhecimento da multidão" que poderá ajudar o desenvolvedor nos problemas de programação.

Na parte servidora (*Server side*), existe um componente para coleção e classificação dos dados, que é responsável por coletar e classificar os pares Q&A provenientes do SOF. As perguntas e respostas do SOF foram importadas a partir de um *data dump* público disponibilizado na forma de arquivos XML. Os dados foram extraídos através de um *XML dump importer* e armazenados em um banco de dados relacional por razões de desempenho. Após isso, foi utilizado o classificador *Logistic Regression* para encontrar quais são os pares Q&A da categoria *How-to-do* para o conjunto de APIs pré-selecionadas no estudo (a saber, *Swing* do Java, *Boost* do C++ e *LINQ* do C#).

Após este processo de classificação, foi construído um programa para a criação de arquivos XML no formato exigido pelo motor de busca *Apache Solr*. Após a criação desses arquivos XML, foi possível popular o índice de documentos do *Apache Solr*. Quando os documentos são indexados pelo motor de busca, eles tornam-se disponíveis para consulta. O *Apache Solr* provê uma *interface RESTful* para executar pesquisas por meio de solicitações *GET* e *POST*. Os documentos relevantes para a *query* podem ser retornados em formato XML ou *JSON (JavaScript Object Notation)* [10] (portanto, este índice pode ser consultado em qualquer linguagem de programação). No trabalho proposto, o *Apache Solr* foi configurado para apresentar a resposta em formato *JSON*.

A parte cliente (*Client side*) do *plugin Nuggets Miner* é responsável por consultar o índice *Solr*, deserializar a resposta *JSON* (converter a representação em objetos para ser possível sua manipulação). Além disso, esta parte possui dois motores de busca:

[10] `http://www.json.org/`

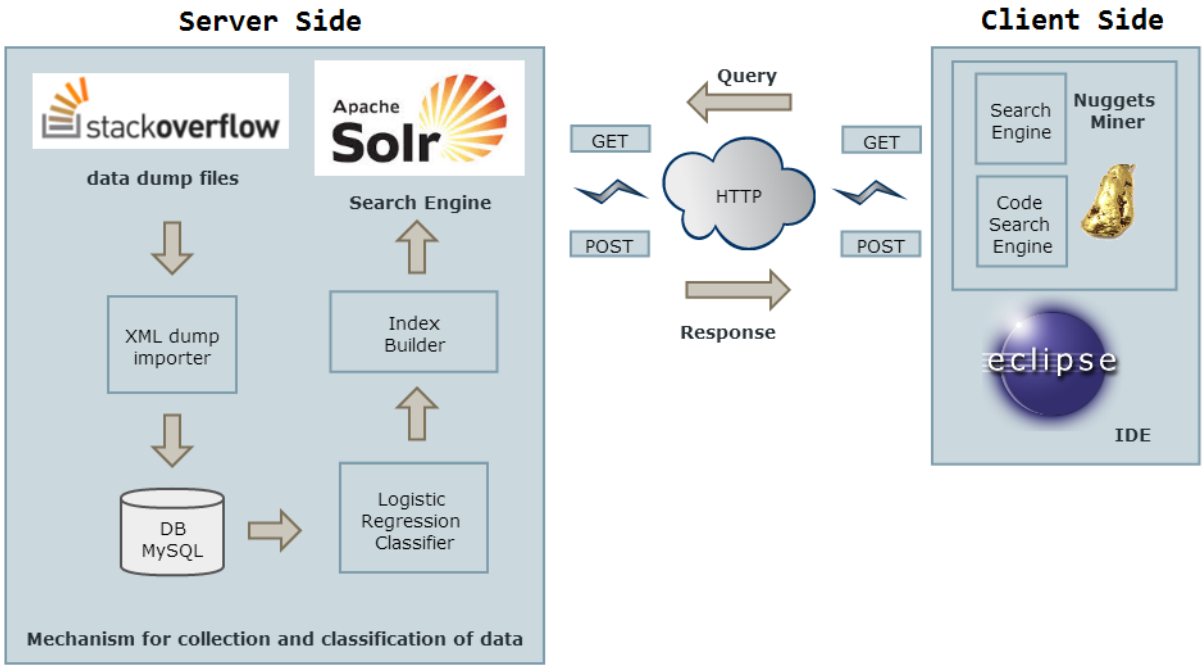

Figura 3.8: Arquitetura do *plugin Nuggets Miner*.

- *Search Engine*: é o motor de busca responsável por recuperar os pares Q&A mais similares com a consulta (formada por um conjunto de termos) para que o *Nuggets Miner* aplique a metodologia de ranqueamento dos pares Q&A e apresente para o usuário do sistema. Desta forma, o usuário poderá efetuar a leitura dos pares recomendados e arrastar trechos de código-fonte para dentro do editor de texto do programa via drag & drop;
- *Code Search Engine*: é o motor de busca responsável por recuperar os *posts* do SOF mais similares para um dado trecho de código de entrada (a consulta é um trecho de código) para que o *Nuggets Miner* apresente os resultados de maneira ranqueada ao usuário do sistema (os *posts* com trechos de código mais similares deverão aparecer primeiro no *ranking*).

Segundo Robillard et al., a interação com sistemas de recomendação pode ser tanto manual (i.e., a *query* é inserida pelo usuário) quanto automática (i.e., o motor de recomendação gera a *query*) [42]. *Nuggets Miner* suporta ambos: o usuário pode manualmente escrever consultas para recuperar os pares Q&A, ou *Nuggets Miner* pode extrair palavras-chave a partir de entidades de código, construir uma *query* e sugerir os pares Q&A.

3.9 Conclusões

Uma nova abordagem para aproveitar o "conhecimento da multidão" foi apresentada neste capítulo. Esta abordagem recomenda uma lista ranqueada de pares Q&A provenientes da base de conhecimento de desenvolvimento de *software* do SOF. Os critérios

de ranqueamento levam em conta a similaridade textual dos pares Q&A em relação ao problema do desenvolvedor, a qualidade desses pares e a caracterização *How-to-do* deles. Foram realizados experimentos considerando 35 tarefas de programação distribuídas em três diferentes APIs (*Swing*, *Boost* e *LINQ*) amplamente utilizados pela comunidade de desenvolvimento de *software*.

Uma análise manual qualitativa dos pares Q&A recomendados pelo sistema de recomendação foi realizada considerando dois critérios: **Relevância** e **Reprodutibilidade**. Foram obtidos os seguintes valores de NDCG: 0,3583 para o primeiro critério e 0,5243 para o segundo critério. Os resultados mostraram que em 27 das 35 atividades (77,14%), no mínimo um par recomendado provou ser útil para o problema de programação alvo. Além disso, para todas as 35 atividades, no mínimo um par recomendado possuía um trecho de código-fonte reprodutível ou quase reprodutível. Estes resultados sugerem que a abordagem proposta supera os resultados obtidos por Ponzanelli et al. [40].

Para disseminar o uso da abordagem proposta, um *plugin* para Eclipse IDE denominado *Nuggets Miner* [6] foi implementado.

Capítulo 4

Recomendação Baseada em *Snippets* para Auxílio a Correção de *"Crowd Bugs"*

4.1 Introdução

A atividade de depuração consiste em entender por que um trecho de código-fonte (do inglês, *code snippet*) não se comporta como o esperado. Os desenvolvedores podem depurar código que eles acabaram de escrever ou um trecho de código antigo para o qual um usuário tenha fornecido um relatório de *bug*. O processo de depuração de um determinado *bug* está fortemente relacionado com a natureza do *bug*. Por exemplo, depurar uma falha de segmentação consiste em focar em alocação e desalocação de memória, enquanto que depurar um erro de *NullPointerException* significa encontrar qual linha de código está acessando algum atributo ou operação de um objeto nulo.

Quando se trata de um *"crowd bug"*, é provável que este *bug* já tenha ocorrido várias vezes em diferentes domínios de aplicação e que, portanto, existe uma descrição deste problema em algum lugar da *Web* contendo sua explicação e correção. Em outras palavras, a multidão já identificou o *bug* e sua solução. Desta forma, esta solução pode ser recomendada para um desenvolvedor que esteja lidando com um determinado *"crowd bug"* e não sabe como resolvê-lo. A tarefa de fornecer um trecho de código-fonte para a multidão com o intuito de obter a sua correção foi denominada por Monperrus et al. [35] como "depuração com a multidão".

Portanto, a natureza dos *"crowd bugs"* (i.e., independentes de domínio de aplicação, ocorrem várias vezes com o mesmo sintoma) habilita um novo tipo de depuração (i.e., perguntar para a multidão). Este trabalho propõe uma abordagem de depuração que visa recomendar *posts* do SOF com o intuito de corrigir os diferentes *"crowd bugs"* presentes nas linguagens de programação Java e JavaScript.

As contribuições apresentadas nesta seção são:

- Análise empírica utilizando trechos de código-fonte do SOF pertencentes às linguagens de programação Java e JavaScript que mostrou que o motor de busca do SOF não é adequado quando se usa código como consulta de entrada;
- Construção de índices específicos do *Apache Solr* [1] contendo informações léxicas e sintáticas de trechos de código-fonte pertencentes às linguagens de programação Java e JavaScript com o intuito de comparar os resultados obtidos com o índice nativo do SOF;
- Construção de um *dataset* de *"crowd bugs"* contendo 30 trechos de código-fonte retirados do OHLOH [2] (sendo 15 da linguagem Java e 15 da linguagem JavaScript). Cada trecho de código foi mapeado com o seu respectivo endereço no OHLOH e com um endereço de *post* no SOF que corrige o possível *"crowd bug"* presente no trecho de código;
- Análise qualitativa e quantitativa do ranqueamento geral realizado pelo sistema de recomendação para os *"crowd bugs"* das linguagens Java e JavaScript presentes no *dataset*. Para cada *"crowd bug"* pesquisado no sistema de recomendação, é fornecido um *ranking* contendo os Top-30 *posts* do SOF mais similares com o *"crowd bug"* de entrada. É desejável que o *post* do SOF forneça uma correção para o *"crowd bug"*.
- Desenvolvimento de um sistema de recomendação que possibilita consultar trechos de código-fonte em um índice do *Apache Solr* previamente construído.

