Matheus Camilo da Silva

Um Método de Recomendação de Desenvolvedores Baseado em Métricas de Qualidade de Código

Curitiba - PR, Brasil

2021

Matheus Camilo da Silva

Um Método de Recomendação de Desenvolvedores Baseado em Métricas de Qualidade de Código

Apresentado ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de mestre em Informática.

Pontifícia Universidade Católica do Paraná - PUCPR
Programa de Pós-Graduação em Informática - PPGIa

Orientador: Prof. Dr. Emerson Cabrera Paraiso

Curitiba - PR, Brasil
2021

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Pamela Travassos de Freitas – CRB 9/1960

S586m
2020

Silva, Matheus Camilo da
Um método de recomendação de desenvolvedores baseado em métricas de qualidade de código / Matheus Camilo da Silva ; orientador: Emerson Cabrera Paraiso. – 2020.
79 f. : il.; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2020
Bibliografia: f. 73-79

1. Informática. 2. Controle de qualidade – Software. 3. Engenharia de Software. 4. Padrões de software. 5. Software – Desenvolvimento. 6. Software – Validação. I. Paraiso, Emerson Cabrera. II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD 20. ed. – 004

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

DECLARAÇÃO

Declaro para os devidos fins que o aluno **MATHEUS CAMILO DA SILVA**, defendeu sua dissertação de Mestrado intitulada "Um Método de Recomendação de Desenvolvedores Baseado em Métricas de Qualidade de Código", na área de concentração Ciência da Computação, no dia 11 de setembro de 2020, no qual foi aprovado.

Declaro ainda que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade, firmo a presente declaração.

Curitiba, 15 de dezembro de 2020.

____
Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática
Pontifícia Universidade Católica do Paraná

Rua Imaculada Conceição,1155 Prado Velho CEP 80215-901 Curitiba Paraná Brasil
Fone: (41) 3271-1669 e-mail: secretaria@ppgia.pucpr.br

Agradecimentos

Em primeiro lugar, eu quero agradecer a Deus, que sempre esteve comigo.

Também sou muito grato à minha família, que é a minha fortaleza.

Agradeço aos professores e funcionários do PPGIA, os quais proporcionaram o desenvolvimento deste trabalho através do ambiente de aprendizagem que criam.

Em especial, agradeço ao meu orientador Prof. Dr. Emerson Cabrera Paraiso, pela paciência, pelo aprendizado acadêmico e pessoal, e por me guiar nessa parte da minha vida.

E agradeço a Siemens Ltda pelo suporte financeiro e parceria durante o desenvolvimento do projeto.

Resumo

A tarefa de desenvolvimento de software é complexa e colaborativa, onde o trabalho de diferentes participantes deve ser coordenado. Durante o ciclo de desenvolvimento de um produto de software, é comum que requisitos e funcionalidades do software mudem e, como consequência, que erros de código apareçam. Para lidar com essas alterações imprevistas, uma possibilidade é a utilização do artefato conhecido como *requisição de alteração* (Change Requests). A atribuição do responsável por esta alteração é uma etapa importante do processo de desenvolvimento. Produtos podem receber diariamente um número muito elevado de Change Requests, o que torna a automação desse processo interessante. Este trabalho propõe um método para atribuir Change Requests a partir do perfil de desenvolvedores. Os perfis são um meio de agrupar desenvolvedores com características de trabalho semelhantes. O método proposto consiste de três etapas. A primeira etapa é referente à extração de métricas de qualidade de código, dados de *commit* e Change Requests previamente resolvidas, com o intuito de criar perfis de desenvolvedores por meio da mineração de repositórios. A segunda etapa é referente à seleção do perfil de desenvolvedores candidatos por meio da aplicação de técnicas de processamento de linguagem natural e recuperação da informação. E por fim, na terceira etapa os desenvolvedores apropriados são selecionados a partir da qualidade de seu código-fonte e no impacto de seus *commits*. Os resultados da avaliação experimental mostram que o método é capaz de recomendar mais desenvolvedores com um impacto positivo na qualidade do repositório comparado a um método estabelecido da literatura *iMacPro*.

Palavras-chave: Recomendação de Desenvolvedores, Resolução de Change Requests, Métricas de Qualidade de Software, Agrupamento de Desenvolvedores, Perfil de Desenvolvedores.

Abstract

The software development task is complex and collaborative, where the work of different participants must be coordinated. During the development cycle of a project, it is common for software requirements and functionality to change and, as a consequence, for code errors to appear. To deal with these unforeseen changes, one possibility is to use the artifact known as *change request*. The assignment of the person responsible for this change is an important step in the development process. Projects can receive a very high number of requests daily, which makes the automation of this process interesting. This work proposes a method for assigning unresolved requests from the developer profile. Profiles are a way of bringing together developers with similar work characteristics. The proposed method consists of three steps. The first step refers to the extraction of code quality metrics, data from *commit* and previously resolved requests, in order to create developer profiles through the mining of repositories. The second stage concerns the selection of the profile of candidate developers through the application of natural language processing and information retrieval techniques. Finally, in the third stage, the appropriate developers are selected based on the quality of their source code and the impact of their *commits*. The results of the experimental evaluation show that the method is able to recommend more developers with a positive impact on the quality of the repository compared to the base method of the literature *iMacPro*.

Keywords: Developer Recommendation, Change Request Resolution, Software Quality Metrics, Developer Grouping, Developer Profile.

Lista de ilustrações

Figura 1 – Ciclo de vida de uma **Change Request** 16
Figura 2 – Filtragem Baseada em Conteúdo 32
Figura 3 – Filtragem Baseada em Colaboração 32
Figura 4 – Método geral de recomendação de desenvolvedores 40
Figura 5 – Estrutura da pesquisa 41
Figura 6 – Visão geral do método proposto 46
Figura 7 – Exemplo Típico de Requisição de Alteração 49
Figura 8 – Exemplo de Requisição Resolvida 51
Figura 9 – Exemplo de Atribuição de Requisição 51
Figura 10 – Indexação de Requisições 53
Figura 11 – Representação Tridimensional dos Dados no Repositório *Guava* 54
Figura 12 – Gráfico de *Explained Variance para PCA* no Repositório *Guava* 55
Figura 13 – Grau de Variância dos Dados no Repositório *Guava* 56
Figura 14 – Inércia medido no repositório do *Guava* 57
Figura 15 – Seleção de Desenvolvedores candidatos 60
Figura 16 – Captura de tela da Ferramenta Visual 62
Figura 17 – Recomendação da Ferramenta Visual 65
Figura 18 – Exemplo de Arquivo de Extração da Ferramenta CK 68
Figura 19 – Subconjunto de 25% Repositório *Mockito* 73

Lista de tabelas

Quadro 1 – Definições de Métricas 27
Quadro 2 – Intervalos de Métricas 30
Quadro 3 – Repositórios Detalhados 66
Quadro 4 – Recall@k 70
Quadro 5 – Grau de Contribuição 71

Lista de abreviaturas e siglas

API *Application programming interface*

ARFF *Attribute-Relation File Format*

CR *Change Request*

CTM *Collaborative Topic Mining*

KNN *K nearest neighbors*

LSI *Latent Semantic Indexing*

SVD *Singular-Value Decomposition*

PLN Processamento de Linguagem Natural

RI Recuperção da Informação

XML *eXtensible Markup Language*

VG *McCabe Ciclomatic Complexity*

PAR *Number of Parameters*

NBD *Nested Block Depth*

CA *Afferent Coupling*

CE *Efferent Coupling*

RMI *Instability*

RMA *Abstractness*

RMD *Normalized Distance*

DIT *Depth Inheritance Tree*

WMC *Weight Method Class*

NSC *Number of Children*

NORM *Number of Overridden*

LCOM *Lack of Cohesion of Methods*

NOF	*Number of Functions*
NSF	*Number of Static Attributes*
NOM	*Number of Methods*
NSM	*Number of Static Methods*
SIX	*Specialization Index*
NOC	*Number of Classes*
NOI	*Number of Interfaces*
NOP	*Number of Packages*
TLOC	*Total Lines of Code*
MLOC	*Method Lines of Code*
NOSI	*Number of static invocations*

Sumário

1 INTRODUÇÃO . . . 15
1.1 Motivação . . . 18
1.2 Objetivo . . . 19
1.3 Hipóteses de Trabalho . . . 19
1.4 Contribuições Científicas e Tecnológicas . . . 20
1.5 Escopo do Trabalho . . . 20
1.6 Organização . . . 20

2 FUNDAMENTAÇÃO TEÓRICA . . . 22
2.1 Modelagem de Usuários . . . 22
2.1.1 Composição do modelo . . . 23
2.1.2 Representação do modelo . . . 23
2.1.3 Aquisição do modelo . . . 24
2.1.4 Aprendizado do modelo . . . 24
2.1.5 Manutenção do modelo . . . 25
2.2 Qualidade de Código . . . 25
2.2.1 Métricas de Qualidade de Código Orientado à Objetos . . . 26
2.2.1.1 Influência em Parâmetros de Qualidade . . . 29
2.2.1.2 Intervalo de Risco . . . 30
2.3 Modelos de Recomendação . . . 31
2.4 Processamento de Linguagem Natural e Recuperação de Informação 33
2.4.1 Processamento de Linguagem Natural . . . 33
2.4.2 Recuperação de Informação . . . 33
2.4.3 Latent Semantic Indexing . . . 34

3 TRABALHOS CORRELATOS . . . 35
3.1 Modelagem de Desenvolvedores . . . 35
3.2 Recomendação de Desenvolvedores . . . 37
3.3 Considerações Finais . . . 38

4 PROCEDIMENTOS METODOLÓGICOS . . . 41
4.1 Fase de Planejamento Inicial . . . 42
4.2 Fase Exploratória . . . 43
4.3 Fase de Desenvolvimento . . . 43
4.4 Fase de Avaliação . . . 44

5 MÉTODO PROPOSTO 45
5.1 Pressupostos do Método 45
5.2 Visão Geral 45
5.2.1 Modelagem de Desenvolvedores 46
5.2.1.1 Composição do Modelo 47
5.2.1.2 Representação 47
5.2.1.3 Aquisição 50
5.2.1.4 Aprendizado 52
5.2.1.5 Manutenção do Modelo 55
5.2.2 Atribuição de Change Request 56
5.2.3 Recomendação do Desenvolvedor 60

6 RESULTADOS 62
6.1 Método para o suporte de qualidade de código colaborativo 62
6.2 Recomendação de Desenvolvedores 65
6.2.1 Ferramentas Utilizadas 66
6.2.2 Experimentos 67
6.3 Aplicação do método proposto em ambiente privado 71

7 CONCLUSÃO 74

REFERÊNCIAS 75

1 Introdução

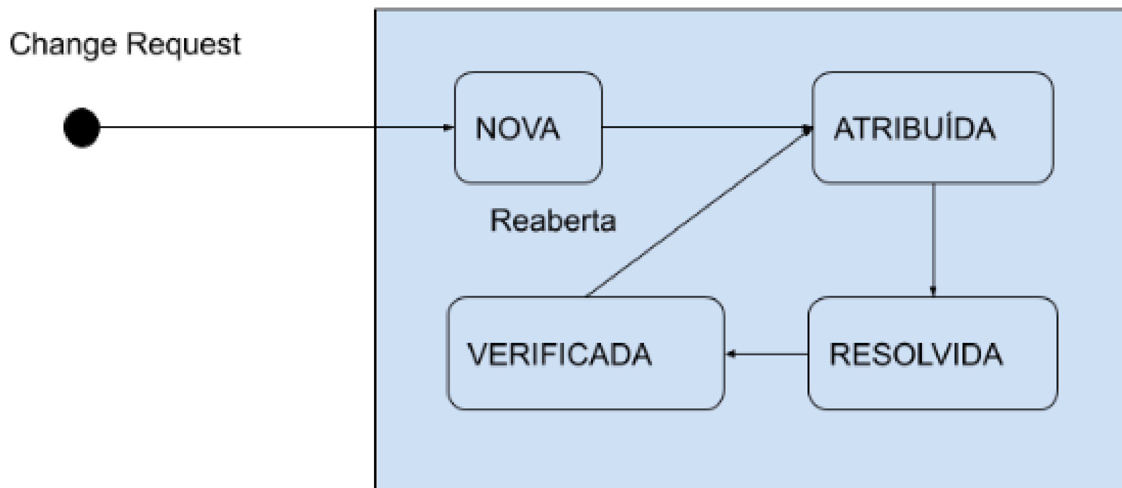
A popularização de *smartphones* e computadores trouxe um aumento no acesso de pessoas à internet, chegando a cerca de 70% de pessoas conectadas à internet no mundo todo em 2015 (POUSHTER et al., 2016) e consequentemente também no uso de software. Essa revolução digital tem grande repercussão na sociedade, uma vez que ela traz mudanças na comunicação, socialização e exercício das atividades das pessoas (MCNUTT, 2014). O crescente uso de tecnologias para atividades humanas, engloba não somente o ambiente corporativo, como também afetam diretamente o cotidiano das pessoas (MCNUTT, 2014), criando uma demanda cada vez maior por desenvolvimento de software. Nesse contexto, a tarefa de desenvolvimento de software tem um papel muito relevante na manutenção da sociedade.

No início do ciclo de desenvolvimento de software são estabelecidos os requisitos do produto. Entretanto, com o decorrer do tempo, é comum que alterações de projeto aconteçam. Essas alterações são organizadas por meio dos artefatos de software chamados de Requisições de Alterações ou *Change Request* (CR), em inglês. A natureza dessas requisições podem variar de acordo com o projeto e os aspectos organizacionais da equipe envolvida. As alterações tendem a tratar de correção de defeitos encontrados ou mudanças de funcionalidades e devem ser atribuídas a um ou mais desenvolvedores.

Por conta do aumento do uso de sistemas em diferentes contextos, o desenvolvimento de software se torna cada vez mais uma tarefa complexa e colaborativa (MOHTASHAMI; MARLOWE; KU, 2011), envolvendo a coordenação de profissionais com diferentes características de trabalho. A coordenação dessa atividade deve ser feita de forma que se mantenha a qualidade do software em si, pois o impacto de falhas e manutenções frequentes podem ter consequências graves ao atingir potencialmente um grande número de pessoas. Dentre as ferramentas utilizadas para a coordenação do desenvolvimento colaborativo estão os sistemas ou plataformas de gestão de defeitos como o Bugzilla [1] e o Redmine [2]. Ferramentas de gestão de defeitos, em geral, possuem um ciclo de resolução de requisições conforme apresentado na Figura 1.