O restante deste capítulo é organizado da seguinte maneira. A Seção 4.2 revisa o conceito de *"crowd bugs"* definido por Monperrus et al. [35]. A Seção 4.3 apresenta o escopo do estudo (perguntas de pesquisa, APIs consideradas, banco de dados utilizado e as funções de pré-processamento de trechos de código utilizadas). A Seção 4.4 apresenta os experimentos realizados com os trechos de código do SOF considerando dois motores de busca diferentes: o motor de busca do SOF e o motor de busca do *Apache Solr*. A Seção 4.5 apresenta a metodologia utilizada na construção do *dataset* de *"crowd bugs"*. A Seção 4.6 mostra os experimentos realizados com trechos de código encontrados em projetos de *software* hospedados no OHLOH que são candidatos a conter *"crowd bugs"*, uma vez que esses trechos contém chamadas de métodos de API que geram saída incorreta dependendo da maneira de como eles são utilizados pelo desenvolvedor. A Seção 4.7 discute os resultados obtidos nos experimentos com *"crowd bugs"* deste *dataset*. A Seção 4.9 apresenta dois trabalhos relacionados na área e as conclusões do estudo.

[1] http://lucene.apache.org/solr/
[2] https://code.ohloh.net/

4.2 *"Crowd Bugs"*

Para definir o que é um *"crowd bug"*, sejam inicialmente dois 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 de um ângulo. Apesar desta descrição aparentemente simples e do nome auto-descrito, este método coloca um problema para muitos desenvolvedores, como testemunhado por dezenas de Q&As neste tópico no *site* SOF [3] (isto é provavelmente uma subestimação, pois acredita-se que muitos desenvolvedores procuram o problema em máquinas de busca de propósito geral como o *Google* antes de fazer uma pergunta em uma lista de discussão ou em um *site* Q&A). Várias Q&As estão relacionadas com o mesmo problema: *Por que Math.cos() produz resultado incorreto?*

A resposta é que se o argumento do método **Math.cos** for passado em graus ao invés de radianos, o método produz resultado incorreto. Por que a mesma pergunta é realizada repetidas vezes? A hipótese é que a semântica do método **Math.cos** é contra-intuitiva para a maioria dos desenvolvedores (i.e., eles assumem erroneamente que o ângulo deve ser passado em graus), e consequentemente, o mesmo problema ocorre em muitas situações de desenvolvimento, independentemente do domínio da aplicação. A correção para este problema é utilizar o método **Math.toRadians** para converter primeiro o ângulo em radianos, para só depois invocar o método **Math.cos** sobre o argumento convertido, ou seja, "Math.cos(Math.toRadians(angle))".

Seja um segundo exemplo. Em *JavaScript*, existe uma função denominada **parseInt**, que analisa uma determinada *string* de entrada e retorna o correspondente valor inteiro. Apesar desta descrição aparentemente simples, esta função também apresenta problemas para muitos desenvolvedores, como testemunhado por dezenas de Q&As neste tópico no *site* SOF [4]. Muitas perguntas estão relacionadas ao mesmo problema: "Por que *parseInt*("08") produz "0" e não "8" ?

A resposta é que o argumento da função **parseInt** começa com "0", sendo processada como octal (ou seja, no sistema de numeração cuja base é 8). Por que esta pergunta é realizada repetidas vezes? De acordo com Monperrus et al. [35], uma possível explicação para isto é que a semântica da função **parseInt** é contra-intuitiva para a maior parte dos desenvolvedores (i.e., eles assumem erroneamente que a base numérica é 10 ao invés de 8, talvez porque o sistema decimal numérico é mais frequentemente utilizado em operações gerais envolvendo números), e consequentemente, o mesmo problema ocorre em várias situações de desenvolvimento, independente do domínio da aplicação. A correção para este problema é informar explicitamente a base numérica, ou seja, "parseInt("08", 10)".

[3] http://stackoverflow.com/search?q=[java]+Math.cos
[4] http://stackoverflow.com/search?q=[javascript]+title:parseInt

Neste trabalho será utilizada a mesma definição de *"crowd bugs"* proposta por Monperrus et al. [35]:

Definição: Um "crowd bug" é um bug que causa uma saída ou comportamento inesperado e incorreto resultante do uso comum e intuitivo de uma API.

Esta classe de *bugs* independe do domínio da aplicação. Portanto, um mesmo *"crowd bug"* pode ocorrer em diferentes projetos e em diferentes domínios de aplicação. É importante observar que *"crowd bugs"* são o oposto de *bugs* específicos de projeto. Estes, por sua vez, são resultado de uma implementação incorreta das regras de negócio.

É importante destacar que *"crowd bugs"* não são *bugs* nas funções ou métodos de uma API. No presente estudo, considera-se que a API não contém *bugs*. Os *"crowd bugs"* ocorrem devido a um uso incorreto das funções ou métodos da API por parte do desenvolvedor.

Quando se fala em *"crowd bugs"*, uma pergunta imediata vem à tona: *Como os "crowd bugs" aparecem?* Eles aparecem quando um desenvolvedor de uma API realiza decisões de *design* que vão contra a intuição e a expectativa da maioria dos desenvolvedores. No exemplo da função **parseInt**, a maioria dos desenvolvedores esperam que **parseInt("08")** retorne "8".

Duas razões principais podem explicar a origem dos *"crowd bugs"*:

- Primeiro, a API em questão pode parecer compreensível o suficiente por parte dos desenvolvedores, que por sua vez, não realizam a leitura da documentação oficial;
- Segundo, a API pode assumir alguma coisa implicitamente. Por exemplo, a função **parseInt** implicitamente assume que prefixar a *string* com "0" significa que a base numérica octal (base 8) foi escolhida. Ao contrário dos desenvolvedores, que acreditam que a base numérica utilizada pela função é a decimal (base 10).

Um estudo mais detalhado sobre a causa do aparecimento dos *"crowd bugs"* está fora do escopo deste trabalho. Além disso, este estudo requer um trabalho inter-disciplinar entre os diferentes campos como Engenharia de *Software* e Psicologia.

Outra pergunta recorrente é: *Como corrigir os "crowd bugs"?* Para corrigir este tipo de *bug*, o desenvolvedor primeiramente precisa ter certeza que o comportamento incorreto observado não é específico do domínio da aplicação (i.e., não está relacionado a nenhuma regra de negócio). A forma comum e sensata de corrigir o *bug* é pesquisar por problemas similares através da *Internet*. Uma vez que os *"crowd bugs"* não estão relacionados a um conhecimento de domínio específico, eles podem ser depurados pela própria multidão. A descrição do sintoma é o suficiente para outros desenvolvedores recordarem a ocorrência do mesmo *bug* e fornecer a correção. Para corrigir um *"crowd bug"*, pergunte para a multidão. [34].

Por exemplo, no *site* SOF, todas as perguntas relacionadas com a característica não-intuitiva da função **parseInt** são respondidas. A resposta mais votada [5] foi dada somente um minuto depois que a pergunta foi realizada no *site*.

"Crowd bugs" são pouco estudados na literatura. Somente Carzaniga et al. [8] provaram que a *Web* contém algumas descrições de *"crowd bugs"*. Monperrus et al. [35] mostraram com o exemplo da função **parseInt** que o SOF também contém este tipo de *bugs*. Neste trabalho, foi construído um *dataset* com 30 *"crowd bugs"* das linguagens Java e JavaScript confirmando que o SOF está repleto de *posts* que contém descrições de como corrigir estes *"crowd bugs"*.

4.3 Escopo do Estudo

Para propor um motor de busca que recebe trechos de código-fonte como entrada (i.e., *"crowd bugs"*), foram feitas algumas perguntas de pesquisa (Q1, Q2, Q3 e Q4). Além disso, foram definidas as tecnologias sobre as quais o sistema de recomendação foi proposto (i.e., Java Android, Java não-Android e JavaScript). A Subseção 4.3.1 apresenta quais são essas perguntas de pesquisa. Na Subseção 4.3.2, são apresentadas as funções de pré-processamento de trechos de código-fonte adotadas no estudo. A Subseção 4.3.3 explica como foi construído o banco de dados local com os dados do SOF.

4.3.1 Perguntas de Pesquisa

Esta subseção apresenta as 4 perguntas de pesquisa consideradas nesta parte do estudo.

Q1: Qual é a eficácia do SOF em resposta a consultas que são trechos de código-fonte?

A pergunta **Q1** pretende investigar se o motor de busca do SOF consegue lidar bem com consultas baseadas em código, que são muitas vezes realizadas pelos desenvolvedores de *software* durante as tarefas de programação. Para responder a essa pergunta de pesquisa, é necessário realizar o **Experimento I**, que consiste em realizar consultas no motor de busca do SOF utilizando trechos de código do próprio SOF. Trata-se de um experimento de *loopback*, ou seja, dado que um trecho de código está presente em algum *post* do SOF e este mesmo trecho de código é utilizado como entrada para o motor de busca nativo do SOF, pretende-se saber se o SOF encontra o respectivo *post* que contém o trecho de código consultado.

[5] http://stackoverflow.com/questions/850341/how-do-i-work-around-javascripts-parseint-octal-behavior

Q2: Qual mecanismo de consulta por trechos de código-fonte é mais eficaz para as tecnologias consideradas neste estudo: pesquisar em um índice do *Apache Solr* previamente construído ou pesquisar no índice nativo do SOF?

A pergunta de pesquisa **Q2** pretende investigar se a construção de índices do *Apache Solr* que contém informações de trechos de código melhoram as consultas baseadas em código, muitas vezes realizadas pelos desenvolvedores de *software* durante as tarefas de programação. Para responder a essa pergunta de pesquisa, é necessário realizar o **Experimento I** e comparar os resultados com o **Experimento II**. Dessa forma, é possível saber qual dos mecanismos de consulta por trechos de código é o mais eficaz para as tecnologias consideradas neste trabalho (i.e., Java Android, Java não-Android e JavaScript).

Q3: Com qual extensão as funções de pré-processamento de trechos de código-fonte melhoram a eficácia na consulta por estes trechos?

A pergunta de pesquisa **Q3** pretende investigar com qual extensão a utilização de funções de pré-processamento de trechos de código para indexação e consulta melhora a eficácia da consulta baseada em código. Para responder a essa pergunta de pesquisa, é necessário realizar o **Experimento II**, pois neste experimento foram construídos vários índices do *Apache Solr*, alguns sem utilizar nenhuma função de pré-processamento e outros utilizando algumas funções de pré-processamento de trechos de código.

Q4: Dado que o SOF possui cobertura dos *"crowd bugs"*, como está o ranqueamento feito pelo sistema de recomendação proposto utilizando *"crowd bugs"* pertencentes a projetos reais de *software*?

A pergunta de pesquisa **Q4** pretende investigar como o sistema de recomendação proposto lida com *"crowd bugs"* pertencentes a projetos reais de *software*, ou seja, verificar como está o ranqueamento feito pelo sistema de recomendação proposto (e.g., porcentagem de *posts* do SOF que corrigem *"crowd bugs"* encontrados no top-15). Para responder a essa pergunta de pesquisa, é necessário realizar o **Experimento III**, pois este experimento é dedicado para investigar como está a recomendação para correção dos *"crowd bugs"*, utilizando-se trechos de código extraídos do *OHLOH* [6]. Já os experimentos I e II utilizam os trechos de código presentes no SOF.

4.3.2 Funções de Pré-processamento de Trechos de Código

Para responder se o pré-processamento de trechos de código nas etapas de indexação dos dados e consulta dos mesmos melhora a eficácia da consulta (i.e., pergunta de pesquisa

[6] `https://code.openhub.net/`

Q3), é necessário definir algumas funções de pré-processamento e construir índices específicos do *Apache Solr* utilizando estas funções. O trabalho proposto investiga o potencial de 7 funções de pré-processamento (apresentadas abaixo) para as 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 (e.g., 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 adotadas no presente estudo.

Função *"raw"*

Esta função não modifica o conteúdo do trecho de código-fonte, i.e., não realiza nenhum pré-processamento neste código.

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.

Função Sintática para Java (*pp2*)

Esta função considera elementos da gramática livre de contexto da linguagem: processa o trecho de código em 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 [7].

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 em JavaScript utilizando a API *Rhino* [8].

Partial Program Analysis para Java (*ppa*)

Partial Program Analysis (PPA) [9] [13] é um *framework* de análise estática que transforma o código-fonte de um programa Java incompleto em uma árvore de sintaxe abstrata tipada, resolvendo as construções sintáticas ambíguas. Este *framework* possui muitas vantagens em relação à função sintática **pp2**, dentre elas:

[7] http://www.eclipse.org/jdt/core/
[8] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[9] http://www.sable.mcgill.ca/ppa/

- **ppa** resolve construções de sintaxe ambíguas;
- **ppa** infere informação inexistente de tipo quando possível;
- **ppa** permite lidar com código incompleto em Java.

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

Esta função processa o trecho de código Java e extrai as respectivas chamadas de métodos. As características deste *parser* são:

- *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 (e.g., **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 (e.g., **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**. Optou-se por indexar somente o nome do método (ou seja, o que aparece após o ".") para cobrir os casos em que não é possível resolver o *binding* (alguns trechos de código do SOF são incompletos e não possuem o comando de declaração da variável).

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 (e.g., a palavra *"function"* é uma palavra reservada da linguagem JavaScript, enquanto que 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"* (e.g., **function nomeDaFunção()**). Neste estudo, não é indexado os nomes de função de domínio específico do problema, uma vez que o objeto alvo do estudo são os métodos das bibliotecas. Estes, por sua vez, tem uma alta probabilidade de serem encontrados no conhecimento da multidão do SOF, dado que são independentes do domínio da aplicação.

A Tabela 4.1 mostra as funções de pré-processamento de trechos de código-fonte para as linguagens de programação consideradas neste estudo.

Tabela 4.1: Funções de pré-processamento de trechos de código-fonte para as linguagem de programação consideradas.

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

4.3.3 Banco de Dados Local do SOF

Foi obtida uma versão do *data dump* [10] público do SOF (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*.

4.4 Experimentos com os Trechos de Código do SOF

Foram realizados experimentos com os motores de busca do SOF e do *Apache Solr* para responder as questões de pesquisa (**Q1**, **Q2** e **Q3**). A Subseção 4.4.1 tem como objetivo responder a pergunta de pesquisa **Q1**, i.e., responder se o motor de busca nativo do SOF possui boa eficácia ao lidar com consultas que são trechos de código-fonte.

Antes de detalhar quais experimentos foram realizados, apresenta-se brevemente o que é indexação e como isso é importante no trabalho proposto. A indexação fornece aos usuários uma experiência de pesquisa em tempo real sobre um conjunto de documentos que englobam milhões de perguntas e respostas. A técnica padrão de pesquisa baseada em indexação consiste de duas funções de transformação t_{index} e t_{query}. A primeira função t_{index} transforma os documentos (i.e., Q&As) em termos e a segunda função t_{query} transforma a consulta (i.e., *query*) também em termos. O ranqueamento consiste em combinar os termos do documento contra os termos da consulta, possivelmente com algumas estratégias de peso. As funções t_{index} e t_{query} não necessariamente são idênticas, uma vez que a natureza dos documentos (estilo, estrutura, etc.) é frequentemente diferente da natureza das consultas.

4.4.1 Experimento I: Consultas Envolvendo o Índice Nativo do SOF

O objetivo consiste em responder se o motor de busca do SOF é capaz de lidar bem com trechos de código-fonte como consulta de entrada. A idéia foi alimentar o motor de busca do SOF com trechos de código que já existem no SOF.

Considera-se que um *site* Q&A lida bem com um trecho de código se o *post* que contém

[10] http://blog.stackoverflow.com/category/cc-wiki-dump/

tal código está ranqueado nas primeiras posições dos resultados do motor de busca. Isto corresponde ao caso no qual o desenvolvedor pesquisa um *"crowd bug"* e obtém a resposta correta (i.e., o *post* no SOF que corrige o *"crowd bug"* pesquisado).

Nos casos em que um *site* Q&A não encontra os *posts* com as soluções dos *"crowd bugs"*, é dito que este *site* não responde bem às consultas que são trechos de código. Isto é um indício de que uma técnica especial de indexação necessita ser desenvolvida para obter melhores resultados de busca.

Para responder a pergunta de pesquisa **Q1**, o seguinte processo experimental foi realizado em 4 passos:

- **Passo 1**: Foram selecionados aleatoriamente 1000 trechos de código do SOF para cada um dos tópicos considerados neste estudo (i.e., Java Android, Java não-Android e JavaScript);
- **Passo 2**: Para cada trecho de código selecionado, foi mapeado o respectivo *post* do SOF que contém este trecho de código;
- **Passo 3**: Para cada trecho de código selecionado, foram realizadas 2 consultas diferentes no *site* do SOF: uma consulta considerando o trecho de código no seu estado natural (i.e., *"raw"*) e outra consulta considerando o resultado gerado pela função de pré-processamento *pp1* no trecho de código;
- **Passo 4**: Para cada trecho de código consultado, foi calculada em qual posição no Top-50 o *post* do SOF que foi mapeado para este trecho de código no **Passo 2** aparece.

A Tabela 4.2 mostra os resultados obtidos no Experimento I.

Tabela 4.2: Utilizando trechos de código do SOF como consulta de entrada (t_{index}=SOF significa que o índice utilizado foi o índice nativo do SOF; NE = *Post* do SOF Não Encontrado.)

Tópico	Consulta	NE	Rank #1	Rank ≤ 5	Rank ≤ 10	Rank ≤ 50
Java Android	t_{index}=SOF, t_{query}=raw	893	89	98	99	107
Java Android	t_{index}=SOF, t_{query}=pp1	947	47	49	52	53
Java não-Android	t_{index}=SOF, t_{query}=raw	904	87	94	94	96
Java não-Android	t_{index}=SOF, t_{query}=pp1	891	105	105	106	109
JavaScript	t_{index}=SOF, t_{query}=raw	950	45	49	50	50
JavaScript	t_{index}=SOF, t_{query}=pp1	860	127	138	139	140

Resposta à Q1: Qual é a eficácia do SOF em resposta a consultas que são trechos de código-fonte? Os resultados do Experimento I mostraram que o SOF não possui boa eficácia ao tratar trechos de código como consulta. Além disso, este experimento confirma os resultados obtidos por Monperrus et al. [35]. Os resultados comprovam que o SOF usam diferentes t_{index} e t_{query}. Isto não configura um *bug*. Estes resultados apenas mostram que o SOF não foi designado para tratar consultas baseadas em código. Isto conduz a: 1) melhorar a consulta baseada em trechos de código utilizando funções de pré-processamento destes trechos; 2) construir vários índices dedicados contendo informações destes trechos de código do SOF para investigar se esses índices renderiam melhores resultados do que o índice nativo do SOF e quais desses índices eram mais adequados para a consulta de trechos de código. Todos estes índices foram construídos utilizando o motor de busca do *Apache Solr*.

4.4.2 Experimento II: Consultas Envolvendo os Índices *raw*, *pp1*, *pp2*, *pp3*, *ppa*, *lsab-java* e *lsab-js* do *Apache Solr*

Para responder às questões de pesquisa **Q2**, **Q3** e **Q4**, foram construídos 9 índices do *Apache Solr* no total. São eles:

- **Java-raw**: composto por todos os trechos de código Java do banco de dados local sem nenhum pré-processamento;
- **Java-pp1**: composto por informações de todos os trechos de código Java do banco de dados local processados com a função léxica *pp1*;
- **Java-pp2**: composto por informações de todos os trechos de código Java do banco de dados local processados com a função sintática *pp2*;
- **Java-ppa**: composto por informações de todos os trechos de código Java do banco de dados local processados com o *framework* PPA (*Partial Program Analysis for Java*).
- **Java-lsab**: composto por informações de todos os trechos de código Java do banco de dados local processados com o *parser Lightweight Syntactic Analysis and Binding for Java*.
- **JavaScript-raw**: composto por todos os trechos de código JavaScript do banco de dados local sem nenhum pré-processamento;
- **JavaScript-pp1**: composto por informações de todos os trechos de código JavaScript do banco de dados local processados com a função léxica *pp1*;
- **JavaScript-pp3**: composto por informações de todos os trechos de código JavaScript do banco de dados local processados com a função sintática *pp3*;

- **JavaScript-lsab**: composto por informações de todos os trechos de código JavaScript do banco de dados local processados com o *parser Lightweight Syntactic Analysis and Binding for JavaScript*.

Como dito anteriormente no Experimento I, para cada tópico (i.e., Java Android, Java não-Android e JavaScript) foram selecionados aleatoriamente 1000 trechos de código do SOF. Tais trechos foram utilizados para a realização do Experimento II. Assim, para cada trecho de código Java, foi feito uma consulta em cada um dos 5 índices do *Apache Solr* (a saber: **Java-raw**, **Java-pp1**, **Java-pp2**, **Java-ppa** e **Java-lsab**). No caso em que o trecho de código for escrito em JavaScript, foi feito uma consulta em cada um dos 4 índices do *Apache Solr* (a saber: **JavaScript-raw**, **JavaScript-pp1**, **JavaScript-pp3** e **JavaScript-lsab**).