[1] https://www.bugzilla.org/
[2] https://www.redmine.org/

Figura 1 – Ciclo de vida de uma **Change Request**



Fonte: (RAKHA; BEZEMER; HASSAN, 2017)(traduzido)

O ciclo se inicia quando uma nova Change Request é criada. Esta é composta de elementos textuais que descrevem a alteração requisitada. Após a sua criação ocorre o processo de triagem, onde ela é atribuída a um ou mais desenvolvedores apropriados para a sua resolução. O que caracteriza um desenvolvedor como o mais apropriado para ser o atribuído (*Assignee*), varia na literatura e nos ambientes de desenvolvimento (CAVALCANTI et al., 2016). A escolha do *Assignee* é fundamental, dado que cada alteração do projeto pode afetar aspectos da qualidade do código do projeto, além de custar recursos e tempo (AKTAş; YILMAZ, 2020) (NAGUIB et al., 2013).

Após o encerramento da triagem, o desenvolvedor atribuído analisa a alteração e gera valores de resolução referentes ao estado da requisição. A resolução *FIXED* indica que a alteração foi aceita e resolvida, enquanto *WONTFIX* significa que o atribuído não é capaz de realizar a alteração e *INVALID* que a alteração não é pertinente. O valor *DUPLICATE* aponta que uma requisição muito semelhante ou idêntica já existe na plataforma e *WORKSFORME* significa que o problema descrito na Change Request não foi possível de reproduzir, podendo indicar que o problema está no ambiente do autor. A partir da criação de um valor de resolução, a requisição é considerada resolvida e deve ser verificada na próxima etapa. Uma Change Request pode ser resolvida e não necessariamente gerar alteração de código.

A última etapa do ciclo é referente a verificação do trabalho realizado pelo *Assignee* verificando o valor da resolução gerado e a alteração requisitada. Caso os colaboradores encarregados de verificar não estejam satisfeitos com o trabalho realizado, a requisição

pode ser re-aberta e atribuída a outro desenvolvedor, iniciando o ciclo novamente. Caso a verificação seja positiva, o processo é encerrado. Todo esse processo e as interações entre os diferentes colaboradores são realizadas e estão registradas nas ferramentas de gestão de erros. Isto faz com que elas sejam uma rica fonte de dados sobre o histórico de desenvolvimento de pessoas em ambientes colaborativos, permitindo a criação de perfis de desenvolvedores, como será mostrado nos próximos capítulos.

A realização manual do processo de triagem por parte de uma equipe pode não ser viável, uma vez que é algo que consome tempo e recursos. De acordo com o tamanho de um projeto, o número de requisições pode chegar a dezenas de milhares, como acontece no repositório do *Elastic* [3]. Em projetos onde não é possível a realização manual da recomendação de desenvolvedores, alguma abordagem automatizada pode ser empregada. A partir da extração de características da requisição alvo, os métodos encontrados na literatura tendem a atribuir a requisição alvo utilizando duas abordagens: a de que o desenvolvedor mais apropriado é o que tem mais experiência com requisições similares (ZHANG et al., 2016) (RAHMAN; RUHE; ZIMMERMANN, 2009), ou a de que o desenvolvedor mais apropriado pode ser encontrado além da análise de requisições passadas, mas também a partir da mineração de dados dos repositórios de códigos (CAVALCANTI et al., 2016) (SUN et al., 2017) (KAGDI et al., 2012) e/ou de defeitos. O método apresentado neste trabalho segue a segunda abordagem. Ao levar-se em conta somente o nível de experiência de desenvolvedores como fator para a recomendação de requisições, põe-se em risco a qualidade do código, uma vez que não é avaliado o impacto da alteração realizada. Também pode ocorrer o desbalanço de trabalho entre desenvolvedores com níveis de experiência diferentes, como no modelo de (ZHANG et al., 2016). No modelo descrito neste documento, o desenvolvedor com o melhor histórico de características de trabalho na resolução de requisições similares à uma requisição não resolvida é recomendado, independentemente do tamanho de seu histórico.

O conceito de modelagem de usuários tem sido feito cada vez mais parte dos produtos de software criados no cenário atual, onde a alta escalabilidade de sistemas implica na geração de grandes volumes de dados (ABDEL-HAFEZ; XU, 2013). Isso faz com que a personalização de sistemas, baseada em perfis de usuários, seja necessária em diferentes aplicações como: sistemas de recomendação (INZUNZA; JUÁREZ-RAMÍREZ; JIMÉNEZ, 2017), sistemas de ensino não-presencial (CILOGLUGIL; INCEOGLU, 2012), redes sociais (ABDEL-HAFEZ; XU, 2013) e sistemas adaptativos.

Na tarefa de desenvolvimento de software, é possível a utilização de modelagem para representar características comportamentais ou pessoais de desenvolvedores. Essas características tendem a mudar durante o ciclo de desenvolvimento e modelos que representam alterações durante períodos de tempo são chamados de dinâmicos. As características

[3] https://github.com/elastic/elasticsearch/

objetivas que compõem modelos dinâmicos de desenvolvedores estão disponíveis para serem obtidas por meio da mineração de repositórios. A modelagem de desenvolvedores proporciona a formação de sistemas de recomendação de desenvolvedores.

Para sistemas de recomendação, a representação objetiva de características de itens a serem recomendados e usuários é essencial. Porém, no contexto do desenvolvimento de software, a definição de características que representem o desenvolvedor e itens é difícil de ser realizada e pode variar de acordo com equipes e projetos (RAWAT; MITTAL; DUBEY, 2012) (IHARA; OHIRA; MATSUMOTO, 2009). Com a análise de métricas de qualidade de código, extraídas de repositórios (ANDERSON, 2004), é possível criar métodos automatizados para construção de perfis dinâmicos de desenvolvedores que permitam, por meio de análise automática das métricas, inferir sobre o nível de habilidades de desenvolvedores. (BEAL, 2014) obteve êxito no desenvolvimento de modelos de classificação de desenvolvedores por meio da extração de valores de métricas durante o processo de desenvolvimento de um mesmo artefato, classificando os participantes em: Desenvolvedor Júnior, Pleno e Sênior, com acurácia de cerca de 90%. Além da possibilidade de classificar os desenvolvedores somente segundo à qualidade de seu código, é possível também, a partir da extração e análise de dados históricos de repositórios de código-fonte, obter informações que representem o seu comportamento durante o desenvolvimento.

1.1 Motivação

O desenvolvimento colaborativo de software é uma atividade realizada por múltiplos participantes com diferentes experiências, perícias e comportamentos, e que exercem papéis variados no desenvolvimento de um projeto (BASSI et al., 2018). As especificações de um projeto são artefatos importantes que auxiliam na coordenação do desenvolvimento ao fornecer um guia para os desenvolvedores de como o software deve se comportar e quais funcionalidades são esperadas dele uma vez que esteja concluído. Entretanto, durante o ciclo de desenvolvimento, as especificações de um projeto podem sofrer alterações devido a evolução de questões como demandas de usuários e visão do projeto (ALI; LAI, 2016).

A atribuição de requisições de alterações para desenvolvedores apropriados é uma etapa essencial do processo de desenvolvimento, uma vez que cada alteração do projeto possui um impacto que pode afetar aspectos da qualidade do código do projeto, como manutenibilidade e testabilidade, por exemplo (LOCK; KOTONYA, 1999). Para a efetiva manutenção e garantia de qualidade de tais processos de desenvolvimento colaborativo, é necessária uma análise precisa das características de desenvolvimento dos participantes, dado que elas são resultado direto de seu trabalho (FENTON; NEIL, 2000). Essa avaliação é frequentemente realizada com critérios subjetivos que podem não refletir a real habilidade do desenvolvedor ou o ambiente no qual esse se encontra.

O trabalho de (CAVALCANTI et al., 2016), realiza uma pesquisa com profissionais atuantes no mercado que ocupam, em sua maioria, posições de gerência em times de desenvolvimento, e os resultados indicam que as atribuições de alterações são realizadas por meio de heurísticas subjetivas, suportando sua abordagem semi-automatizada.

Dentro das abordagens automatizadas, também existem trabalhos que usam somente experiência como fator determinante para a recomendação de desenvolvedores, como (ZHANG et al., 2016) e (KAGDI et al., 2012). A recomendação de desenvolvedores para alterações sem considerar características de suas alterações passadas pode ter consequências negativas para a qualidade do código de um projeto. Isso é ainda mais grave em ambientes colaborativos com um grande número de desenvolvedores e requisições.

Apesar da literatura apresentar abordagens direcionadas a resolver o problema da etapa de atribuição automatizada de requisições para desenvolvedores durante o processo de desenvolvimento, não existe um método capaz de utilizar técnicas de *Modelagem de Usuário* para criar perfis de desenvolvedores, baseado não somente em características de sua perícia e experiência, mas também em métricas de qualidade de código, mantendo a qualidade geral do código-fonte do projeto. Assim, dentro desse contexto, a criação de tal método é a principal motivação deste trabalho.

1.2 Objetivo

Esta pesquisa tem como objetivo desenvolver um método capaz de atribuir automaticamente requisições de alterações de código para desenvolvedores, a partir de seu perfil histórico de desenvolvimento.

Os objetivos específicos compreendem:

- Construir um modelo de desenvolvedor com base em dados históricos sobre o processo de triagem, por meio de aquisição implícita de dados;
- Gerar uma base de dados referentes à *commits*, qualidade de código e requisições de alterações;
- Implementar um protótipo do método para validação e avaliação do método proposto.

1.3 Hipóteses de Trabalho

As hipóteses desta pesquisa são:

- H1) O método proposto recomenda desenvolvedores tão efetivamente quanto métodos presentes na literatura quando aplicados em bases de código-livre

- H2) O método proposto recomenda mais desenvolvedores com impacto positivo na qualidade do código de um repositório comparado com métodos presentes na literatura

1.4 Contribuições Científicas e Tecnológicas

Em suma as contribuições chave desta pesquisa são:

1. Criação de uma base de perfis de desenvolvedores, a partir de dados de *commits*, qualidade de código e requisições de alterações de repositórios de código-livre.
2. Realização de testes comparativos e estatísticos entre o método de recomendação proposto e a implementação do *iMacPro*(HOSSEN; KAGDI; POSHYVANYK, 2014).
3. Desenvolvimento de uma ferramenta, que auxilia a visualização do impacto de desenvolvedores na qualidade do código e fornece sugestões de alterações em componentes de código com baixa qualidade (SILVA et al., 2019)

1.5 Escopo do Trabalho

Este projeto de pesquisa se resume a proposição de um modelo de recomendação automática de desenvolvedores para a resolução de alterações de software em ambiente colaborativo, por meio da modelagem de usuários baseado em dados históricos relacionados ao seu processo de desenvolvimento e a qualidade de seu código produzido. Para o escopo do trabalho, o método proposto é aplicado somente em bases de código desenvolvidos no paradigma de programação orientado a objetos. Apesar de não ser o mais eficiente em questão de performance (CHATZIGEORGIOU; STEPHANIDES, 2002), o paradigma de programação orientado a objetos é o mais amplamente utilizado em ambientes acadêmicos e profissionais (MITCHELL, 2001), isso faz com que haja uma grande disponibilidade de recursos que facilitam a análise e trabalho com repositórios de orientação a objetos.

1.6 Organização

Este documento é composto por sete capítulos. O primeiro capítulo é composto pelas considerações iniciais, que introduzem aspectos importantes para a contextualização do trabalho, pela motivação que dá base para a construção da proposta do trabalho, os objetivos, as contribuições científicas esperadas e pelas hipóteses de trabalho que se deseja verificar, além do escopo do trabalho.

O Capítulo 2 apresenta a fundamentação teórica do trabalho, que aborda conceitos técnicos necessários para o embasamento teórico da proposta apresentada, como Modelagem

de Usuários e criação de Perfis, e conceito e técnicas de Modelos de Recomendação. Também são apresentados os conceitos de Qualidade de Software e Medidas e Métricas. Ainda são apresentados os conceitos e técnicas de áreas que darão suporte para a Modelagem de Usuários como a Recuperação da Informação e Processamento de Linguagem Natural, que serão utilizados nas análises dos textos das Requisições de Alteração.

O Capítulo 3 apresenta os principais trabalhos levantados da literatura, com propostas relacionadas à este trabalho de pesquisa. E o conjunto de termos relativos ao tema proposto por este trabalho, utilizados para a criação de strings de busca. Esse capítulo foi dividido em duas sessões. A primeira seção trata dos trabalhos relacionados à Modelagem de Desenvolvedores e a segunda seção traz trabalhos referentes à Recomendação de Desenvolvedores.

O Capítulo 4 apresenta o método proposto de recomendação de desenvolvedores, baseado em Modelagem de Usuário.

O Capítulo 5 apresenta os procedimentos metodológicos, que descrevem as ferramentas utilizadas na implementação do método, a base de dados e os experimentos que serão realizados através do método.

O Capítulo 6 apresenta os resultados preliminares obtidos com a pesquisa até o momento.

E por fim, no Capítulo 7 do documento, são apresentadas as considerações finais e o cronograma dos próximos passos a serem feitos.

2 Fundamentação Teórica

Para o desenvolvimento deste trabalho foi necessário o uso de técnicas e conceitos de diferentes áreas da computação. Neste capítulo serão apresentados os principais fundamentos teóricos necessários para o embasamento do método descrito no Capítulo 5.

As seções a seguir tratam sobre Modelagem de Usuários, Qualidade de Código, Modelos de Recomendação, Processamento de Linguagem Natural e Recuperação da Informação.

2.1 Modelagem de Usuários

A Modelagem de usuários é uma sub-área da Interação Humano-Computador (IHC), que abrange o processo de construir e modificar estruturas de dados pessoais que caracterizem usuários dentro de um sistema, afim de prover uma interação personalizada entre pessoas e sistemas (FISCHER, 2001). Para (BISWAS, 2012), é uma representação do conhecimento e preferências de usuários. Para (RICH, 1983), pode relacionar uma grande variedade de conhecimentos sobre usuários. Perfis de usuário são as instâncias dos dados estruturados em um modelo (PIAO; BRESLIN, 2018).

Existem diferentes abordagens para a modelagem de usuário:

- A modelagem estática, que é a criação de um modelo composto por características fixas (comportamentais ou pessoais) que não se alteram ao longo do tempo, como nome e registro geral. Nessa abordagem, as mudanças de comportamento do usuário não são registradas;
- A modelagem dinâmica, onde as características do usuário se modificam ao longo do tempo, conforme o usuário interage com o sistema e mais dados são coletados (WANDERLEY et al., 2012);
- A modelagem baseada em estereótipos, onde os mesmos são criados a partir de estatísticas gerais de perfis estáticos. Com base nas informações coletadas, os usuários são classificados em estereótipos comuns, permitindo previsões sobre um usuário, mesmo que haja pouca informação particular(ALNANIH; ORMANDJIEVA; RADHAKRISHNAN, 2013), a partir de novos dados é possível a atualização dos esteriótipos;
- E por fim, o modelo altamente adaptativo de usuário, onde a abordagem representa a forma extrema de personalização que necessita do máximo de informações possíveis para gerar perfis específicos para cada usuário.