Por exemplo: dado um trecho de código em Java Android ou Java não-Android, tal código é processado utilizando cada uma das funções de pré-processamento para código Java (i.e., "raw", "pp1", "pp2", "ppa", "lsab-java"). Após o pré-processamento do código por uma das funções de pré-processamento, o resultado é consultado no respectivo índice do *Apache Solr* (i.e., código processado com a função **pp1** só consulta o índice **Java-pp1**, código processado com a função **pp2** só consulta o índice **Java-pp2**, e assim por diante). Se o trecho de código for escrito em JavaScript, tal código é processado utilizando cada uma das funções de pré-processamento para código JavaScript (i.e., "raw", "pp1", "pp3" e "lsab-js"). Após o pré-processamento do código por uma das funções de pré-processamento, o resultado é consultado no respectivo índice do *Apache Solr*.

O motor de busca do *Apache Solr* produz um ranqueamento contendo os *posts* do SOF que possuem maior similaridade textual com o código pesquisado. Como se conhece o *post* do SOF que contém o código consultado, é possível saber, utilizando o *id* do *post*, em qual posição do *ranking* ele aparece nos resultados de busca do *Apache Solr*. Existem casos em que o *post* não é encontrado no *ranking*.

Antes de mostrar os resultados obtidos no Experimento II, é indispensável entender melhor como foi o processo de construção do índice **Java-ppa** do *Apache Solr*. A construção deste índice foi demorada (aproximadamente 11 dias = 264 horas para criar todo o índice), devido ao processo interno do **ppa** para a geração dos fatos sintáticos. A Subseção 4.4.2 detalha todo o processo de construção adotado para a construção deste índice.

Processo de construção do índice *Java-ppa* do *Apache Solr*

Subramanian et al. [46] analisaram 39.000 trechos de código do SOF e observaram que 6.766 (17%) eram arquivos completos, i.e., contendo a classe e as declarações de métodos, 6.302 (16%) eram trechos de código que continham apenas o corpo do método desprovidos da declaração da classe. Os restantes (66%) continham instruções de código-fonte soltas. Eles utilizaram o *framework* **ppa** para lidar com informação incompleta destes trechos de

código.

Para utilizar o **ppa** adequadamente, os seguintes passos foram realizados:

- **Passo 1**: Todos os *posts* do SOF considerados na construção do índice **Java-ppa** foram filtrados utilizando a *tag* "<java>";
- **Passo 2**: Para cada trecho de código presente no *post* do SOF, foi criado um arquivo com extensão ".java" contendo o código. Para os trechos que eram desprovidos de classe, o conteúdo deles foi colocado dentro de uma classe padrão definida (i.e., **public class NoName**);
- **Passo 3**: Cada um desses arquivos foi passado como entrada para o **ppa**. A partir deles, foram extraídos os fatos sintáticos (i.e., informações de *bindings* de métodos e tipos de variáveis);
- **Passo 4**: Para cada *post* processado com sucesso pelo **ppa** (i.e., composto por n trechos de código cujos fatos sintáticos foram extraídos com sucesso pelo **ppa**), foi criado um documento no índice do *Apache Solr* contendo o título e o *id* do *post* do SOF, juntamente com os respectivos fatos sintáticos gerados pelo **ppa**.

A Tabela 4.3 mostra os resultados obtidos após o processamento do **ppa** em todos os trechos de código Java do banco de dados local. Foram gerados 1.123.363 arquivos com extensão ".java". Deste total, 1.066.170 (94,9087%), foram processados com sucesso pelo **ppa**, enquanto que 57.193 (5,0912%) foram processados com erro pelo **ppa**. Estes erros estão associados com a falta de unidade de compilação em alguns trechos de código Java.

Tabela 4.3: Resultados após o processamento do *ppa*.

Total de Arquivos Gerados	Qtd. Sucesso *ppa*	Qtd. Erro *ppa*
1.123.363	1.066.170	57.193

Para cada um dos 2000 trechos de código-fonte do SOF utilizados no Experimento I (i.e., 1000 para Java Android e 1000 para Java não-Android), foi criado um arquivo para ser processado como entrada para o **ppa**. Após a extração dos fatos **ppa** sobre um determinado arquivo, tais fatos foram pesquisados no índice do *Apache Solr*. A partir daí, foi realizado o processo de identificar em qual posição no *ranking* gerado pelo motor de busca *Apache Solr* aparece o o *id* do *post* do SOF que contém o trecho de código pesquisado.

A Tabela 4.4 mostra os resultados obtidos no Experimento II. A coluna **NE** desta tabela mostra o total de resultados não encontrados durante a consulta nestes índices. A coluna **Erro** desta tabela mostra a quantidade de erros ao executar uma determinada função de pré-processamento sobre um trecho de código-fonte. Por exemplo, para a função de pré-processamento **ppa**, ocorreram 226 casos de erro ao tentar executar alguns trechos de código do tópico "Java Android". O motivo dos erros era o fato do código-fonte não

possuir unidade de compilação. Desta forma, não era possível gerar a Árvore Sintática Abstrata (AST) a partir do trecho de código-fonte, inviabilizando a extração de fatos sintáticos.

Tabela 4.4: Utilizando trechos de código do SOF como consulta (SOLR refere-se ao *Apache Solr*, portanto t_{index}=SOLR significa que o índice de busca foi construído utilizando o *Apache Solr*; NE = Não Encontrado.)

Tópico	Consulta	NE	Erro	Rank#1	Rank≤5	Rank≤10	Rank≤50
Java Android	t_{index}=SOLR$_{raw}$, t_{query}=raw	151	0	619	740	785	849
Java Android	t_{index}=SOLR$_{pp1}$, t_{query}=pp1	31	0	857	942	950	969
Java Android	t_{index}=SOLR$_{pp2}$, t_{query}=pp2	111	0	713	790	817	889
Java Android	t_{index}=SOLR$_{ppa}$, t_{query}=ppa	61	226	634	683	693	713
Java Android	t_{index}=SOLR$_{lsab}$, t_{query}=lsab-java	27	233	599	682	695	740
Java não-Android	t_{index}=SOLR$_{raw}$, t_{query}=raw	112	0	701	824	854	888
Java não-Android	t_{index}=SOLR$_{pp1}$, t_{query}=pp1	35	0	900	950	959	965
Java não-Android	t_{index}=SOLR$_{pp2}$, t_{query}=pp2	170	0	638	739	765	830
Java não-Android	t_{index}=SOLR$_{ppa}$, t_{query}=ppa	127	76	700	758	768	797
Java não-Android	t_{index}=SOLR$_{lsab}$, t_{query}=lsab-java	20	79	798	869	887	901
JavaScript	t_{index}=SOLR$_{raw}$, t_{query}=raw	73	0	743	887	904	927
JavaScript	t_{index}=SOLR$_{pp1}$, t_{query}=pp1	9	0	905	981	988	991
JavaScript	t_{index}=SOLR$_{pp3}$, t_{query}=pp3	119	0	775	856	868	881
JavaScript	t_{index}=SOLR$_{lsab}$, t_{query}=lsab-js	110	129	516	583	618	761

Resposta à Q2: Qual mecanismo de consulta por trechos de código-fonte é mais eficaz para as tecnologias consideradas neste estudo: pesquisar em um índice do *Apache Solr* previamente construído ou pesquisar no índice nativo do SOF? Os resultados do Experimento II mostram que pesquisar em índices do *Apache Solr* previamente construídos é muito mais eficiente do que pesquisar no índice nativo do SOF. A coluna **NE** da Tabela 4.4 mostrou que em todos os índices, o número de resultados encontrados foi ≤ 170. Além disso, a coluna Rank ≤ 10 da Tabela 4.4 mostrou o alto aproveitamento de todos os índices na consulta por trechos de código (≥ 618).

Resposta à Q3: Com qual extensão as funções de pré-processamento de trechos de código-fonte melhoram a eficácia na consulta por estes trechos? O Experimento II mostrou que existem funções de pré-processamento de código que possuem eficácia muito maior do que outras. Por exemplo, no Experimento II a função léxica

pp1 obteve mais de 95% no Top-10 em todas as tecnologias ("Java Android", "Java não-Android" e "JavaScript"), obtendo assim, uma menor quantidade de erros e de resultados não encontrados em relação às outras funções de pré-processamento. A coluna **NE** da Tabela 4.4 mostrou que em alguns casos, as funções **pp2** e **pp3** tiveram uma quantidade de resultados não encontrados maior do que a função **raw** (i.e., para as tecnologias "Java não-Android" e "JavaScript"). A função **ppa** do Experimento II obteve resultados bons. Porém a coluna **Erro** da Tabela 4.4 aponta uma alta quantidade de erros para "Java Android" (22,6%), mostrando que o **ppa** lidou melhor com trechos de código pertencentes ao tópico "Java não-Android". A função de pré-processamento **lsab-java** foi muito melhor no tópico "Java não-Android" do que no tópico "Java Android". Uma possível explicação para isso é que a tecnologia "Java Android" não utiliza apenas a linguagem Java para criação da interface gráfica com o usuário, mas também faz uso da linguagem de marcação XML. Todavia, o *parser* **lsab-java** não foi designado para processamento de XML.

4.5 Construção do *Dataset* de *"Crowd Bugs"*

Com o objetivo de realizar experimentos com *"crowd bugs"* presentes em diferentes projetos de *software* do mundo real, foi necessário construir um *dataset* de *"crowd bugs"*.

Os seguintes passos foram realizados até a construção do *dataset*:

- **1º Passo**: Foi pesquisado no motor de busca do SOF as seguintes expressões: *"unexpected output"* e *"incorrect output"* com o intuito de filtrar apenas os *posts* do SOF candidatos a serem *"crowd bugs"*;
- **2º Passo**: Após a consulta, foi feita uma análise qualitativa que resultou na identificação de 30 *"crowd bugs"*, sendo 15 da linguagem Java e 15 da linguagem JavaScript;
- **3º Passo**: Visando utilizar trechos de código presentes em projetos reais de *software*, cada um desses *"crowd bugs"* do SOF foi pesquisado no *site OHLOH* [11] . Desta forma, foram encontrados 30 trechos de código no *OHLOH* que utilizavam os mesmos métodos da API presentes nestes 30 *"crowd bugs"* do SOF;
- **4º Passo**: Para cada *"crowd bug"* do *dataset*, foi mapeado dois endereços *Web*: o endereço do *OHLOH* que supostamente conteria o *"crowd bug"* (cabe ressaltar que o trecho de código do *OHLOH* não tem o *"crowd bug"*, mas seria um código potencial para ter o *bug* pois contém a chamada de método da API que pode levar ao *"crowd bug"*) e o endereço de um *post* no SOF que corrige este *"crowd bug"* (OBS.: pode existir mais de um *post* no SOF que corrige o *"crowd bug"*, o que aumenta a probabilidade de correção dos mesmos).