Um exemplo de uma aplicação baseada em modelos estáticos de usuário pode ser o site de uma locadora de veículos que aluga os veículos em relação à carteira de motorista do cliente. Caso o usuário precise modificar uma das suas preferências, ele deverá editá-la manualmente. Um exemplo de perfil dinâmico é o a plataforma *Netflix*, que molda as listas de filme indicados de acordo com os filmes assistidos por seus usuários ao longo do tempo, uma vez que os gostos dos usuários podem variar. O perfil do desenvolvedor proposto por este trabalho é dinâmico, uma vez que ele é composto por grupos e a disposição desses grupos pode mudar conforme novas requisições de alteração são resolvidas em um repositório. Este perfil é construído a partir de dados históricos colhidos, de forma não-invasiva, sobre o trabalho de programação de desenvolvedores. O perfil pode mudar conforme novos dados são extraídos de repositórios, uma vez que em uma janela maior de tempo, deve-se levar em conta que o comportamento de desenvolvedores muda, eles podem adquirir novos expertises ou experiências que mudam a sua capacidade técnica e consequentemente seu perfil de desenvolvimento deve refletir isso.

O processo de construção de modelos de usuários é composto de diversas etapas. Segundo (BARTH, 2010), o processo é organizado nas seguintes etapas: Composição do modelo, Representação, Aquisição, Aprendizado e Manutenção do Modelo. Cada etapa por sua vez é caracterizada pela aplicação de técnicas e métodos específicos, conforme descrito a seguir.

2.1.1 Composição do modelo

Para a composição de um modelo de usuário é preciso identificar quais dados são relevantes para a caracterização de aspectos do usuário, sejam comportamentais ou pessoais. Essas características são definidas de acordo com a utilização do modelo. Se o modelo de usuário será utilizado para fazer recomendações de veículos, o modelo deverá conter dados sobre o tipo do veículo que o usuário é habilitado a dirigir, por exemplo.

2.1.2 Representação do modelo

O modelo de usuário pode ser representado por diversas formas. Em (BARTH, 2010), são descritos exemplos de representações baseadas em dados históricos, vetores e conjunto de regras. As representações baseadas em dados históricos utilizam dados coletados ao longo do tempo sobre os usuários para compor o modelo. As representações baseadas em vetores, consideram o modelo como um vetor de características, onde cada característica possui um valor associado a algum aspecto do usuário. Os modelos baseados em conjunto de regras, podem ser representados por árvores de decisão (HALL, 2005) ou por regras em uma linguagem como Prolog (STERLING; SHAPIRO, 1994).

Qualquer tipo de formato que estruture dados pode ser utilizado na representação de

modelos de usuários. A representação em formato XML[1] é favorável para a representação de modelos dinâmicos, uma vez esse formato fornece uma estruturação flexível com capacidade de adição ou remoção de elementos com facilidade e, por isso, é compatível com modelos onde características podem mudar ao longo do tempo.

O modelo apresentado na Listagem 1, indica a intensidade do interesse de um usuário em diferentes gêneros de música. Este modelo é dinâmico, pois o interesse do usuário pode mudar com o passar do tempo. Este modelo poderia ser utilizado para a criação de um sistema de recomendação de músicas, por exemplo.

```
<USUARIO>
<!-- Gêneros de música e valores de interesse para o usuário -->
 <NOME>Matheus Camilo</NOME>
 <ROCK>4</ROCK>
 <POP>2</POP>
 <SERTANEJO>0</SERTANEJO>
<USUARIO/>
```

Listing 1 – Exemplo de representação de modelo de usuário em XML(Fonte: Autor)

2.1.3 Aquisição do modelo

Em relação à aquisição de dados para os modelos de usuários, esta pode ser feita por meio de métodos implícitos ou explícitos. Os métodos explícitos conseguem adquirir dados estáticos através do preenchimento de formulários ou questionários, realizado pelo usuário. Em contrapartida, os métodos implícitos, obtém os dados dinâmicos de maneira automatizada, sem a necessidade de auxílio humano. Os dados são obtidos por meio de sensores de coletas não invasiva (PAPATHEODOROU, 1999) (BEAL; BASSI; PARAISO, 2017). Os dados obtidos pelo método implícito formam um conjunto de exemplos que podem ser utilizados para o aprendizado do modelo.

2.1.4 Aprendizado do modelo

Nesta etapa, são aplicados os algoritmos de Aprendizagem de Máquina, eles são utilizados no intuito de classificar ou agrupar usuários (SCHWAB; KOBSA, 2002) que possuem alguma relação em suas características, gerando os perfis de usuários. Ou seja, ao rotular ou agrupar as instâncias de usuários, o algoritmo estará atribuindo perfis a estes usuários. Os algoritmos de Aprendizagem de Máquina induzem o modelo do usuário a partir de um conjunto de exemplos com dados sobre o contexto de uma aplicação. São exemplos de técnicas de aprendizagem de máquina que podem ser utilizadas na indução do modelo do usuário, a Árvores de Decisão e SVM (NGUYEN, 2009), Naïve Bayes (WEN;

[1] http://www.w3.org/TR/REC-xml/

FANG; GUAN, 2008), Redes Neurais Artificiais (CHEN; NORCIO, 1991) e k-NN(HWANG; WEN, 1998).

2.1.5 Manutenção do modelo

As características dos usuários (comportamentais e pessoais) e do sistema podem mudar ao longo do tempo, isso faz com que seja necessário a realização da manutenção do modelo do usuário. No âmbito da manutenção do modelo, tanto o método explícito quanto o implícito podem ser utilizados.

No método explícito o próprio usuário realiza as alterações em seus dados manualmente, por meio do acesso à edição de seu perfil. Entretanto, esse método exige que o esforço de construção e manutenção do modelo seja posta sobre o usuário. Entretanto depender de usuários para a manutenção do modelo pode ter impactos negativos, uma vez que usuários nem sempre fazem as modificações corretamente ou sequer estão dispostos a fazê-las. No método implícito, o sistema é encarregado de fazer de forma automática a manutenção do modelo, com base na obtenção de novos dados sobre as interações de usuários com o sistema. A manutenção dos modelos dinâmicos envolve algoritmos de aprendizagem de máquina e necessita que estes sejam retreinados constantemente, a não ser em modelos adaptativos.

2.2 Qualidade de Código

O conceito de qualidade de software e como avaliá-lo é discutido em diversos trabalhos da literatura. Uma definição popular é encontrada em (ISO/IEC, 2016), onde qualidade é dada como a combinação de fatores como funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. Neste sentido, a qualidade de um software depende de diferentes fatores e de diferentes pontos de vista. As características que um usuário utiliza para avaliar a qualidade do produto de software podem ser diferentes das utilizadas por um desenvolvedor, por exemplo. Segundo (MCCONNELL, 2004), as qualidades de software podem ser dividas em Qualidade Externa e Qualidade Interna.

Qualidade Externa é relacionada à parte operacional de um software onde a sua qualidade está relacionada com as percepções de valor dadas por usuários ao interagir com o software. Esse tipo de qualidade pode ser medida por meio de testes de recursos, controle de qualidade e comentários de usuários. Esta é a qualidade que afeta usuários de uma maneira direta, ao contrário da Qualidade Interna que os afeta indiretamente.

Qualidade Interna é relacionada à parte de construção de um software onde seus elementos não são aparentes aos seus usuários e seus aspectos de qualidade podem ser medidos por métricas de código, que permitem o calculo quantitativo do nível de Qualidade Interna de um produto de software.

2.2.1 Métricas de Qualidade de Código Orientado à Objetos

O processo de avaliação da qualidade de um software é um processo árduo uma vez que projetos de software geralmente continuam a crescer e a evoluir para atender novas demandas enquanto estes estiverem em uso. Neste contexto, a avaliação da qualidade deve ser feita não somente sobre um software em seu estado atual, mas também levando em consideração como o mesmo se comporta com as possíveis mudanças que serão feitas.

Métricas de Código auxiliam o controle da qualidade de software funcionam como indicadores da Qualidade Interna de um software e auxiliam na manutenção da qualidade ao servirem de referência às ações desenvolvedores devem tomar (SINGH, 2013). Para que sejam úteis as métricas devem ser compreensíveis e bem estabelecidas. Diversas métricas diferentes relacionadas à Qualidade Interna do software foram desenvolvidas e são largamente utilizadas (RAWAT; MITTAL; DUBEY, 2012).

Existem diversos paradigmas de programação, um dos mais relevantes considerando o cenário atual é o paradigma de orientação a objetos. O software construído a partir desse paradigma é baseado nos conceitos de classe, métodos, encapsulamento, herança e polimorfismo. Para esse tipo de produto, é possível a aplicação de métricas de código-fonte, que refletem aspectos do código como complexidade, manutenibilidade, reusabilidade e testabilidade. Estas medidas tem como objetivo permitir o controle de complexidade das estruturas, melhorar a manutenibilidade e o desenvolvimento dos códigos-fonte e consequentemente o próprio software (YU; ZHOU, 2010).

Desde a década de 70 é praticado o cálculo de métricas com a introdução do cálculo da Complexidade Ciclomática de McCabe (CCM)(MCCABE, 1976). Atualmente existem ferramentas que permitem o cálculo de métricas de código fonte em um projeto, como Sonar[2], Metrics[3] para o Eclipse[4], Understand[5],inclusive com uma interface visual.

As métricas de código-fonte mais citadas na literatura atualmente podem ser agrupadas nas categorias: Métricas Básicas, Métricas de Complexidade, Métricas de CK e Métricas de Acoplamento (FILÓ; BIGONHA; FERREIRA, 2015). Nesse agrupamento, cada métrica pertence somente a um grupo. O agrupamento apresentado tem o intuito ilustrativo e pode ser realizado de diferentes maneiras. As métricas apresentadas e descritas em Quadro 1, e suas influências na qualidade do código serão discutidas na próxima sub-seção.

[2] https://www.sonarqube.org/
[3] http://metrics2.sourceforge.net/
[4] https://www.eclipse.org/ide/
[5] See https://scitools.com/features/

Quadro 1 – Métricas e suas definições

Métrica	**Definição**	**Grupo**
Complexidade Ciclomática de McCabe (VG)	Mede a quantidade de caminhos lógicos, linearmente independentes, de um código-fonte.	Complexidade
Número de Parâmetros (PAR)	Calcula a quantidade de parâmetros passados em um método.	Tamanho
Profundidade de Blocos Aninhados (NBD)	Calcula a profundidade de blocos de instruções aninhados.	Complexidade
Acoplamento Aferente (Ca)	É o número de classes de outros pacotes que dependem de classes de um pacote considerado.	Acoplamento
Acoplamento Eferente (Ce)	É o número de classes de um pacote que dependem de classes de outros pacotes.	Acoplamento
Instabilidade (RMI)	Um pacote é instável quando ele depende de muitos outros pacotes.	Acoplamento
Abstração (RMA)	Mede o número de classes abstratas e interfaces, dividido pelo número total de tipos em um pacote.	Acoplamento
Distância Normalizada (RMD)	Relação entre Abstração e Instabilidade.	Acoplamento
Profundidade da Árvore de Herança (DIT)	Mede a quantidade de classes ancestrais que antecedem uma classe numa árvore de herança.	Herança
Métodos Ponderados por Classe (WMC)	É o somatório das complexidades de cada método de uma classe	Complexidade
Número de Filhos (NSC)	Mede o número de subclasses imediatas de uma classe em uma árvore de herança.	Tamanho
Métodos Sobrescritos (NORM)	Mede o número de métodos sobrescritos.	Herança
Falta de Coesão em Métodos (LCOM)	Mede a coesão entre os métodos de uma classe.	Complexidade
Número de Atributos por Classe (NOF)	Mede o número de atributos de uma classe.	Tamanho

Continua

Continuação

Métrica	Definição	Grupo
Número de Atributos Estáticos (NSF)	Calcula a quantidade de atributos estáticos de uma classe.	Tamanho
Número de Métodos (NOM)	Número de Métodos por classe	Tamanho
Número de Métodos Estáticos (NSM)	Calcula a quantidade de métodos estáticos de uma classe.	Tamanho
Índice de Especialização (SIX)	Calcula o índice de especialização de cada subclasse.	Herança
Número de Classes (NOC)	Indica o número de classes por pacote	Herança
Número de Interfaces (NOI)	Indica a quantidade de interfaces	Tamanho
Número de Pacotes (NOP)	Indica a quantidade de pacotes por projeto	Tamanho
Total de Linhas de Código (TLOC)	Indica o total de linhas de código de um projeto	Tamanho
Linhas de Código por Métodos (MLOC)	Calcula o número de linhas não-nulas de métodos.	Tamanho

Fonte: o autor 2020

2.2.1.1 Influência em Parâmetros de Qualidade

Estudos mostram que métricas de código fonte são fortes indicadores de parâmetros de qualidade relacionados a manutenibilidade, testabilidade, reusabilidade e complexidade (FILÓ; BIGONHA; FERREIRA, 2015), (SILVA; KODAGODA; PERERA, 2012), (LI, 2008), (OLIVEIRA et al., 2008), (OLAGUE; ETZKORN; COX, 2006), (ANDERSON, 2004), (HORSTMAN, 2002) (HENDERSON-SELLERS, 1995) e (CHIDAMBER; KEMERER, 1994).

A complexidade de um código está diretamente ligado à compreensão ou legibilidade de um programa por participantes do processo de desenvolvimento. Um código altamente complexo acaba gerando custos de tempo e esforço durante o processo de desenvolvimento (MISRA; AKMAN, 2008).

A testabilidade de software é referente à capacidade de um produto de software ou componente de ser testado. O processo de testes de software é a etapa em que diferentes aspectos da funcionalidade de um programa são verificados com a intensão de encontrar erros e atualmente é uma das técnicas mais utilizadas para avaliar a validade da implementação do sistema (GAROUSI; FELDERER; KıLıçASLAN, 2019). Diversas métricas têm sido propostas para prever e melhorar a capacidade da testabilidade de softwares, especialmente no nível de código-fonte, uma vez que ajudam os testadores a detectar partes do sistema que possivelmente sejam difíceis de testar, permitindo que se organize e planeje previamente o trabalho de teste (GAROUSI; FELDERER; KıLıçASLAN, 2019) e (BRUNTINK; DEURSEN, 2004).