[11] https://code.ohloh.net/

O *dataset* de *"crowd bugs"* está disponível em: [12]. A Seção 4.6 apresenta com detalhes os resultados obtidos no experimento com os *"crowd bugs"* do *dataset*.

4.6 Experimento III: Consultas Envolvendo os *"Crowd Bugs"* do *Dataset*

Para analisar a recomendação feita pela abordagem proposta, foram feitas consultas com os *"crowd bugs"* presentes no *dataset*. Assim, foi criado um programa para exibir para cada *"crowd bug"*, a posição no *ranking* em que o *post* do SOF mapeado para ele aparece. Esta é uma forma de avaliar a qualidade da recomendação apresentada. Como dito anteriormente, todos os *"crowd bugs"* do *dataset* proposto estão mapeados com um *post* do SOF que o corrige.

Além disso, este experimento irá nos permitir responder à pergunta de pesquisa **Q4**. O programa criado para conduzir o experimento realiza diferentes consultas nos índices do *Apache Solr*, sendo que para cada trecho de código pesquisado, são analisados os Top-100 *posts* do SOF retornados pelo motor de busca para descobrir em qual posição do *ranking* aparece o *post* do SOF que corrige o *"crowd bug"*. A entrada para o programa são trechos de programas que supostamente poderiam conter *"crowd bugs"*. A saída do programa é um arquivo contendo as posições no *ranking* ocupadas pelos *posts* mapeados do SOF.

Antes de explicar como foram feitas as consultas dos *"crowd bugs"* do *dataset* nos índices do motor de busca *Apache Solr*, é necessário entender sobre o filtro de palavras-chave que foi utilizado como parte da *query* deste motor de busca.

A Tabela 4.5 mostra as palavras-chave que foram utilizadas neste filtro. Tais palavras aparecem frequentemente nos títulos dos *posts* do SOF que contém correções de *"crowd bugs"*. O filtro é adicionado na *query* do *Apache Solr* (i.e., no momento em que é feito a consulta no índice). Estas palavras foram retiradas a partir de uma análise qualitativa envolvendo 70 *posts* do SOF que contém descrições de *"crowd bugs"*.

Tabela 4.5: Palavras-chave consideradas como filtro na *query* do *Apache Solr*.

Palavras-chave	unexpected, incorrect, wrong, output, return, returns, result, results, behavior, weird, strange, odd, problem, problems

As Figuras 4.1 e 4.2 mostram para um dado trecho de código Java presente no *dataset* (trecho de código número 15), como são construídas as consultas do *Apache Solr*. A Figura 4.1 mostra o uso do filtro de palavras-chave, enquanto que a Figura 4.2 mostra a mesma consulta sem a utilização do filtro de palavras-chave. Cada documento adicionado no índice do *Apache Solr*, possui o mesmo conjunto de campos. Estes campos armazenam as informações de cadastro do documento. A Figura 4.1 mostra dois campos do índice que

[12] http://lascam.facom.ufu.br:8080/crowdbugs/

são consultados: *"postTitle"* e *"codeText"*. O campo *"postTitle"* refere-se ao título do *post* do SOF. Já o campo *"codeText"* contém o resultado do pré-processamento do trecho de código. Neste caso, o trecho de código Java foi pré-processado com a função *lsab-java*. As figuras mostram as chamadas de método de API que foram extraídas a partir do trecho de código Java.

A consulta referente à Figura 4.1 possui o seguinte significado: retornar os *posts* do SOF (i.e., documentos do índice) que possuem pelo menos uma das palavras-chave definidas na Tabela 4.5 em seu título e cujo código possui as chamadas de método de API do trecho de código pesquisado. Já a consulta referente à Figura 4.2 possui o seguinte significado: retornar os *posts* do SOF (i.e., documentos do índice) cujo código possui as chamadas de método de API do trecho de código pesquisado.

```
((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 4.1: Exemplo de *query* do *Apache Solr* com o filtro de palavras-chave.

```
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 4.2: Exemplo de *query* do *Apache Solr* sem o filtro de palavras-chave.

Para cada um dos 15 *"crowd bugs"* selecionados para a linguagem Java, foram feitas 8 consultas diferentes, sendo 2 consultas por índice (i.e., **pp1** sem filtro, **pp1** com filtro, **pp2** sem filtro, **pp2** com filtro, **ppa** sem filtro, **ppa** com filtro, **lsab-java** sem filtro e **lsab-java** com filtro). Para cada um dos 15 *"crowd bugs"* selecionados para a linguagem JavaScript, foram feitas 6 consultas diferentes, 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). Observa-se que a primeira consulta de cada índice não utilizou o filtro de palavras-chave, enquanto que a segunda consulta de cada índice utilizou tal filtro.

Para a realização do Experimento III, foram utilizados os mesmos índices definidos no Experimento II (ver Subseção 4.4.2), com exceção dos índices **Java-raw** e **JavaScript-raw** (em que nenhum pré-processamento é realizado nos trechos de código-fonte). Tanto a função de pré-processamento **pp1** quanto a função de pré-processamento **raw** sofrem do problema da presença de palavras reservadas da linguagem (e.g., *"stop words"* como: *public*, *class*, *Double*, etc.). Por isso, no Experimento III, os índices **Java-raw** e **JavaScript-raw** não foram considerados. No lugar deles, foram considerados os índices léxicos **Java-pp1** e **JavaScript-pp1**, que representam os casos onde há a presença de palavras reservadas da linguagem na *query* do *Apache Solr*.

Para a construção destes índices, foi utilizada a seguinte abordagem baseada em regras:

- **Regra 1**: Os *posts* do SOF que irão compor o índice do *Apache Solr* deverão conter obrigatoriamente código na pergunta;
- **Regra 2**: Todos os trechos de código do *post* do SOF (i.e., trecho(s) de código da pergunta juntamente com o(s) trecho(s) de código da(s) resposta(s)), após a realização do pré-processamento, deverão ser indexados de forma com que cada *post* corresponda a exatamente um documento no índice do *Apache Solr*.

A **Regra 1** surgiu pois todos os 30 *posts* do SOF do *dataset* continham trecho de código na pergunta. Isto pode ser explicado por um comportamento comum que a maioria dos desenvolvedores têm quando estão tendo algum problema com um determinado método de uma API, que é postar o código com *bugs* em serviços Q&A com o intuito de obter uma correção para seu problema.

A Tabela 4.6 mostra o número de documentos para cada índice considerado.

Tabela 4.6: Número de Documentos por índice.

Índice	Número de Documentos
Java-raw	232.846
Java-pp1	232.846
Java-pp2	232.846
Java-ppa	232.846
Java-lsab	232.846
JavaScript-raw	277.036
JavaScript-pp1	277.036
JavaScript-pp3	277.036
JavaScript-lsab	277.036

4.7 Resultados do Experimento III

Esta seção apresenta os resultados obtidos no Experimento III.

As Tabelas 4.7 e 4.8 possuem a mesma estrutura de colunas: a primeira coluna que contém o endereço do *post* do SOF que corrige o *"crowd bug"* e as colunas Pos_{pp1}, Pos_{pp2}, Pos_{ppa} e $\text{Pos}_{lsab-java}$ que referem-se às posições ocupadas por estes *posts* do SOF no *ranking* Top-100 produzido pelo motor de busca para as funções de pré-processamento **pp1**, **pp2**, **ppa** e **lsab-java**, respectivamente. Ambas as Tabelas (4.7 e 4.8) mostram os resultados para os 15 *"crowd bugs"* Java do *dataset* porém utilizando consultas diferentes nos índices *Solr*: os resultados da Tabela 4.7 foram obtidos sem a utilização do filtro de palavras-chave na *query* do *Apache Solr*, enquanto que os resultados da Tabela 4.8 foram obtidos com a utilização deste filtro.

As Tabelas 4.9 e 4.10 possuem a mesma estrutura de colunas: a primeira coluna que contém o endereço do *post* do SOF que corrige o *"crowd bug"* e as colunas Pos_{pp1}, Pos_{pp3} e $\text{Pos}_{lsab-js}$ que referem-se às posições ocupadas por estes *posts* do SOF no *ranking* Top-100 produzido pelo motor de busca para as funções de pré-processamento **pp1**, **pp3** e **lsab-js**, respectivamente. Ambas as Tabelas (4.9 e 4.10) mostram os resultados para os 15 *"crowd bugs"* JavaScript do *dataset* porém utilizando consultas diferentes nos índices *Solr*: os resultados da Tabela 4.9 foram obtidos sem a utilização do filtro de palavras-chave na *query* do *Apache Solr*, enquanto que os resultados da Tabela 4.10 foram obtidos com a utilização deste filtro.

As Tabelas 4.11, 4.12, 4.13 e 4.14 mostram as porcentagens dos *posts* do SOF encontradas por tipo de *Ranking* (i.e., "Top-100", "Top-50", "Top-25" e "Top-15") para as funções de pré-processamento de código-fonte consideradas no estudo (com exceção da função "raw" que não participou do Experimento III).

As Tabelas 4.11 e 4.12 apresentam os resultados obtidos para os *"crowd bugs"* da linguagem Java, considerando o uso das funções de pré-processamento de código **pp1**, **pp2**, **ppa** e **lsab-java**. A Tabela 4.11 mostra os resultados obtidos sem a utilização do filtro de palavras-chave na consulta do *Apache Solr*. Já a Tabela 4.12 mostra os resultados obtidos com a utilização deste filtro. Observa-se que a função de pré-processamento **lsab-java** obteve valores promissores: 100% dos *posts* dos 15 *"crowd bugs"* da linguagem Java foram encontrados no Top-100 na abordagem com filtro, sendo 80% no Top-25 e 73,33% no Top-15 (veja na Tabela 4.12). Já a função de pré-processamento **ppa** obteve resultados ruins: nenhum dos 15 *"crowd bugs"* da linguagem Java foi encontrado no Top-100 na abordagem sem filtro (veja Tabela 4.11).

As Tabelas 4.11, 4.12, 4.13 e 4.14 mostram as porcentagens dos *posts* do SOF encontradas por tipo de *Ranking* (i.e., "Top-100", "Top-50", "Top-25" e "Top-15") para as funções de pré-processamento de código-fonte consideradas no estudo (com exceção da função "raw" que não participou do Experimento III).