Reusabilidade é o grau no qual um componente de software pode ser aplicado em diferentes partes de um projeto. Um alto nível de reusabilidade é benéfico pois reduz o custo de desenvolvimento do software e indica um produto modularizado. O reuso de software está intimamente conectado à produtividade, qualidade e confiabilidade, de um produto. As métricas de código podem ser utilizadas para determinar fatores de qualidade que afetam a reusabilidade, uma vez que componentes tem certas características que tendem a facilitar ou não seu reuso (WASHIZAKI; YAMAMOTO; FUKAZAWA, 2003).

A manutenibilidade de software é referente ao nível de facilidade com a qual um sistema de software ou componente pode ser modificado tendo em vista a correção de falhas, melhoria de desempenho e outros atributos, ou de adaptação a um ambiente alterado. A manutenibilidade é um aspecto importante para a avaliação de qualidade de software e tem grande influência sobre custo no ciclo de vida de desenvolvimento do software. As pesquisas relacionadas à previsão de manutenibilidade de software incluem o uso de fatores mensuráveis que tenha efetiva influencia na atividade de manutenção de software (BHATT et al., 2006).

2.2.1.2 Intervalo de Risco

Afim de se inferir sobre a qualidade de um software baseado nos valores de métricas de código, é necessário que se estabeleça intervalos para cada métrica que reflitam diferentes qualidades de código. Diversos trabalhos se dedicaram à essa tarefa. Em (ANDERSON, 2004), é apresentado os intervalos da Complexidade Ciclomática de McCabe. Também é encontrado os intervalos para WMC (OLAGUE; ETZKORN; COX, 2006), LCOM (HORSTMAN, 2002), DIT(HENDERSON-SELLERS, 1995), RFC(CHIDAMBER; KEMERER, 1991), CBO(LI, 2008). Em (FILÓ; BIGONHA; FERREIRA, 2015) são apresentados os intervalos de NBD, NSC, NORM, SIX, MLOC, NOF, NSF, NSM, PAR, NOC, Ca e Ce. Os intervalos das métricas mais relevantes para esse trabalho são apresentados em Quadro 2

Quadro 2 – Métricas e seus intervalos

Métrica	Intervalo	Valores	Fonte
Profundidade de Blocos Aninhados (NBD)	**Ruim :** **Regular :** **Bom :**	$NBD > 3$ $3 \geq NBD > 1$ $1 \geq NBD$	(FILÓ; BIGONHA; FERREIRA, 2015)
Métodos Ponderados por Classe (WMC)	**Ruim :** **Regular :** **Bom :**	$WMC > 100$ $100 \geq WMC \geq 21$ $20 \geq WMC \geq 1$	(OLAGUE; ETZKORN; COX, 2006)
Falta de Coesão em Métodos (LCOM)	**Ruim :** **Regular :** **Bom :**	$LCOM > 1$ $1 \geq LCOM > 0.5$ $0.5 \geq LCOM \geq 0$	(HORSTMAN, 2002)
Número de Métodos Estáticos (NSM)	**Ruim :** **Regular :** **Bom :**	$NSM > 3$ $3 \geq NSM > 1$ $1 \geq NSM$	(FILÓ; BIGONHA; FERREIRA, 2015)
Linhas de Código por Método (MLOC)	**Ruim :** **Regular :** **Bom :**	$MLOC > 30$ $30 \geq MLOC > 10$ $10 \geq MLOC$	(FILÓ; BIGONHA; FERREIRA, 2015)
Acoplamento Entre Objetos (CBO)	**Ruim :** **Bom :**	$CBO > 5$ $5 \geq CBO \geq 0$	fonte
Profundidadeda Árvore de Herança (DIT)	**Ruim :** **Bom :**	$DIT > 5$ $5 \geq DIT \geq 1$	(HENDERSON-SELLERS, 1995)
Resposta por Classe (RFC)	**Ruim :** **Bom :**	$RFC > 50$ $50 \geq RFC \geq 0$	Fonte

2.3 Modelos de Recomendação

Os modelos de recomendação têm como objetivo a remoção de informações redundantes ou irrelevantes de um grande conjunto de dados (Filtragem). Os modelos de recomendação contribuem com essa tarefa pois inferem sobre a *relevância* de informações de itens ao associá-las a usuários(LU; GUO, 2018).

A partir da perspectiva de sistemas de Recuperação de Informação, os usuários interagem apenas com “O universo do conhecimento registrado” ou seja somente com os dados presentes dentro do sistema. Nesse contexto, o computador é apenas uma ferramenta usada para interagir com este universo de dados. A questão da relevância é, segundo (HJøRLAND, 2010), principalmente sobre entender a relação entre as "necessidades do usuário"e todo o universo de informação presente em um sistema.

Dentro de sistemas de recomendação, o termo *itens* representam as informações sobre algo que será recomendado e o termo *usuário* se refere ao que receberá as recomendações. É comum o uso de modelagem de usuários na tarefa de associação, uma vez que permitem a representação de usuários em um sistema de recomendação, o que pode facilitar a inferência sobre quais informações são relevantes a estes usuários.

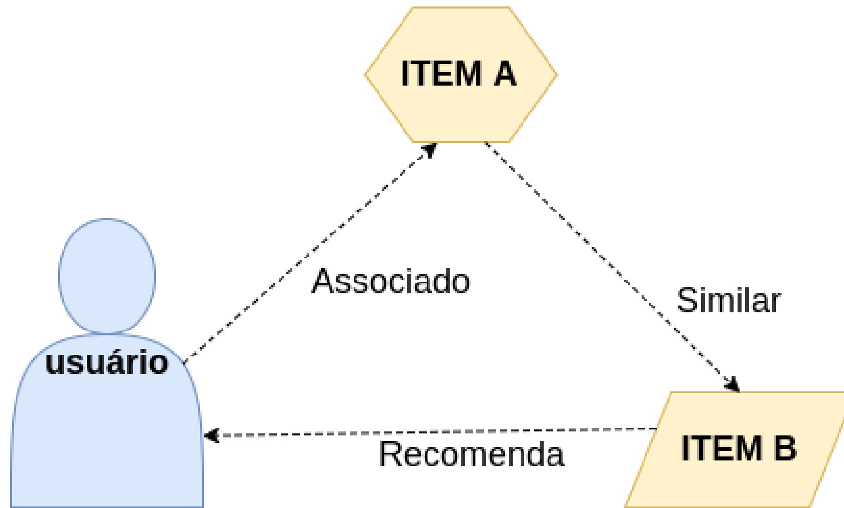
Há na literatura métricas que podem ser usadas na medição da *relevância* de Modelos de Recomendação, como *Precision*Equação 2.2 e *Recall*Equação 2.1. Nesse contexto, a aplicação dessas métricas é feita a partir de um conjunto de itens recuperados (por exemplo, uma lista de imagens produzida por uma consulta em uma plataforma de busca) e um conjunto de itens relevantes (por exemplo, a lista das imagens relevantes ao usuário que fez a consulta na plataforma de busca).

$$Precision = \frac{|Relevantes \cap Recuperados|}{|Recuperados|} \tag{2.1}$$

$$Recall = \frac{|Relevantes \cap Recuperados|}{|Relevantes|} \tag{2.2}$$

Existem basicamente duas abordagens para a tarefa de associação de itens e usuários: a filtragem baseada em conteúdo (em inglês, Content-Based Filtering)(PAZZANI; BILLSUS, 2007) e a filtragem baseada em colaboração (em inglês, Collaborative Filtering)(SARWAR et al., 2001). A filtragem baseada em conteúdo funciona com perfis existentes de usuários. Os perfis tem informações pessoais e comportamentais de usuários. Nessa abordagem as recomendações de itens para usuários são realizadas através da semelhança de um item com outro já associado com o usuário, usando do princípio de "Se o usuário é associado com o item A, e A é semelhante a B, então B é recomendado para o usuário", como é apresentado em Figura 2.

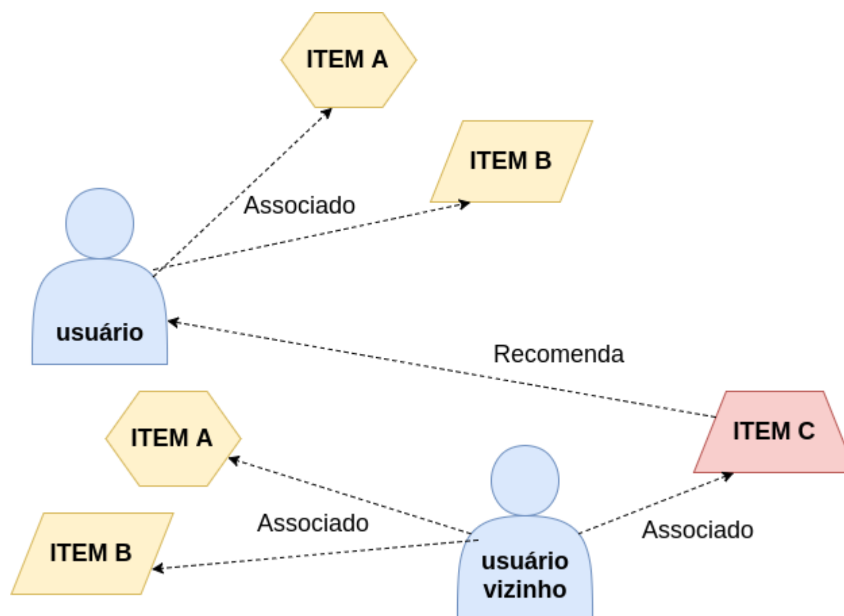
Figura 2 – Filtragem Baseada em Conteúdo



Fonte: o autor 2020

Na filtragem colaborativa, os itens são recomendados a usuários baseado nas associações de outros usuários que partilham de similaridades, os chamados de usuários vizinhos. Essa abordagem usa o princípio de "Se o usuário tem associações com itens similares a um usuário vizinho, então os itens do vizinho podem ser recomendados para o usuário", como é apresentado em Figura 3.

Figura 3 – Filtragem Baseada em Colaboração



Fonte: o autor 2020

2.4 Processamento de Linguagem Natural e Recuperação de Informação

2.4.1 Processamento de Linguagem Natural

Processamento de linguagem Natural (PLN) é a área da Inteligência Artificial que aborda computacionalmente a comunicação humana, considerando formatos e referências, estruturas e significados, contextos e usos. Pode ser definido como o conjunto de técnicas e métodos utilizados para a análise de textos desestruturados em um ou mais níveis linguísticos, com o proposito de simular o processamento humano da língua (RUSSELL; NORVIG, 2016).

A relevância de processamento de linguagem natural pode ser traçada até os anos 50, quando Alan Turing (TURING, 2004) estabeleceu um teste para a avaliação da capacidade de máquinas exibirem comportamento inteligente indistinguível de um ser humano, a inteligência da máquina era avaliada de acordo com a sua habilidade de comunicação em linguagem natural. Na mesma década foi publicado um livro pelo linguista Noam Chomsky (CHOMSKY, 2002) que descreve um sistema baseado em regras sobre como estruturar sentenças gramaticalmente corretas. O que inspirou a criação de muitas técnicas de PLN que processam linguagem através de sequências de regras programadas, criação de gramáticas ou regras heurísticas.

Entretanto o uso estrito dessas técnicas diminuiu com o aumento do poder computacional das máquinas e os avanços da área de Aprendizagem de Máquina, que utiliza de inferência estatística para reconhecer automaticamente padrões em *corpus* com grande volume de conteúdo. Atualmente, diversos algoritmos de aprendizagem de máquina são aplicados para tarefa de PLN.

2.4.2 Recuperação de Informação

Recuperação da Informação (RI) é a área da computação que aborda o armazenamento e a recuperação automática de informações de documentos. A RI aplica catalogação, categorização ou indexação e classificação da informação, particularmente textual, de acordo com a demanda da tarefa (RUSSELL; NORVIG, 2016). O processo de RI consiste em identificar, no conjunto de dados (corpus) de um sistema, quais dados são de interesse ao usuário.

Os sistemas de RI precisam representar o conteúdo recuperado e apresenta-lo ao usuário de maneira que ele compreenda e satisfaça sua necessidade de informação. Para isso RI ordenam os documentos de acordo com seu grau de relevância, ou similaridade, baseado na chave de busca do usuário.

As técnicas de PLN e RI são aplicadas juntas para a resolução de diferentes

atividades, uma vez que PLN viabiliza soluções para alguns dos problemas de RI. A aplicação mais importante de PLN na RI está na interpretação do conteúdo de documentos. Assim, na maioria das vezes as técnicas de PLN são utilizadas na melhoria do desempenho de algumas tarefas da RI tradicional, como a indexação automática.

2.4.3 Latent Semantic Indexing

Indexação Latente Semântica, ou Latent Semantic Indexing em inglês (LSI), é uma das técnicas mais relevantes da área de Processamento de Linguagem Natural aplicadas na tarefa de representar o conteúdo de documentos de texto em um *corpus*.

A técnica é uma extensão do método de recuperação vetorial proposto por (SALTON; MICHAEL, 1983) que explora todas as inter-relações entre os termos e documentos através de seu posicionamento em um espaço vetorial. A técnica assume que há alguma estrutura *Semântica Latente* no padrão de uso das palavras entre documentos de um mesmo *corpus* que permite o uso de técnicas estatísticas (*Indexação*) para estimar essa estrutura latente. A representação dos termos de documentos pelo LSI permite a recuperação de informação sobre o conteúdo dos documentos. Uma vantagem da representação LSI é que uma consulta pode ser muito semelhante a um documento, mesmo quando eles não compartilham explicitamente palavras.

Para a criação do modelo de representação das inter-relações entre os termos e documentos em um espaço vetorial, LSI realiza uma decomposição de valor único (SVD). A decomposição é feita em uma matriz de frequência de termos de documentos, resultando na sua diminuição para uma estrutura que ao invés de representar documentos e consultas diretamente como palavras independentes, as representa como índices ou vetores de valores contínuos.

A recuperação de informação por meio de LSI é possível ao considerar a disposição geométrica dos índices no espaço vetorial. A disposição desses índices no espaço representa a sua similaridade entre si em relação à todos os documentos do *corpus*, logo ao calcular a distância geométrica entre os vetores, é calculado também a similaridade entre os respectivos documentos (MARCUS; MALETIC, 2003).

3 Trabalhos Correlatos

A partir do escopo deste trabalho de pesquisa foi realizado uma revisão da literatura onde uma série de trabalhos científicos foram estudados a fim de possibilitar o estabelecimento de uma base de conhecimento relevante. Essa base tem como finalidade contextualizar trabalhos desenvolvidos com temas das áreas de Mineração de Repositórios de Software e Interação Humano-Computador, fornecendo referências de como as atividades foram realizadas nas pesquisas, o que foi alcançado, além do que ainda pode ser aprimorado na área estudada.

Para a identificação dos trabalhos relacionados, foram utilizados os portais ScienceDirect [1], ACM Digital IEEE Xplore [2], Library [3], GoogleScholar[4], Springer[5]. A procura dos artigos foi realizada a partir de strings de busca relacionadas a termos como “Developer Modeling”, “Developer Profiling”, “Developer Recommendation” e "Source Code".

A versão final da string utilizada para selecionar trabalhos relacionados à tarefa de criação de perfis dinâmicos de desenvolvedores utilizando Modelagem de Usuários foi:

(“user modeling” OR “user profiling”) AND (“developer” OR "collaborator"OR "software developer") AND (“collaborative” OR “collaborative environment” OR “project”)

Para trabalhos relacionados à recomendação de desenvolvedores para a realização de tarefas, a string de busca utilizada foi:

("assigning"OR "recommending") AND ("fixer"OR "developer"OR "expert"OR "software developer"OR "maintainers") AND (“bugs” OR “fixes” OR “features” OR “changes” OR "issues"OR "tasks"OR "requests"OR "change requests") AND ("support"OR "maintenance") AND (“platform” OR “versioning” OR “project” OR “collaborative environment”)

Foram coletados artigos em inglês publicados depois de 2011 que possuem termos das strings de busca em seus conteúdos.

3.1 Modelagem de Desenvolvedores

O foco da tarefa de modelagem de desenvolvedores é a criação de perfis de desenvolvedores baseado em características históricas de seu trabalho.

Em (BEAL; BASSI; PARAISO, 2017), é apresentado um método de modelagem

[1] https://www.sciencedirect.com/
[2] https://ieeexplore.ieee.org/
[3] https://dl.acm.org/
[4] https://scholar.google.com.br/
[5] https://link.springer.com/

dinâmica de desenvolvedores capaz de gerar três possíveis perfis que caracterizam desenvolvedores em relação à qualidade do código produzido pelos mesmos. O trabalho realizou um experimento com estudantes matriculados em diferentes níveis de um curso de programação. No experimento, as métricas do código de cada desenvolvedor são adquiridas afim de compor o modelo de classificação, onde o rótulo de cada instância é referente ao nível de proficiência do desenvolvedor, obtida pelo seu progresso no curso de programação. A partir então de um classificador, é possível perfilar os desenvolvedores de acordo com a qualidade de seu produto.

Em (MEYER; ZIMMERMANN; FRITZ, 2017), é apresentado um modelo de desenvolvedores construído a partir da análise de características das percepções de produtividade de desenvolvedores. O trabalho realizou uma pesquisa com desenvolvedores profissionais na *Microsoft*, onde os mesmos deveriam responder um questionário em relação aos aspectos subjetivos que eles acreditam ter impacto, seja positivo ou negativo, em sua produtividade. Os participantes são agrupados com base na similaridade de suas respostas e os grupos são descritos conforme a frequência de termos de cada grupo, resultando na criação de seis possíveis perfis de desenvolvedores: sociáveis, solitários, focados, balanceados, líderes e orientado a objetivos.

O trabalho de (NAGUIB et al., 2013) trata de um método de recomendação de desenvolvedores para diferentes partes do processo de triagem de relatórios de *bugs*, baseado em Perfis de Atividade. O modelo de desenvolvedores que gera esses perfis é composto de dois conjuntos de dados. O primeiro conjunto de dados é referente ao tipo de atividade que os desenvolvedores podem exercer durante de triagem de *bugs*, sendo essas atividades: Atribuidor, Atribuído e Revisor. O segundo conjunto representa os tópicos abordados pelos relatórios de *bugs* em que o desenvolvedor participou de alguma forma. Os perfis de desenvolvedores gerados são sempre uma combinação dos dois conjuntos.

Outra abordagem para modelar desenvolvedores é apresentada em (CONSTANTINOU; KAPITSAKI, 2016), que consiste em gerar perfis de desenvolvedores a partir da sua perícia apontada pelo histórico de desenvolvimento em diferentes linguagens de programação. A partir de dados extraídos de *commits* em plataformas de versionamento, o trabalho quantifica a participação do desenvolvedor por projeto ao contabilizar o número de *commits* realizados pelo desenvolvedor, além do número de arquivos alterados e o número de linhas alteradas em relação a outros desenvolvedores do mesmo projeto. Permitindo a criação de perfis de desenvolvedores em relação à sua perícia em diferentes linguagens de programação.

3.2 Recomendação de Desenvolvedores

No escopo deste trabalho de pesquisa, o modelo de recomendação deve considerar os desenvolvedores como "itens"a serem recomendados à requisições de alterações de software. A literatura apresenta diversos trabalhos que propõem diferentes abordagens para a tarefa de selecionar, em meio a um conjunto de possíveis candidatos, o desenvolvedor mais adequado para a resolução de *bugs*, adição de novas funcionalidades e/ou outras requisições de alteração de software no geral.

Em (CAVALCANTI et al., 2016), é apresentado uma abordagem semi-automatizada para a atribuição de requisições de alterações de softwares para desenvolvedores, baseada na aplicação de um sistemas especialista combinado com técnicas de aprendizagem de máquina. A abordagem semi-automatizada proposta consiste de três etapas. Na primeira etapa, através de um conjunto de regras de recomendação estabelecidas por uma organização ou membros de uma equipe de desenvolvimento, o sistema especialista busca desenvolvedores da instituição para atribuir a requisições de alterações. Caso o sistema não encontre nenhum desenvolvedor apropriado, a segunda etapa é iniciada, onde um modelo de aprendizagem de máquina é aplicado e gera uma lista de desenvolvedores apropriados baseado em seus históricos de *commits*. O conjunto de desenvolvedores apropriados é usado como entrada ao sistema especialista. Caso o sistema falhe novamente em encontrar algum desenvolvedor apropriado, a fase final é iniciada, onde a atribuição é manual.

O trabalho de Zhang e colegas (ZHANG et al., 2016) propõe uma abordagem para realizar predição de severidade e recomendação de desenvolvedores para a manutenção de *bugs* de software. A proposta utiliza o classificador KNN e técnicas de recuperação de informação para buscar nos relatórios históricos de *bugs*, um conjunto de relatórios semanticamente similares à requisição de manutenção a ser recomendada. A lista de possíveis desenvolvedores é ordenada pelos autores que possuem maior número de manutenções realizadas mais similares à requisição de manutenção alvo. De mesma forma, a severidade do *bug* dessa requisição é dada pela severidade dos *bugs* de requisições similares.

Outra abordagem automatizada para a tarefa de recomendação de desenvolvedores é apresentada por (KAGDI et al., 2012), e consiste no uso de técnicas de recuperação de informação, para localizar unidades de código (classes, métodos ou arquivos) que implementam conceitos relativos à descrição textual de uma Change Request. São então analisadas as versões históricas das unidades de código, através da plataforma de versionamento, com o intuito de recomendar os desenvolvedores segundo o nível de experiência e contribuições às classes e métodos relevantes.

Em (SUN et al., 2017), é apresentado a abordagem "EDR_SI"para a tarefa de recomendação automatizada de desenvolvedores para requisições de alterações de software, baseada em hábitos profissionais e expertise de desenvolvedores. O EDR_SI busca similari-

dades textuais entre a descrição da Change Request desejada com as descrições dos *commits* históricos do mesmo repositório, afim de extrair desenvolvedores que possuem autoria dos *commits* com grau de similaridade acima de um limiar e também extrair os arquivos alterados para a resolução da alteração de cada *commit*. A partir do uso de CTM no conjunto de desenvolvedores e arquivos extraídos, é obtida um conjunto de desenvolvedores associados com listas personalizada de arquivos relevantes a cada desenvolvedor. A lista personalizada é criada a partir dos arquivos que o desenvolvedor alterou historicamente somada com arquivos que possuem tópicos similares aos alterados. O desenvolvedor recomendado é o que possui o conteúdo dos arquivos de sua lista personalizada mais textualmente similar à Change Request.

O trabalho (ZHANG; LEE, 2012) decide usar uma abordagem baseada em rede de relacionamento de desenvolvedores e avaliação de especialistas. O método apresentado consiste de uma sequência de etapas. A primeira etapa consiste na recuperação das requisições de manutenção de *bugs* mais similares à requisição alvo, através de técnicas de processamento de linguagem natural e clusterização. A partir dessas requisições similares, são extraídas características quantitativas como autoria e tempo de manutenção, que permitam a construção de uma rede de relacionamento dos desenvolvedores. O trabalho seleciona os desenvolvedores candidatos baseado no número de relações na rede e através da avaliação de especialistas. Os desenvolvedores candidatos são ordenados a partir de sua experiência e eficiência na tarefa de manutenção de *bugs*.

O trabalho (HOSSEN; KAGDI; POSHYVANYK, 2014) apresenta um método de recomendação de desenvolvedores chamado de *iMacPro*. A partir da descrição textual de uma Change Request recebida, o método localiza unidades de código relevantes (arquivos de código), de uma versão determinada do repositório. As unidades de código-fonte relevantes são ordenadas com base na propensão à mudança (*change proneness*, que é calculada com base no envolvimento em atividades de manutenção anteriores. O método considera os autores dessas unidades de código-fonte, e os mantenedores dessas unidades como desenvolvedores apropriados e os separa em uma lista. Os desenvolvedores nesta lista são ordenados e presume-se que são os candidatos mais adequados para resolver a Change Request de acordo com sua posição no conjunto.

3.3 Considerações Finais

Neste Capítulo foram apresentados trabalhos levantados na literatura que possuem algum tipo de relação com o método proposto por este trabalho de pesquisa. O método proposto busca possibilitar a recomendação de desenvolvedores para a realização de alterações em repositórios, sem comprometer a qualidade de código dos mesmos. Os trabalhos encontrados na literatura, não pautam seus modelos e métodos de acordo com a

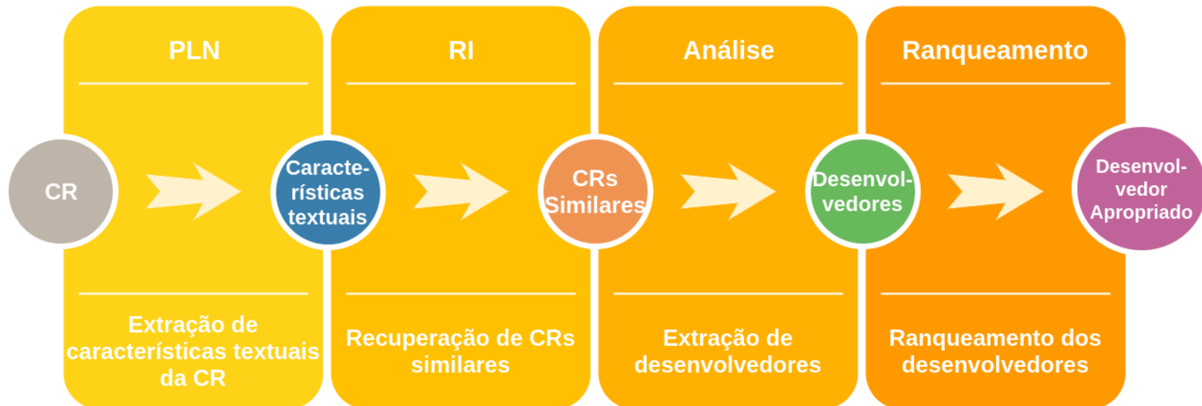
manutenção da saúde de repositórios. A Revisão da Literatura mostrou que o tema tratado por este trabalho ainda é um nicho pequeno com um baixo número de artigos científicos que aplicam o conceito de Modelagem de Usuários ou Sistemas de Recomendação na tarefa de resolução de *Change Requests*. Os trabalhos científicos relevantes foram separados e tratados em duas seções:

Na seção 3.1 foram apresentados trabalhos que aplicam o conceito de modelagem de usuário no âmbito do desenvolvimento de software afim de tentar caracterizar o perfil de desenvolvedores. Os modelos que mais se aproximam na questão de modelagem de desenvolvedores proposta por este trabalho de pesquisa, são os métodos apresentados por (BEAL; BASSI; PARAISO, 2017), (CONSTANTINOU; KAPITSAKI, 2016) e (NAGUIB et al., 2013), que escolhem por caracterizar desenvolvedores por meio da análise quantitativa de seu trabalho, ao contrário de (MEYER; ZIMMERMANN; FRITZ, 2017) que perfila desenvolvedores com base em percepções subjetivas de produtividade. Apesar de buscarem características objetivas da atividade de desenvolvimento, os métodos falham por avaliar somente um aspecto da atividade, a qualidade do código do desenvolvedor em (BEAL; BASSI; PARAISO, 2017) e a perícia do desenvolvedor em (CONSTANTINOU; KAPITSAKI, 2016) e de certa forma em (NAGUIB et al., 2013), fazendo com que o primeiro modelo assuma que a qualidade do código do desenvolvedor independe do tipo de atividade que o mesmo exerce, e com que os outros dois assumam que a perícia de um desenvolvedor é livre da qualidade de seu código. Logo, esses modelos não são indicados para a construção de um método de recomendação automatizado, uma vez que os perfis de desenvolvedores que eles produzem não representa aspectos diversos o suficiente do trabalho de desenvolvimento, podendo gerar um impacto negativo na qualidade do código de um repositório.

A seção 3.2 é composta de trabalhos que apresentam abordagens para a tarefa de atribuir uma Change Request ao desenvolvedor mais adequado, no contexto de ambiente de desenvolvimento colaborativo. No geral, as abordagens encontradas na literatura possuem uma estrutura semelhante à apresentada na Figura 4, em que, a intenção é basicamente caracterizar a alteração por meio de diferentes técnicas de PLN, afim de possibilitar a seleção de requisições de alteração similares baseada na análise histórica do repositório, e assim, recomendar desenvolvedores ativos na realização de requisições passadas, variando também os requisitos de recomendação dos desenvolvedores mais apropriados entre os trabalhos correlatos. O trabalho que apresenta um método de recomendação mais aproximo ao proposto por este trabalho é (HOSSEN; KAGDI; POSHYVANYK, 2014), que utiliza técnicas parecidas para a recuperação de requisições. Entretanto, o método usa a propensão de mudança (*Change Proneness*) como característica principal de recomendação. Além dessa distinção, ele se limita a análise pontual de uma versão do código de um repositório, sem levar em consideração as possíveis variações no trabalho de desenvolvedores com o passar do tempo.

Por fim, nenhum trabalho apresentou um método de recomendação de perfis de desenvolvedores para a resolução de requisições de alteração, baseado em qualidade de código, perícia e contexto.