As Tabelas 4.13 e 4.14 apresentam os resultados obtidos para os *"crowd bugs"* da linguagem JavaScript, considerando o uso das funções de pré-processamento de código **pp1**, **pp3** e **lsab-js**. A Tabela 4.13 mostra os resultados obtidos sem a utilização do filtro

Tabela 4.7: Resultados das consultas nos índices *pp1*, *pp2*, *ppa* e *lsab-java* **sem a utilização do filtro de palavras-chave** (-1 = Não encontrado no *ranking*. "Pos" referem-se às posições ocupadas pelos *posts* do SOF no *ranking*.)

SOF *post* que corrige o *"crowd bug"*	Pos_{pp1}	Pos_{pp2}	Pos_{ppa}	$Pos_{lsab-java}$
http://stackoverflow.com/questions/7215621/why-does-javas-date-getyear-return-111-instead-of-2011	-1	53	-1	27
http://stackoverflow.com/questions/12175674/is-java-math-bigintegers-usage-wrong	2	6	-1	2
http://stackoverflow.com/questions/12975924/math-cos-java-gives-wrong-result	-1	-1	-1	1
http://stackoverflow.com/questions/10603336/bigdecimal-floor-rounding-goes-wrong	-1	-1	-1	77
http://stackoverflow.com/questions/9065727/why-long-tohexstring0xffffffff-returns-ffffffffffffffff	-1	-1	-1	2
http://stackoverflow.com/questions/11597244/why-the-result-of-integer-tobinarystring-for-short-type-argument-includes-32-bit	29	-1	-1	36
http://stackoverflow.com/questions/14836004/java-date-giving-the-wrong-date	-1	67	-1	17
http://stackoverflow.com/questions/9956471/wrong-result-by-java-math-pow	-1	-1	-1	-1
http://stackoverflow.com/questions/12213877/is-identityhashmap-class-wrong	-1	-1	-1	1
http://stackoverflow.com/questions/14213778/unexpected-result-with-outputstream-in-java	-1	-1	-1	34
http://stackoverflow.com/questions/9230961/unexpected-output-while-converting-a-string-to-date-in-java	-1	-1	-1	-1
http://stackoverflow.com/questions/6899019/java-simpledateformat-returns-unexpected-result	-1	-1	-1	-1
http://stackoverflow.com/questions/1755199/calendar-returns-wrong-month	-1	-1	-1	-1
http://stackoverflow.com/questions/13896614/surprising-result-from-math-pow65-17-3233	-1	-1	-1	-1
http://stackoverflow.com/questions/14102593/unexpected-output-with-iso-time-8601	-1	-1	-1	39

de palavras-chave na consulta do *Apache Solr*. Já a Tabela 4.14 mostra os resultados obtidos com a utilização deste filtro. Observa-se que a função de pré-processamento **lsab-js** obteve bons resultados: 66,66% dos *posts* dos 15 *"crowd bugs"* da linguagem JavaScript foram encontrados no Top-25 na abordagem com filtro, sendo 46,66% no Top-15 (veja Tabela 4.14). Já a função de pré-processamento **pp3** obteve resultados ruins: apenas 13,33% dos 15 *"crowd bugs"* da linguagem JavaScript foi encontrado no Top-100 na abordagem sem filtro (veja Tabela 4.13).

Tabela 4.8: Resultados das consultas nos índices *pp1*, *pp2*, *ppa* e *lsab-java* **com a utilização do filtro de palavras-chave** (-1 = Não encontrado no *ranking*. "Pos" referem-se às posições ocupadas pelos *posts* do SOF no *ranking*.)

SOF *post* que corrige o *"crowd bug"*	Pos_{pp1}	Pos_{pp2}	Pos_{ppa}	$Pos_{lsab-java}$
http://stackoverflow.com/questions/7215621/why-does-javas-date-getyear-return-111-instead-of-2011	-1	4	-1	3
http://stackoverflow.com/questions/12175674/is-java-math-bigintegers-usage-wrong	2	1	10	1
http://stackoverflow.com/questions/12975924/math-cos-java-gives-wrong-result	25	-1	-1	1
http://stackoverflow.com/questions/10603336/bigdecimal-floor-rounding-goes-wrong	-1	16	23	9
http://stackoverflow.com/questions/9065727/why-long-tohexstring0xffffffff-returns-ffffffffffffffff	-1	-1	-1	1
http://stackoverflow.com/questions/11597244/why-the-result-of-integer-tobinarystring-for-short-type-argument-includes-32-bit	6	-1	-1	5
http://stackoverflow.com/questions/14836004/java-date-giving-the-wrong-date	-1	9	85	3
http://stackoverflow.com/questions/9956471/wrong-result-by-java-math-pow	35	-1	-1	12
http://stackoverflow.com/questions/12213877/is-identityhashmap-class-wrong	-1	34	-1	1
http://stackoverflow.com/questions/14213778/unexpected-result-with-outputstream-in-java	59	8	71	6
http://stackoverflow.com/questions/9230961/unexpected-output-while-converting-a-string-to-date-in-java	-1	-1	-1	30
http://stackoverflow.com/questions/6899019/java-simpledateformat-returns-unexpected-result	6	2	96	18
http://stackoverflow.com/questions/1755199/calendar-returns-wrong-month	8	-1	86	66
http://stackoverflow.com/questions/13896614/surprising-result-from-math-pow65-17-3233	-1	38	-1	27
http://stackoverflow.com/questions/14102593/unexpected-output-with-iso-time-8601	20	62	-1	6

Tabela 4.9: Resultados das consultas nos índices *pp1*, *pp3* e *lsab-js* **sem a utilização do filtro de palavras-chave** (-1 = Não encontrado no *ranking*. "Pos" referem-se às posições ocupadas pelos *posts* do SOF no *ranking*.)

SOF *post* que corrige o *"crowd bug"*	Pos_{pp1}	Pos_{pp3}	$Pos_{lsab-js}$
http://stackoverflow.com/questions/2587345/javascript-date-parse	-1	-1	-1
http://stackoverflow.com/questions/262427/javascript-arraymap-and-parseint	-1	-1	-1
http://stackoverflow.com/questions/1845001/array-sort-is-not-giving-the-desired-result	-1	-1	-1
http://stackoverflow.com/questions/9766201/javascript-unexpected-behaviour-during-subtraction	-1	-1	100
http://stackoverflow.com/questions/8524933/json-parse-unexpected-character-error	-1	-1	23
http://stackoverflow.com/questions/18509996/get-index-of-empty-space-returns-wrong-value	-1	-1	-1
http://stackoverflow.com/questions/12318830/parseint08-returns-0	-1	83	-1
http://stackoverflow.com/questions/10706272/unexpected-javascript-date-behavior	-1	35	50
http://stackoverflow.com/questions/9859995/unexpected-output-in-javascript	-1	-1	-1
http://stackoverflow.com/questions/8160550/unexpected-result-adding-numbers-79-0014-95-why	-1	-1	-1
http://stackoverflow.com/questions/11477415/why-does-javascripts-regex-exec-not-always-return-the-same-value	-1	-1	-1
http://stackoverflow.com/questions/8979093/json-stringifyobject-incorrect	-1	-1	-1
http://stackoverflow.com/questions/11363526/weird-output-of-97-98-mapstring-fromcharcode	44	-1	-1
http://stackoverflow.com/questions/6223616/javascript-math-cos-and-math-sin-are-inaccurate-is-there-any-solution	78	-1	2
http://stackoverflow.com/questions/834757/why-does-getday-return-incorrect-values-javascript	-1	-1	5

Tabela 4.10: Resultados das consultas nos índices *pp1*, *pp3* e *lsab-js* **com a utilização do filtro de palavras-chave** (-1 = Não encontrado no *ranking*. "Pos" referem-se às posições ocupadas pelos *posts* do SOF no *ranking*.)

SOF *post* que corrige o *"crowd bug"*	Pos_{pp1}	Pos_{pp3}	$Pos_{lsab-js}$
http://stackoverflow.com/questions/2587345/javascript-date-parse	-1	-1	-1
http://stackoverflow.com/questions/262427/javascript-arraymap-and-parseint	-1	-1	-1
http://stackoverflow.com/questions/1845001/array-sort-is-not-giving-the-desired-result	-1	-1	-1
http://stackoverflow.com/questions/9766201/javascript-unexpected-behaviour-during-subtraction	22	-1	1
http://stackoverflow.com/questions/8524933/json-parse-unexpected-character-error	-1	-1	2
http://stackoverflow.com/questions/18509996/get-index-of-empty-space-returns-wrong-value	-1	-1	9
http://stackoverflow.com/questions/12318830/parseint08-returns-0	-1	12	17
http://stackoverflow.com/questions/10706272/unexpected-javascript-date-behavior	7	1	2
http://stackoverflow.com/questions/9859995/unexpected-output-in-javascript	-1	-1	5
http://stackoverflow.com/questions/8160550/unexpected-result-adding-numbers-79-0014-95-why	-1	-1	-1
http://stackoverflow.com/questions/11477415/why-does-javascripts-regex-exec-not-always-return-the-same-value	-1	-1	22
http://stackoverflow.com/questions/8979093/json-stringifyobject-incorrect	-1	83	22
http://stackoverflow.com/questions/11363526/weird-output-of-97-98-mapstring-fromcharcode	2	-1	14
http://stackoverflow.com/questions/6223616/javascript-math-cos-and-math-sin-are-inaccurate-is-there-any-solution	-1	-1	-1
http://stackoverflow.com/questions/834757/why-does-getday-return-incorrect-values-javascript	3	9	1

Tabela 4.11: Porcentagens de *posts* do SOF encontrados por tipo de *ranking* para as funções de pré-processamento **pp1**, **pp2**, **ppa** e **lsab-java**, sem utilizar o filtro de palavras-chave na consulta do *Apache Solr*.

Ranking	pp1	pp2	ppa	lsab-java
Top-100	13,33%	20%	0%	66,66%
Top-50	13,33%	6,66%	0%	60%
Top-25	6,66%	6,66%	0%	33,33%
Top-15	6,66%	6,66%	0%	26,66%

Tabela 4.12: Porcentagens de *posts* do SOF encontrados por tipo de *ranking* para as funções de pré-processamento **pp1**, **pp2**, **ppa** e **lsab-java**, considerando o uso do filtro de palavras-chave na consulta do *Apache Solr*.

Ranking	pp1	pp2	ppa	lsab-java
Top-100	53,33%	60%	40%	100%
Top-50	46,66%	53,33%	13,33%	93,33%
Top-25	40%	40%	13,33%	80%
Top-15	26,66%	33,33%	6,66%	73,33%

Tabela 4.13: Porcentagens de *posts* do SOF encontrados por tipo de *ranking* para as funções de pré-processamento **pp1**, **pp3** e **lsab-js**, sem utilizar o filtro de palavras-chave na consulta do *Apache Solr*.