Figura 4 – Método geral de recomendação de desenvolvedores



Fonte: o autor 2020

4 Procedimentos Metodológicos

Este capítulo apresenta os procedimentos metodológicos utilizados durante o desenvolvimento desta pesquisa. A pesquisa foi feita a partir da adaptação da metodologia *Design Science Research*, que foca no desenvolvimento e avaliação de artefatos. O desenvolvimento da pesquisa foi estruturado basicamente em quatro fases: Planejamento inicial, Fase exploratória, Desenvolvimento e Avaliação. A Figura 5 apresenta uma visão geral dessas fases e suas tarefas.

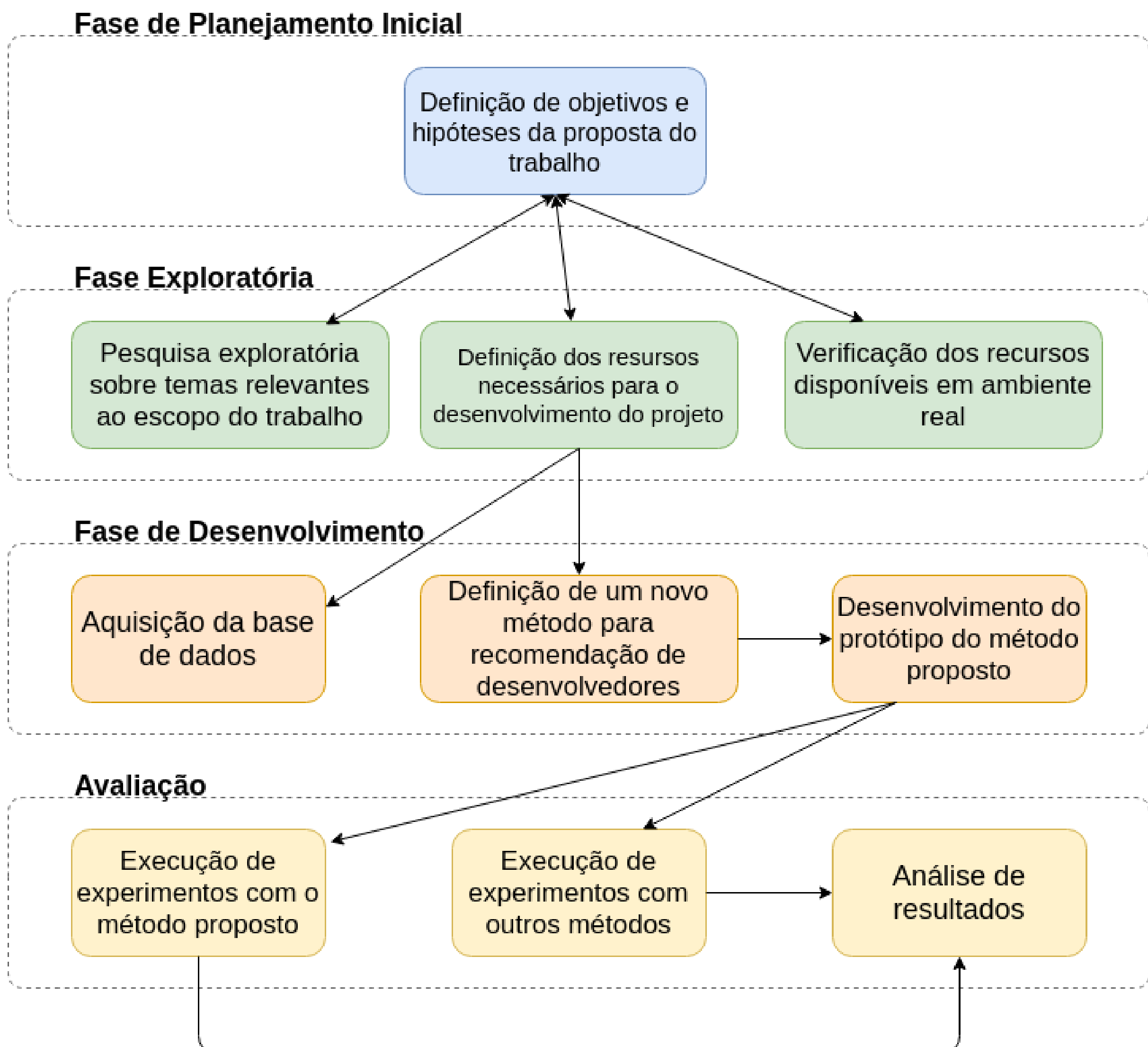


Figura 5 – Estrutura da pesquisa

4.1 Fase de Planejamento Inicial

A realização de uma revisão na literatura, com trabalhos relacionados à aplicação de técnicas das linhas de pesquisa relevantes para a tarefa de recomendação de desenvolvedores, permitiu constatar que os temas são geralmente abordados separadamente, com trabalhos se dedicando à tarefa de modelagem de desenvolvedores (BEAL; BASSI; PARAISO, 2017)(MEYER; ZIMMERMANN; FRITZ, 2017) e outros à recomendação de desenvolvedores (KAGDI et al., 2012)(CAVALCANTI et al., 2016), sem ativamente modelar os desenvolvedores. Notou-se na literatura que apesar do conhecimento dos benefícios, ainda existem muitos fatores que impedem a promoção de colaboração eficaz de equipes de desenvolvimento de software (MISTRÍK et al., 2010). Fatores como a alta velocidade com que os requisitos de um software evoluem (ROBINSON; SHARP, 2010) e a diversidade de contribuidores durante o processo de desenvolvimento atualmente (RICHARDSON et al., 2010). Um dos artefatos mais importantes do desenvolvimento colaborativo é a Change Request e o seu gerenciamento é uma das atividades que mais exigem coordenação de participantes, tendo em vista que mudanças de requisitos de um projeto podem ter impactos negativos.

Nesse contexto, definiu-se como objetivo geral deste trabalho desenvolver um método capaz de atribuir automaticamente Change Requests de código para desenvolvedores, a partir de seu perfil histórico de desenvolvimento. O primeiro desafio encontrado durante a fase exploratória foi a inexistência de uma base de dados pública composta de características de diferentes aspectos da tarefa de desenvolvimento, extraídas de repositório de códigos. Por conta disso, o primeiro objetivo específico foi definido pela aplicação de modelagem de usuários para a tarefa de representar estruturadamente o trabalho de desenvolvedores na realização de alterações de código. Isso também implica na definição do escopo de quais aspectos do trabalho seriam considerados para a modelagem de desenvolvedores. Isso a partir da execução da etapa de exploração em paralelo, e quais dados estão disponíveis em ferramentas de desenvolvimento colaborativo.

O segundo objetivo definido foi a criação de uma base de dados referentes à *commits*, qualidade de código e Change Requests.

O último objetivo é referente à construção do método de recomendação de Change Requests e de um protótipo computacional para a realização da avaliação do método proposto. Com o complemento dos primeiros objetivos específicos, o desenvolvimento de um protótipo é possível de ser realizado e avaliado em um ambiente real.

4.2 Fase Exploratória

A primeira tarefa desta fase é a realização de uma pesquisa exploratória a partir do escopo deste trabalho. Foram pesquisados diversos trabalhos científicos nas áreas de Mineração de Repositórios de Software e Interação Humano-Computador. Essa etapa é essencial na concepção dos objetivos de pesquisa e a criação do método, pois possibilita a compreensão do campo de estudo ao dar o contexto de abordagens relacionadas.

A pesquisa exploratória foi feita na forma de revisão da literatura. Nesta revisão foram explorados os trabalhos relacionados ao uso de modelagem de usuários para caracterizar desenvolvedores, e modelos de recomendação para a tarefa de atribuição de desenvolvedores e Change Request.

Após a coleta e leitura dos artigos foi possível realizar as duas outras tarefas da fase exploratória. Para o estabelecimento da dimensão dos recursos disponíveis em ambientes reais de desenvolvimento, foi realizada uma série de visitas à empresa parceira: Siemens Ltda, sede Curitiba/PR. Essas visitas também tiveram o intuito de conhecer quais ferramentas são utilizadas em ambiente corporativo, tais como sistemas de controle de versionamento por exemplo.

Com base nos recursos disponíveis na literatura, combinados com observações realizadas nas visitas, foram selecionados as técnicas e ferramentas que compõem o método proposto por este trabalho. Essa fase estabelece o escopo dos recursos que estarão presentes para a tarefa de Aquisição da base de Dados na fase de Desenvolvimento apresentada na próxima seção.

4.3 Fase de Desenvolvimento

A fase de desenvolvimento consiste de três tarefas: coletar dados para a formação de uma base com dados de *commits*, qualidade de código e Change Requests resolvidas; definir o método para a recomendação de desenvolvedores e o desenvolvimento de seu protótipo.

A tarefa de coleta de dados é realizada em duas etapas. Na primeira etapa, são extraídas características históricas de código, de *commits* e de Change Requests em repositórios de código livre selecionados por critérios como: número de *commits*, número de desenvolvedores, número de Change Request, idade e abrangência do projeto, por exemplo. Também foi usado como critério de seleção os projetos que já foram utilizados na literatura em outras abordagens de recomendação de desenvolvedores, afim de facilitar a avaliação.

Na etapa de desenvolvimento do método proposto, foram estudados os modelos de recomendação de trabalhos correlatos da literatura e suas características. Após esse estudo foram definidas as etapas do método, descrito mais profundamente no Capítulo 5.

Apoś a definição do método, foi possível iniciar a montagem de um protótipo com a utilização de ferramentas presentes na literatura, levando em consideração a base de dados montada na primeira tarefa. Após a construção do método, foram realizados testes empíricos para definir as melhores configurações do método utilizado, além de avaliar a eficácia do método em si.

4.4 Fase de Avaliação

Após o término da fase de desenvolvimento, é preciso realizar testes a fim de estipular a melhor configuração do método proposto e da base de dados adquirida em fases anteriores. No geral, dois grandes aspectos do método proposto devem ser avaliados no intuito de determinar a sua efetividade. O primeiro aspecto a ser avaliado é a capacidade do método de modelar desenvolvedores efetivamente e o segundo aspecto a ser avaliado é a capacidade do método de atribuir Change Request à desenvolvedores apropriados.

Levando em conta a diversidade dos dados passíveis de serem extraídos por meio das ferramentas de desenvolvimento colaborativo durante a fase de exploração, é possível inferir diferentes modelos de desenvolvedores através de aspectos variados do processo de resolução de Change Request. Existem na literatura abordagens para a modelagem de desenvolvedores com subconjuntos das características propostas para o modelo de desenvolvedor deste trabalho, abordagens que levam em conta somente a qualidade do código (BEAL, 2014) como aspecto modelador, ou somente as características do *commit* (CONSTANTINOU; KAPITSAKI, 2016). Entretanto, não existe, até onde se sabe, nenhum trabalho que proponha o mesmo conjunto, o que torna difícil a comparação de modelos.

Para a verificação da eficácia dos perfis e a configuração ideal do modelo de desenvolvedor, serão aplicados testes como o cálculo da inércia e o "método do cotovelo", com o intuito de analisar a distribuição das instâncias em grupos, a fim de se obter o número ideal de *clusters*. Uma vez estabelecida a configuração do modelo de desenvolvedor, é possível a realização de outros testes empíricos com foco em avaliar a eficácia e auxiliar a configuração das etapas do método de recomendação proposto. Será aplicado o método correlato **iMacPro** na base de dados adquirida na fase de desenvolvimento. Para avaliar a precisão de atribuição de Change Requests será utilizado a métrica recall@k (Equação 6.2) para 1, 3 e 5 desenvolvedores apropriados conforme (SUN et al., 2017), e para avaliar o impacto na qualidade do código do repositório será utilizado a métrica de contribuição (Equação 6.1) proposta por (BASSI et al., 2018). A seção de Resultados deste trabalho apresentará com mais detalhes ambas as métricas.

5 Método Proposto

Neste capítulo é apresentado o método proposto neste trabalho. O método é capaz de atribuir automaticamente Change Requests em ambientes de desenvolvimento colaborativo para contribuidores a partir de seu perfil dinâmico. Esse perfil é construído a partir da extração de características objetivas e históricas de plataformas de versionamento.

Neste capítulo também são apresentados os pressupostos do método e descritas as etapas e ferramentas que o compõem.

5.1 Pressupostos do Método

Como a tarefa principal do método é a atribuição de Change Requests para desenvolvedores em um ambiente colaborativo, alguns pressupostos foram definidos para o projeto do método, tais como:

1) A ferramenta implementada a partir do método deve possuir acesso a uma plataforma de versionamento com repositório de códigos, gerenciamento de Change Request e dados de *commits*;

2) Os projetos analisados devem seguir estritamente o paradigma de orientação a objetos;

3) O método deve coletar os dados de desenvolvedores de forma não-invasiva;

4) O método considerará como desenvolvedor potencial para a realização de uma alteração em um projeto, aquele que já tenha resolvido alguma requisição de alteração anteriormente.

5.2 Visão Geral

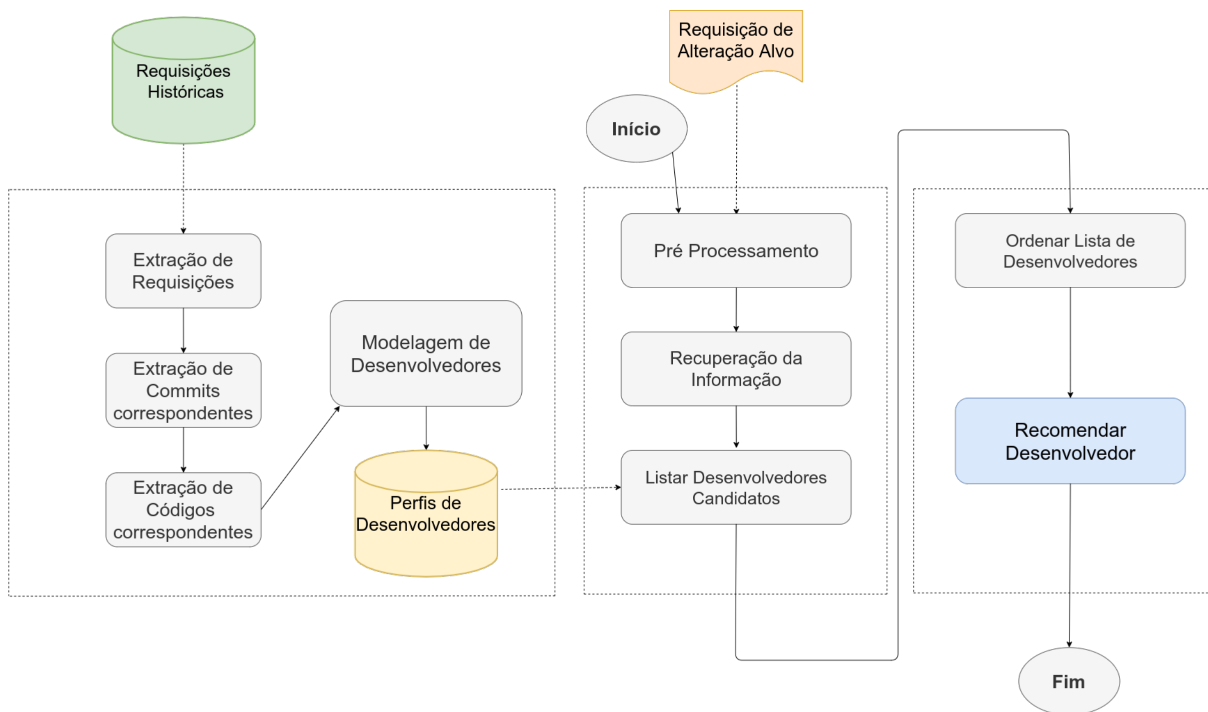
O método proposto não prevê interferência humana para a realização da atribuição de requisições de alteração a desenvolvedores. O método pode ser dividido basicamente em três etapas.

1. A partir de técnicas de aprendizagem de máquina e modelagem do usuário, são criadas representações das características de desenvolvedores na resolução de requisições de alteração. O resultado dessa etapa é a base de perfis de desenvolvedores, onde indivíduos de um grupo possuem características semelhantes.

2. As características textuais de uma requisição alvo são pré-processadas e usadas para encontrar, através do uso de técnicas de Recuperação de Informação, a requisição resolvida mais similar a esta. O método busca na base de perfis, os autores das requisições que estão agrupadas com a requisição mais similar e os lista como desenvolvedores candidatos.

3. Os desenvolvedores candidatos são ordenados em relação ao seu impacto na qualidade de código e o desenvolvedor com o menor impacto é recomendado para a resolução da requisição alvo.

Figura 6 – Visão geral do método proposto



Fonte: Autoria Própria

5.2.1 Modelagem de Desenvolvedores

O modelo de desenvolvedor proposto por este trabalho é um modelo dinâmico, uma vez que as características do desenvolvedor se modificam ao longo do tempo, conforme ele participa do desenvolvimento de diferentes componentes de software. Segundo (BARTH, 2010) a tarefa de construção de modelos dinâmicos é organizada nas seguintes fases: Composição do modelo, Representação, Aquisição, Aprendizado e Manutenção do Modelo. Cada fase por sua vez é caracterizada pela aplicação de técnicas e métodos conforme descrito a seguir. O objetivo dessa etapa do método é a criação de uma base de perfis de desenvolvedores.

5.2.1.1 Composição do Modelo

Para a composição do modelo dinâmico de desenvolvedor proposto, é preciso identificar quais dados são relevantes para a representação de aspectos comportamentais de desenvolvedores em ambiente de desenvolvimento colaborativo. Durante a fase exploratória descrita no Cap.4, foi estabelecido a necessidade do uso de três tipos diferentes de dados históricos para a composição de um modelo que auxiliasse na atribuição de desenvolvedores para requisições de alteração: dados sobre a qualidade do código de um desenvolvedor, afim de inferir sobre o quão bem ele resolve uma requisição; dados sobre *commits*, que revelam informações sobre o tamanho de alterações realizadas por esse desenvolvedor; dados sobre a natureza das requisições resolvidas por ele(a), afim de se estabelecer quais os tipos de alteração o desenvolvedor tem experiência em realizar.

5.2.1.2 Representação

Como o modelo dinâmico possui uma fase de aprendizado de máquina, foi estabelecido o uso do formato *arff* para a representação do modelo de desenvolvedor proposto por este trabalho. Este formato de arquivo descreve uma lista de instâncias que compartilham um conjunto de características, sendo favorável para o uso em modelos dinâmicos por possuir, como o XML, uma estruturação flexível com capacidade de adição ou remoção de elementos com facilidade(ROBU; STOICU-TIVADAR, 2010).

As características que refletem a qualidade de código são descritas através de diferentes métricas de código. A seleção de métricas de código para compor o conjunto foi realizada, no geral, de acordo com a influência da métrica na avaliação objetiva da qualidade de software e os recursos disponíveis pela ferramenta de extração CK(ANICHE, 2015). As métricas **WMC**, **DIT** e **NOC** fazem parte do conjunto de métricas de Chimdaber e Kemerer(CHIDAMBER; KEMERER, 1991), que é um dos conjuntos mais importantes de métricas orientadas a objetos (SOLIMAN; EL-SWESY; AHMED, 2010), e mais amplamente referenciado. As métricas de Chimdaber e Kemerer, são utilizadas em muitos trabalhos para monitorar a qualidade do software em desenvolvimento (PLOSCH et al., 2010), estabelecendo intervalos de valores para medidas de interesses. As métricas relevantes ao modelo do desenvolvedor proposto são:

- **CBO**: *Coupling between objects*
- **DIT**: *Depth Inheritance Tree*
- **NOSI**: *Number of static invocations*
- **RFC**: *Response for a Class*
- **WMC**: *Weight Method Class*

- **LCOM**: *Lack of Cohesion of Methods*
- **NBD**: *Max nested blocks*
- **LOC**: *Lines of code*

Os dados presentes em *commits* de repositórios em sistemas de controle de versão, têm como intuito a caracterização da atividade de um desenvolvedor no repositório. Esses dados fornecem informação sobre o comportamento do desenvolvedor com relação à fatores como frequência e tamanho da alteração de código. Os dados presentes em *commits* relevantes ao modelo do desenvolvedor proposto são:

- Id repo (identificação do repositório)
- Id author (identificação do autor)
- Id *commit* (identificação do *commit*)
- Creation date (data de criação do *commit*)
- *Commit* additions (Total de adições realizadas no *commit*)
- *Commit* deletions (Total de remoções realizadas no *commit*)
- *Commit* total changes (Total de alterações realizadas no *commit*)

As características que identificam a perícia do desenvolvedor em relação às áreas de conhecimento necessárias para a resolução de requisições de alterações, podem ser encontradas nas próprias requisições. Cada requisição possui elementos textuais que são criadas pelo autor da requisição com o objetivo de descrever a natureza da alteração requirida Figura 7, como: título, descrição e comentários. Um *corpus* é criado a partir desses elementos, esse corpus é indexado e o índice de cada documento representa o texto não-estruturado de uma requisição resolvida.

Figura 7 – Exemplo Típico de Requisição de Alteração

Fonte: (GITHUBINC, 2008)

As características dos conjuntos que representam o comportamento relevante do desenvolvedor para este trabalho, são agrupadas em um único conjunto para compor o modelo dinâmico proposto. A representação do modelo proposto em *arff* é apresentado em seguida.

```
@RELATION ModeloDesenvolvedor
% Atributo relacionado a conteudo conceitual da requisicao de alteracao
@ATTRIBUTE Indice string
% Atributos relacionados ao commit do codigo de uma alteracao
@ATTRIBUTE Id repositório numeric
@ATTRIBUTE Id desenvolvedor numeric
@ATTRIBUTE Id commit numeric
@ATTRIBUTE Adições no commit numeric
@ATTRIBUTE Remoções no commit numeric
@ATTRIBUTE Alterações no commit numeric
@ATTRIBUTE Data do commit numeric
% Atributos relacionados a qualidade de software
@ATTRIBUTE CBO numeric
@ATTRIBUTE WMC numeric
@ATTRIBUTE DIT numeric
```

```
@ATTRIBUTE RFC numeric
@ATTRIBUTE LCOM numeric
@ATTRIBUTE NOSI numeric
@ATTRIBUTE LOC numeric
@ATTRIBUTE NBD numeric
```

5.2.1.3 Aquisição

O método proposto realiza o processo de aquisição de dados a partir da extração de características de requisições históricas resolvidas em sistemas de controle de versionamento. O termo "resolvida"é empregado à requisições que possuem elementos como: a descrição da alteração; Possivelmente uma sequência de comentários de diferentes contribuidores sobre a alteração, conforme mostrado em Figura 7; Desenvolvedor atribuído para a resolução da requisição; E a identificação do *commit* com o código referente às alterações da requisição.

A Figura 8 apresenta uma requisição resolvida presente na ferramenta de gerenciamento de versão do repositório Elastic Search[1] e seus elementos:

1 Status da requisição

2 Autor da requisição

3 Descrição

4 Título e id da requisição

5 Desenvolvedor a quem a requisição foi atribuída

A Figura 9 é referente à continuação da requisição exemplo, e possui:

1 Desenvolvedor a quem a requisição foi atribuída

2 Descrição do *commit*

3 id do *commit*

Durante o ciclo de vida de uma requisição, é possível que a mesma não seja aceita como válida ou que ela seja resolvida sem alteração no código. Essas requisições não são relevantes para o escopo deste trabalho por serem resolvidas sem a atribuição à um desenvolvedor.

A aquisição do conjunto de características textuais é realizada através do agrupamento de todos os elementos textuais de uma requisição resolvida em documentos

[1] <https://github.com/elastic/elasticsearch/issues/40250>

Figura 8 – Exemplo de Requisição Resolvida

Add Bulk Delete API to Blob Store Interface #40250

New issue

Closed original-brownbear opened this issue on Mar 20 · 1 comment

1 2 3 4 5

original-brownb... commented on Mar 20 Member

Continuing from #39656 here:

Currently deleting snapshots can be a very slow process if the delete entails removing a large number of blobs from the repository (i.e. when one or more indices that consistent of many files becomes unreferenced and has to be deleted).
The current implementation of the blob store only offers synchronous and sequential deletes of these blobs, which means that one needs at least as many network calls as the number of to be deleted blobs.
In addition to being potentially very slow, this also might incur higher than necessary cost for blob stores cloud solutions that bill for each API request.

There are at least two possible improvements to be made to performance:

1. Make the delete API non-blocking so blobs can be removed in parallel where possible. (Suggested implementation in #40144)
2. Use bulk deletion APIs where possible (e.g. S3's bulk delete)

I think we should do the latter here and adjust `org.elasticsearch.common.blobstore.BlobContainer#deleteBlob` to accept a list of blobs to delete and leverage bulk delete APIs in implementations where possible.

Assignees
original-brownbear

Labels
:Distributed/Snapshot/Restore
>enhancement

Projects
None yet

Milestone
No milestone

Notifications
Subscribe
You're not receiving notifications from this thread.

Fonte: Elastic Search

Figura 9 – Exemplo de Atribuição de Requisição

original-brownbear self-assigned this on Mar 21 1

original-brownbear added a commit to original-brownbear/elasticsearch that referenced this issue on Mar 21

3 ed24c02

```
Add Bulk Delete Api to BlobStore ...

* Adds Bulk delete API to blob container
* Implement bulk delete API for S3
* Adjust S3Fixture to accept both path styles for bulk deletes since the S3
SDK uses both during our ITs
* Closes elastic#40250
```

2

Fonte: Elastic Search

pertencentes a um *corpus*. Graças a criação do *corpus* é possível a representação do conteúdo textual de cada documento e a recuperação de informações contidas nos mesmos por meio de técnicas de Processamento de Linguagem Natural e Recuperação de Informação, conforme será mostrado na próxima subseção.

Por meio da extração de uma requisição resolvida e atribuída, é iniciada a extração

do *commit* relacionado à sua resolução. Os dados sobre o *commit* e o seu repositório, são obtidos por meio da API da ferramenta de controle de versionamento.

A partir da extração do conjunto anterior, são obtidos os arquivos do repositório na versão do *commit*. Os arquivos são extraídos e através do uso da ferramenta *CK*, é calculado os valores das métricas de qualidade somente dos arquivos alterados pelo desenvolvedor no *commit* relacionado à requisição de alteração. Os dados das métricas de qualidade de um *commit* são comparados com os de seu antecessor, possibilitando medir precisamente o impacto na qualidade de código que um desenvolvedor teve com as suas alterações.

5.2.1.4 Aprendizado

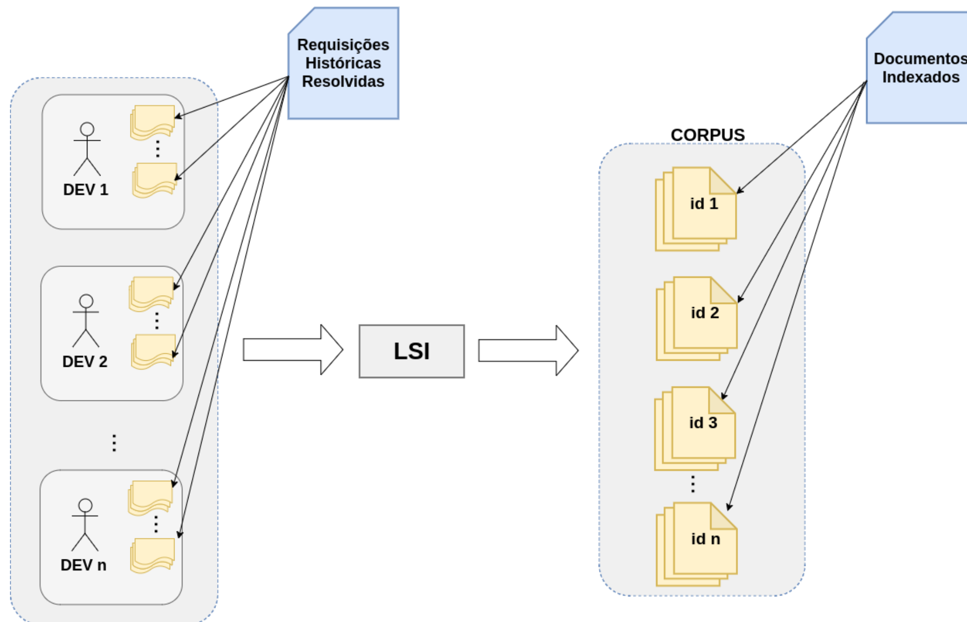
A fase de Aprendizado é responsável pelo pré-processamento e análise dos dados, além da implementação dos métodos de aprendizagem de máquina. Os conjuntos de dados de *commits* e métricas de qualidade de código não precisam de nenhum pré-processamento para serem usados no Modelo de Desenvolvedor proposto por este trabalho, uma vez que representam seus respectivos aspectos do trabalho de desenvolvimento de uma maneira quantitativa. Entretanto, o conjunto de dados relativo ao texto das Requisições de Alterações precisa ser quantificado. Logo, a primeira tarefa da etapa de Aprendizado é indexar os documentos do *corpus* criado na subseção de Aquisição.

O processo de indexação é realizado através da aplicação de uma técnica de PLN chamada de *Latent Semantic Indexing* (LSI). Essa técnica cria uma assinatura única (índice) para cada documento (DEERWESTER et al., 1990). O resultado desta técnica é a representação dos textos das Requisições de Alteração por um vetor de valores relativo aos termos mais importantes de sua respectiva Requisição em relação ao *corpus* inteiro. O *corpus* é então efetivamente representado por uma matriz de índices de seus documentos, o que permite o uso de métricas de similaridade para relacionar índices e consequentemente relacionar requisições de alteração. O número ideal de tópicos latentes foi obtido através da análise do tamanho textual médio dos textos de cada repositório e da especificidade dos tópicos gerados.