Ranking	pp1	pp3	lsab-js
Top-100	13,33%	13,33%	33,33%
Top-50	6,66%	6,66%	26,66%
Top-25	0%	0%	20%
Top-15	0%	0%	13,33%

Tabela 4.14: Porcentagens de *posts* do SOF encontrados por tipo de *ranking* para as funções de pré-processamento **pp1**, **pp3** e **lsab-js**, considerando o uso do filtro de palavras-chave na consulta do *Apache Solr*.

Ranking	pp1	pp3	lsab-js
Top-100	26,66%	26,66%	66,66%
Top-50	26,66%	20%	66,66%
Top-25	26,66%	20%	66,66%
Top-15	20%	20%	46,66%

4.8 Discussão dos Resultados do Experimento III

Esta seção discute os resultados obtidos no Experimento III.

Os resultados das Tabelas 4.8 e 4.10 demonstram que as funções de pré-processamento **lsab-java** com filtro e **lsab-js** com filtro foram superiores às outras funções de pré-processamento consideradas neste experimento.

Monperrus et al. [35] propuseram uma nova abordagem de depuração para *"crowd bugs"*, que é baseada em fazer o casamento de padrões entre o trecho de código que está sendo depurado e trechos de código relacionados presentes no serviço Q&A SOF. Neste trabalho, investigou-se um tipo específico de *"crowd bugs"*, que são aqueles relacionados com chamadas de método de uma API (*Application Programming Interfaces*) presentes no trecho de código-fonte. Existem várias diferenças entre este trabalho e o artigo de Monperrus et al. [35]:

- Aquele artigo não apresentou um *dataset* contendo *"crowd bugs"*. O presente trabalho apresenta um *dataset* contendo 30 *"crowd bugs"* (15 da linguagem Java e 15 da linguagem JavaScript) extraídos do *site OHLOH Code Search*;
- Ambos os trabalhos apresentam uma estratégia para correção de *"crowd bugs"*. Porém, aquele artigo investigou somente os *"crowd bugs"* que são escritos utilizando a linguagem JavaScript, enquanto que o presente trabalho investiga tanto os *"crowd bugs"* que são escritos utilizando a linguagem JavaScript quanto os *"crowd bugs"* que são escritos utilizando a linguagem Java;
- Aquele artigo investigou somente 3 funções de pré-processamento para trechos de código-fonte (i.e., **"raw"**, **"pp1"** e **"pp3"**). O presente trabalho conduziu experimentos com 7 funções de pré-processamento para trechos de código (i.e., **"raw"**, **"pp1"**, **"pp2"**, **"pp3"**, **"ppa"**, **"lsab-java"** e **"lsab-js"**).

As Figuras 4.3 e 4.4 mostram gráficos de barras contendo as porcentagens de *posts* do SOF encontrados no Top-100 e no Top-15, respectivamente, para os *"crowd bugs"* da linguagem Java, considerando a utilização ou não-utilização do filtro de palavras-chave.

As Figuras 4.5 e 4.6 mostram gráficos de barras contendo as porcentagens de *posts* do SOF encontrados no Top-100 e no Top-25, respectivamente, para os *"crowd bugs"* da linguagem JavaScript, considerando a utilização ou não-utilização do filtro de palavras-chave.

Os gráficos mostram uma clara vantagem em torno de 40% da abordagem **com o uso do filtro** em relação à abordagem **sem o uso do filtro**. Conclui-se portanto, que a utilização deste filtro melhora o *ranking* produzido pelo motor de busca. Para os *"crowd bugs"* da linguagem Java, a estratégia de se utilizar o índice **Java-lsab** em conjunto com o filtro de palavras-chave na *query* do *Apache Solr* mostrou-se superior às demais. Para os *"crowd bugs"* da linguagem JavaScript, os melhores resultados obtidos até o momento

apontam para o uso do índice **JavaScript-lsab** em conjunto com o filtro de palavras-chave na *query* do *Apache Solr*.

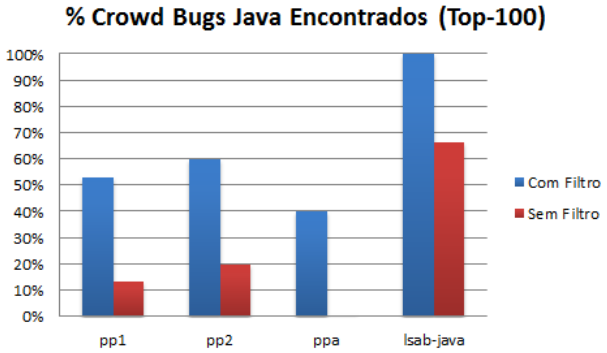

Figura 4.3: Porcentagem de *"crowd bugs"* Java encontrados no Top-100.

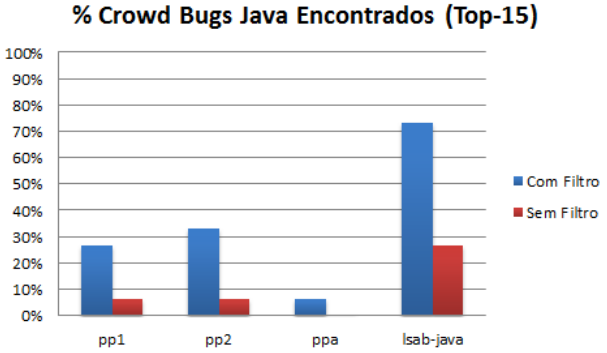

Figura 4.4: Porcentagem de *"crowd bugs"* Java encontrados no Top-15.

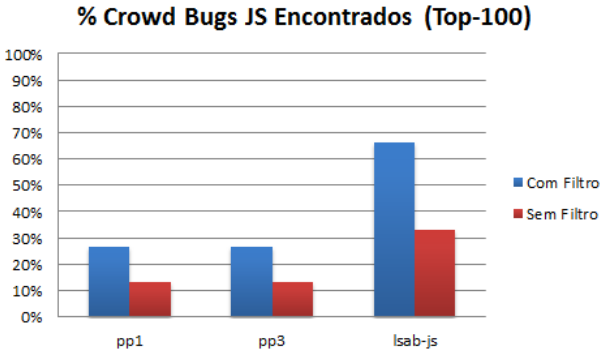

Figura 4.5: Porcentagem de *"crowd bugs"* JavaScript (JS) encontrados no Top-100.

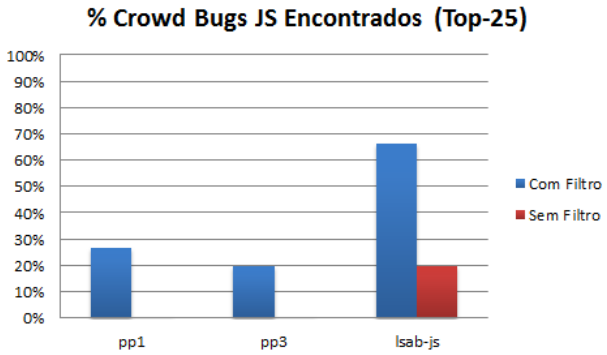

Figura 4.6: Porcentagem de *"crowd bugs"* JavaScript (JS) encontrados no Top-25.

Resposta à Q4: Dado que o SOF possui cobertura dos *"crowd bugs"*, como está o ranqueamento feito pelo sistema de recomendação proposto utilizando *"crowd bugs"* pertencentes a projetos reais de *software*? Os experimentos com *"crowd bugs"* reais revelaram que a abordagem proposta obteve melhores resultados com os *"crowd bugs"* da linguagem Java do que com os *"crowd bugs"* da linguagem JavaScript. Para a linguagem Java, o sistema de recomendação proposto obteve resultados promissores utilizando a função de pré-processamento **lsab-java** combinada com o filtro de palavras-chave (100% dos *posts* foram encontrados no Top-100, sendo 73,33% no Top-15). Para a linguagem JavaScript, o sistema de recomendação obteve resultados promissores utilizando a função de pré-processamento **lsab-js** combinada com o filtro de palavras-chave (66,66% dos *posts* do SOF foram encontrados no Top-25, sendo 46,66% no Top-15). Os resultados sugerem uma maior viabilidade de uso dos índices que são construídos utilizando a abordagem *Lightweight Syntactic Analysis and Binding* (i.e., extração das chamadas de métodos das bibliotecas a partir dos trechos de código-fonte).

4.9 Conclusões

Até o momento, o tema *"crowd bugs"* não foi muito estudado na literatura.

Monperrus et al. [35] definiram o conceito de *"crowd bugs"* e propuseram a idéia de "depurar com a multidão". Eles implementaram um novo sistema de *debug* para linguagem JavaScript composto por 70.060 Q&As extraídos do SOF. O presente trabalho propôs um sistema de recomendação de correções para *"crowd bugs"* para as linguagens Java e JavaScript. Além disso, um *dataset* composto por *"crowd bugs"* foi disponibilizado e pode ser utilizado em outras pesquisas na área de *"crowd bugs"*.

Carzaniga et al. [8] provou que a *Web* contém algumas descrições de *"crowd bugs"*. O exemplo de motivação sobre **parseInt** bem como o *dataset* proposto mostrou que o SOF contém vários desses *"crowd bugs"* para diferentes linguagens de programação (no caso, Java e JavaScript). Os *bugs* mencionados por Carzaniga et al. também podem ser encontrados no SOF.

Neste capítulo, apresentou-se uma nova abordagem para depuração de *software*. Esta abordagem propõe uma estratégia de recomendação para auxiliar os desenvolvedores durante as tarefas de depuração. Existe uma classe de *bugs*, denominada *"crowd bugs"* para a qual a abordagem de depuração é relevante. A estratégia de recomendação proposta é composta por um motor de busca e um índice, sendo este índice formado por um conjunto de documentos criados a partir do SOF.

Foram realizados diversos experimentos envolvendo várias funções de pré-processamento (léxicas e sintáticas) para as linguagens de programação Java e JavaScript com o objetivo de investigar quais delas possuem melhor eficácia ao lidar com consultas baseadas em trechos de código-fonte. Para isso, foi criado um *dataset* povoado com 30 trechos de código extraídos do *OHLOH Code Search* que supostamente poderiam possuir *"crowd bugs"*, uma vez que tais trechos de código invocam métodos de bibliotecas que frequentemente geram saídas incorretas dependendo da maneira de como estes métodos são utilizados pelo desenvolvedor.

O uso do filtro de palavras-chave na *query* do *Apache Solr* contribuiu de maneira significativa no ranqueamento geral produzido pelo motor de busca. Os resultados são melhores do que a abordagem que não utiliza este filtro, tanto para os *"crowd bugs"* da linguagem Java quanto para os *"crowd bugs"* da linguagem JavaScript.

Os resultados mostram que para a linguagem Java, a melhor configuração foi obtida utilizando a função de pré-processamento **lsab-java** combinada com o filtro de palavras-chave (100% dos *posts* do SOF foram encontrados no Top-100 resultados retornados pelo motor de busca *Apache Solr*, sendo 80% no Top-20 e 73,33% no Top-15). Para a linguagem JavaScript, a melhor configuração foi obtida utilizando a função de pré-processamento **lsab-js** combinada com o filtro de palavras-chave (66,66% dos *posts* foram encontrados no Top-25, sendo 46,66% no Top-15). Desta forma, a estratégia de extrair apenas as chamadas de métodos de API a partir dos trechos de código do SOF, resolvendo os *bindings* de "nome –> tipo" das variáveis, demonstrou ser bastante promissora para a correção dos *"crowd bugs"* para ambas as linguagens de programação estudadas.

Capítulo 5

Considerações Finais

O presente trabalho de dissertação endereçou dois temas de natureza diferentes: auxiliar os desenvolvedores na aprendizagem de API para o desenvolvimento de *software* e auxiliar o desenvolvedor durante as tarefas de depuração para a correção de *bugs*. As soluções para ambos os problemas consistiu na utilização do "conhecimento da multidão" presente no SOF para recomendar *posts* que irão ajudar o desenvolvedor no desenvolvimento e depuração do *software*.

Apesar dos dois temas possuírem características distintas, ambos estão relacionados com o problema de aprendizagem de APIs. O fato das APIs apresentarem, muitas vezes, falta de exemplos de uso explicando como utilizar os elementos da API (i.e., classes e métodos) leva o desenvolvedor a procurar por ajuda nos serviços de mídia social (como o SOF).

Neste sentido, os desenvolvedores podem estar procurando por trechos de código e explicações para resolver uma determinada tarefa de programação em uma dada API ou procurar por correções de *"crowd bugs"* na comunidade (frequentemente, são postados os trechos de código contendo estas chamadas de método de API que geram *bugs* dependendo da forma com que são utilizados pelo desenvolvedor). Além disso, pode-se afirmar que os *"crowd bugs"* estão relacionados a um problema de *design* de API, ou seja, os métodos da API foram implementados de maneira que facilita o aparecimento do *bug*, uma vez que estes métodos de API possuem uma natureza que vai contra o senso comum da maioria dos desenvolvedores que utilizam tal método da API.

Os resultados em ambos os casos mostraram a viabilidade da abordagem para aplicação no cotidiano de equipes de desenvolvimento de *software*.

Como resultado deste trabalho, foram publicados os seguintes artigos em conferências nacionais e internacionais: [15], [7], [6] e [45].

Trabalhos Futuros

Como trabalhos futuros, pretende-se comparar a abordagem de recomendação de pares Q&A proposta com uma pesquisa *Web* utilizando o motor de busca do Google. O Google será escolhido pois é o motor de busca de propósito geral mais popular e é geralmente utilizado para consultar o SOF. Para isso, pretende-se utilizar as mesmas 35 tarefas de programação que foram extraídas dos *cookbooks* referente às APIs (*Swing*, *Boost* e *LINQ*).

Em relação ao trabalho de *"crowd bugs"*, pretende-se propor um sistema de reparo automático de *software* para esta classe de *bugs*. A justificativa para este trabalho é: *"o reparo automático de crowd bugs proporcionam bugs a menos no issue tracker do projeto"*, impactando beneficamente na produtividade de manutenção do projeto. O principal desafio deste trabalho consiste em demonstrar que de fato seria reduzido a quantidade de *"crowd bugs"* no *issue tracker* do projeto. Portanto, será necessário realizar um estudo de viabilidade para avaliar o impacto que a criação de um sistema de reparo automático para *"crowd bugs"* possui no processo de manutenção do *software*.

Referências Bibliográficas

[1] B. Azhagusundari and A. S. Thanamani. Feature selection based on information gain. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, pages 18–19, 2013.

[2] O. Barzilay, C. Treude, and A. Zagalsky. *Facilitating Crowd Sourced Software Engineering via Stack Overflow*, pages 297–316. Springer, New York, 2013.

[3] A. Bialecki, R. Muir, and G. Ingersoll. Apache lucene 4. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, pages 1–8, 2012.

[4] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C.J.Stone. Classification and regression trees. 1984.

[6] E. Campos, L. Souza, and M. d. A. Maia. Nuggets Miner: Assisting Developers by Harnessing the StackOverflow Crowd Knowledge and the GitHub Traceability. In *Proc. of the Brazilian Conference on Software: Theory and Practice (CBSoft 2014) - Tool Session*, page 8pp, Maceió, Brazil, 2014.

[7] E. C. Campos and M. d. A. Maia. Automatic categorization of questions from Q&A sites. In *Proceedings of the 29th Symposium On Applied Computing*, SAC. ACM, 2014.

[8] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 237–246, New York, NY, USA, 2010. ACM.

[9] K. Chai. Expectation of F-measures: Tractable exact computation and some empirical observations of its properties. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 593–594, 2005.

[10] J. Cohen. Weighted kappa: Nominal scale agreement with provision for scaled disagreement or partial credit. *Psychological Bulletin*, 1968.

[11] J. Cordeiro, B. Antunes, and P. Gomes. Context-based recommendation to support problem solving in Software Development. In *Proceedings of 3rd Int. Workshop on RSSE)*, pages 85–89, 2012.

[12] P. Cunningham. Overfitting and diversity in classification ensembles based on feature selection. Technical report, 2000.

[13] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. ACM.

[14] I. F. Darwin. *Java Cookbook*. O'Reilly Media, Sebastopol, CA, USA, 2004.

[15] L. B. L. de Souza, E. C. Campos, and M. d. A. Maia. Ranking Crowd Knowledge to Assist Software Development. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC, pages 72–82. ACM, 2014.

[16] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly Media, Sebastopol, CA, USA, 1998.

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

[18] X. Geng, T. Liu, T. Quin, and H. Li. Feature selection for ranking. pages 1–2, 2007.

[19] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 2003.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software : An update. *SIGKDD Explorations*, pages 10–18, 2009.

[21] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, PTR Upper Saddle River, NJ, USA, 1998.

[22] J. Hilyard and S. Teilhet. *C# 3.0 Cookbook*. O'Reilly Media, Sebastopol, CA, USA, 2007.

[23] R. Holmes and A. Begel. Deep Intellisense: A tool for rehydrating evaporated information. In *Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)*, pages 23–26. ACM, 2008.

[24] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Trans. Softw. Eng.*, pages 952–970, 2006.

[25] R. Kohavi. 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*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, Dec. 1997.

[27] S. le Cessie and J. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[28] M. Linares-Vasquez, C. McMillan, D. Poshyvanyk, and M. Grechanik. On using machine learning to automatically classify software applications into domain categories. pages 7–8, 2009.

[29] A. B. M. L. D. F. Luca Ponzanelli, Andrea Mocci. Improving Low Quality Stack Overflow Post Detection. In *n Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution)*, page to be published. IEEE, 2014.

[30] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2857–2866, New York, NY, USA, 2011. ACM.

[31] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[32] D. Mladenic and M. Grobelnik. Feature selection for unbalanced class distribution and naive bayes. *International Conference on Machine Learning (ICML)*, 1999.

[33] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the Apache Server. In *Proc. of the 22nd ICSE*, pages 263–272. ACM, 2000.

[34] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? An empirical study on the directives of API documentation. *Emp. Softw. Eng.*, 17(6):703–737. Springer US, 2012.

[35] M. Monperrus and A. Maia. Debugging with the Crowd: a Debug Recommendation System based on Stackoverflow. Technical report, INRIA, 2014.

[36] S. Nasehi, J. Sillito, F. Maurer, and C. Burns. 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)*, pages 25–34, 2012.

[37] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. *Georgia Tech, Tech. Rep.* IEEE Comp. Soc., 2012.

[38] M. Pohar, M. Blas, and S. Turk. Comparison of logistic regression and linear discriminant analysis : A simulation study. *MetodoloÅ¡ki Zvezki*, 1(1):143–144, 2004.

[39] A. Polukhin. *Boost C++ application Development Cookbook*. Packt Publ., Birmingham, 2013.

[40] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging Crowd Knowledge for Software Comprehension and Development. In *CSMR*, pages 57–66, 2013.

[41] M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[42] M. Robillard, R. Walker, and T. Zimmermann. Recommendation Systems for Software Engineering,. *IEEE Software*, pages 80–86, 2010. IEEE Computer Society.

[43] N. Sawadsky and G. C. Murphy. Fishtail: From Task Context to Source Code Examples. In *Proceedings of the 1st Workshop on Developing Tools As Plug-ins*, pages 48–51, New York, NY, USA, 2011. ACM.

[44] L. Sehgal, N. Mohan, and P. S. Sandhu. Quality prediction of function based software using decision tree approach. pages 43–44, September 2012.

[45] L. Souza, E. Campos, and M. d. A. Maia. On the Extraction of Cookbooks for APIs from the Crowd Knowledge. In *Proc. of the 28th Brazilian Symposium on Software Engineering*, SBES'2014, page 10pp, Maceió, Brazil, 2014.

[46] S. Subramanian and R. Holmes. Making Sense of Online Code Snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 85–88, Piscataway, NJ, USA, 2013. IEEE Press.

[47] W. Takuya and H. Masuhara. A Spontaneous Code Recommendation Tool Based on Associative Search. In *Proceedings of the 3rd International Workshop on Search-Driven Development*, pages 17–20. ACM, 2011.

[48] C. Treude, O. Barzilay, and M.-A. Storey. How Do Programmers Ask and Answer Questions on the Web? (NIER track). In *Proc. of the 33rd ICSE*, pages 804–807. ACM, 2011.

[49] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807. ACM, 2011.

[50] D. ČubraniĆ, G. C. Murphy, J. Singer, and K. S. Booth. Learning from Project History: A Case Study for Software Development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 82–91, New York, NY, USA, 2004. ACM.

[51] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. *International Conference on Machine Learning (ICML)*, 1997.

[52] Y. Yao, H. Tong, T. Xie, L. Akoglu, F. Xu, and J. Lu. Want a Good Answer? Ask a Good Question First! *ArXiv e-prints*, 2013.