Os vetores de índices são então agrupados através do algoritmo *k-means*, a fim de que cada grupo de Requisições com as mesmas características textuais represente um tipo de alteração que deve ser realizada, conforme indicado em Figura 10. Os grupos de Requisições são enumerados e o número do grupo de cada requisição de alteração é extraído para formar o modelo conforme representado na sub-sessão de Representação, junto com o conjunto de dados de *commit* e métricas de qualidade de código.

Uma vez que os três conjuntos de dados que compõem a base do modelo do desenvolvedor tenham sido adquiridos e processados, é possível a realização da tarefa da criação de perfis de desenvolvedores. Essa tarefa é responsável pela criação de diferentes perfis baseados na análise das instâncias do modelo dinâmico do desenvolvedor. Essa

Figura 10 – Indexação de Requisições



Fonte: o autor 2020

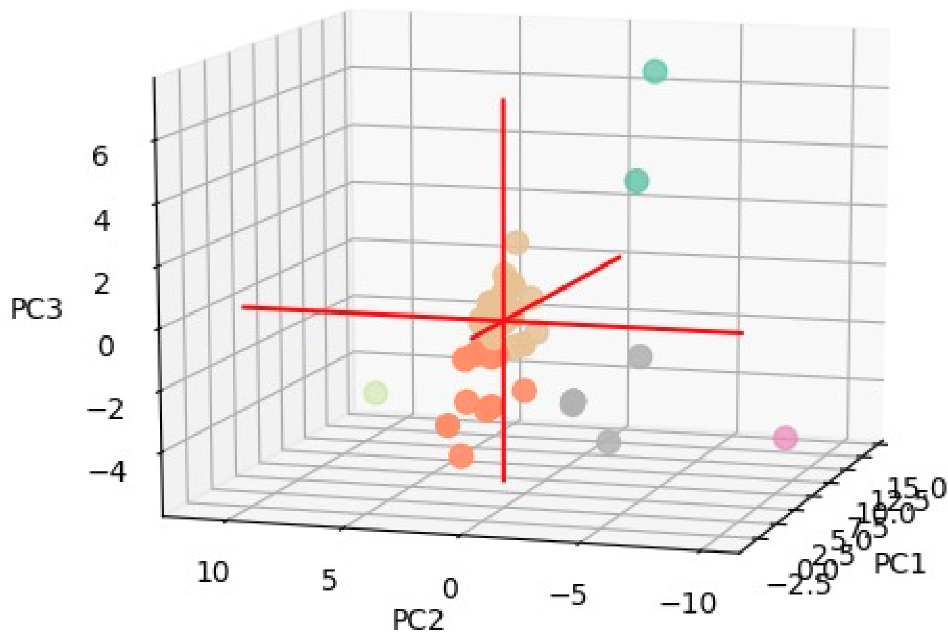
criação pode ser realizada através de técnicas de aprendizado supervisionado ou não (BARTH, 2010).

Contudo, para o escopo deste trabalho, não é possível gerar relações de uma forma objetiva e automática sobre qualquer desenvolvedor em qualquer projeto baseado somente em características de repositório, por exemplo só de analisar as métricas de um *commit* não é possível inferir sobre o nível de experiência do desenvolvedor que o criou. Como não foi encontrado na literatura bases rotuladas com os conjuntos de dados de interesse deste trabalho, e nem é possível a realização de uma rotulação própria, foi escolhida a criação de perfis de desenvolvedores por meio de técnicas de aprendizagem não-supervisionada.

Para escolher a técnica de aprendizagem não-supervisionada mais adequada, foi necessária a realização de uma análise dos dados coletados na subseção anterior. Cada instância da base adquirida possui 12 atributos. Apesar de a base não possuir uma dimensionalidade particularmente alta, foi utilizado a técnica de Análise de Componentes Principais ou *Principal Component Analysis* (PCA) em inglês, nos dados normalizados com o intuito de facilitar a exploração visual de sua distribuição. O uso do PCA para 3 Componentes Principais permitiu a representação dos dados conforme apresentado em Figura 11.

O gráfico Figura 12 exemplifica melhor a relação entre o grau de variância dos atributos para a composição de cada Componente Principal plotado em Figura 11, onde os atributos contribuem com o grau de variância conforme o seu valor e posição no eixo

Figura 11 – Representação Tridimensional dos Dados no Repositório *Guava*



Fonte: o autor 2020

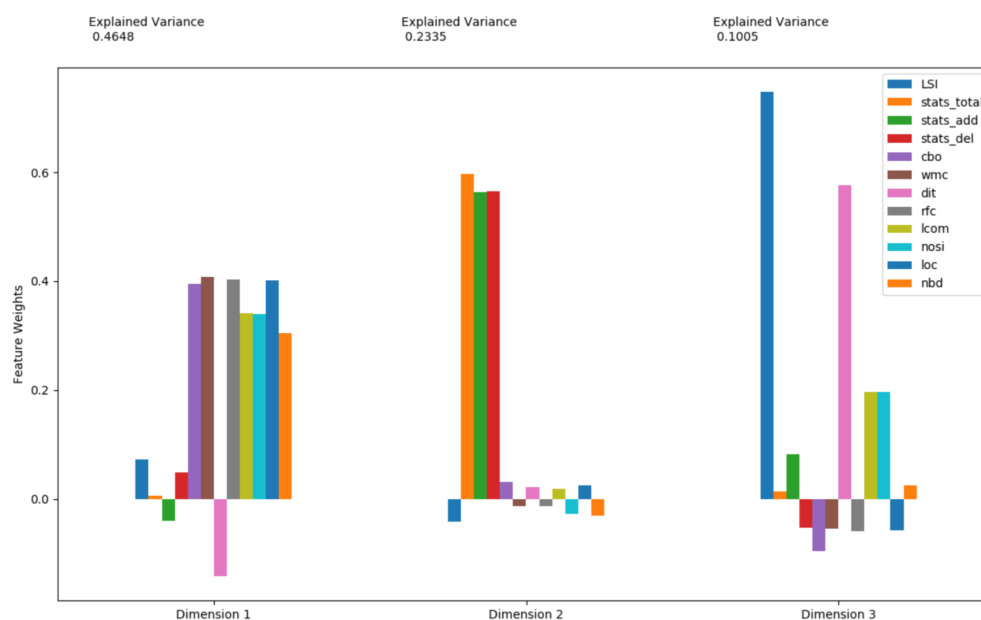
horizontal.

Também foi realizado uma análise do grau de variância de cada Componente Principal entre 1 e 12, a fim de analisar o quanto de informação é perdida com cada diminuição de dimensionalidade dos dados adquiridos. Essa análise foi feita para averiguar a possibilidade de se utilizar menos atributos ao descobrir redundâncias, como dois atributos que representam os dados da mesma forma, logo não teria perda de informação com a exclusão de um ou outro. O gráfico apresentado em Figura 13 indica que a maior parte da informação que representa os dados está contida em 7 atributos (cerca de 96%).

A conclusão da fase de análise foi de que a a distribuição normal(gaussiana) do dados e o baixo número de *outliers* faz com que a aplicação do algoritmo *k-means* seja apropriado para a geração dos perfis de desenvolvedores, além de ser um algoritmo simples, rápido e robusto. E de que a perda de informação com a possível diminuição da dimensionalidade dos dados pelo uso de pca não compensa para a base adquirida, uma vez que a mesma só possui 12 atributos conforme indicado em Figura 13 e a remoção dos atributos com pouca variância não resultaria em ganhos significativos de custo computacional. Deve-se considerar também que a adição de novos dados durante a manutenção do método pode aumentar a relevância de alguns atributos.

O *k-means* é capaz de dividir a população da base do modelo em grupos de instâncias com relação à similaridade de seus atributos. Assim, os grupos são compostos por dados

Figura 12 – Gráfico de *Explained Variance para PCA* no Repositório *Guava*



Fonte: o autor 2020

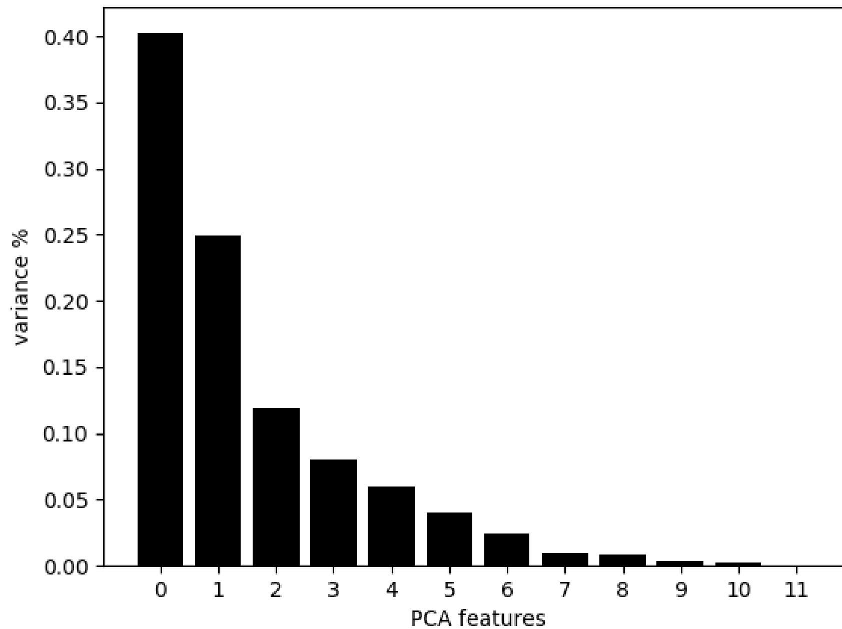
de requisições que foram resolvidas de forma similar, tanto em relação a qualidade da alteração, ao tamanho da alteração quanto ao tipo de alteração. O número ideal de grupos para cada repositório foi determinado usando o chamado "método do cotovelo", que consiste em executar o *k-means* para um intervalo de valores e calcular a soma das distâncias ao quadrado de cada ponto até o centro atribuído (inércia) para cada execução. Quando plotado, o gráfico da inércia assemelha-se a um braço curvado, onde o "cotovelo"indica o k ótimo Figura 14. O número ideal de grupos encontrado foi o de oito grupos, esse valor foi obtido através da média do k ótimo executado 10 vezes para cada repositório.

Após a realização de todo o pré-processamento e aplicação das técnicas de agrupamento, o resultado da etapa de Aprendizado é a geração da base de Perfis de Desenvolvedores, que será útil nas próximas fases do Método.

5.2.1.5 Manutenção do Modelo

A tarefa de desenvolvimento de software é complexa e dinâmica, o que implica que o modelo proposto por este trabalho deve seguir a abordagem dinâmica. O modelo permite a adição de novos dados, afim de acompanhar o progresso do trabalho de desenvolvedores ao longo do tempo. Existem duas maneiras para a realização da manutenção de um modelo de desenvolvedores: o modelo é atualizado através da alteração explicita do perfil por parte dos usuários; ou o modelo é atualizado de maneira implícita, onde os dados são obtidos de

Figura 13 – Grau de Variância dos Dados no Repositório *Guava*



Fonte: o autor 2020

maneira contínua e não-invasiva (BARTH, 2010). A manutenção do modelo proposto por este trabalho é realizada de uma maneira implícita periodicamente com o surgimento de novas requisições resolvidas pela atribuição do próprio método.

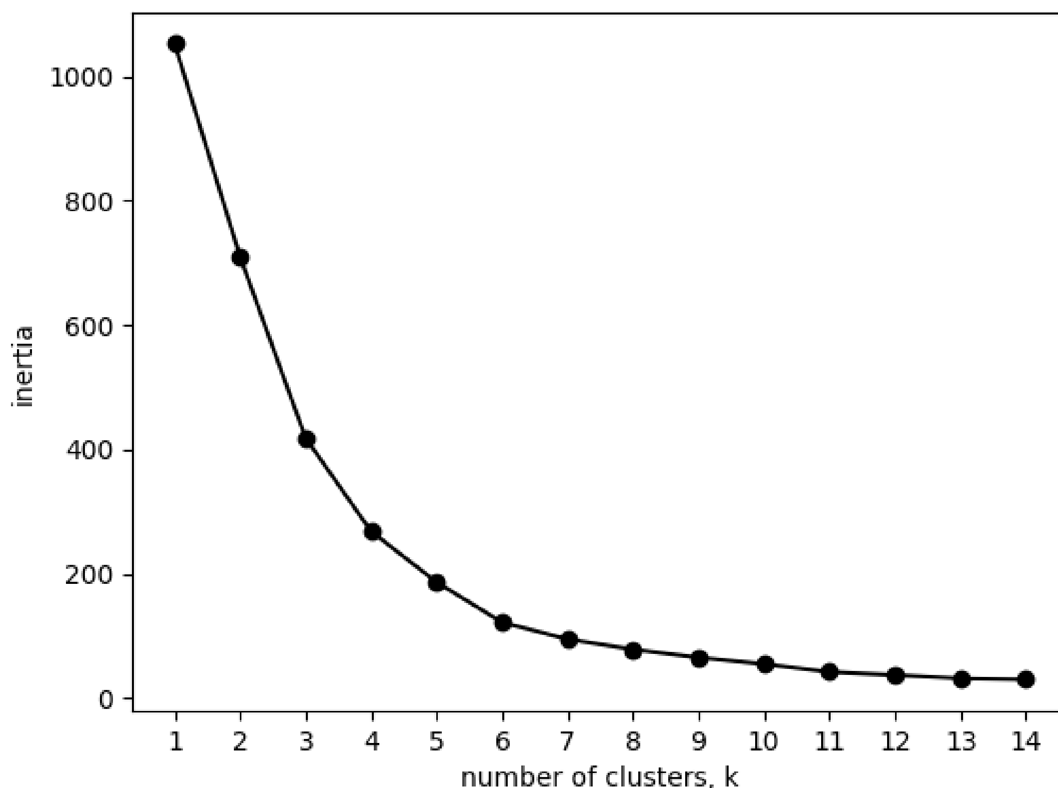
5.2.2 Atribuição de Change Request

A etapa de Atribuição de Requisição é viabilizada pela fase de Aprendizado na etapa de Modelagem de Desenvolvedores. A base de Perfis de Desenvolvedores é essencial para o método de recomendação de desenvolvedores proposto por este trabalho, uma vez que fornece o desenvolvedores que serão associados a uma requisições alvo (requisições em aberto, ou que não possuem um desenvolvedor).

A primeira tarefa da etapa de atribuição do método é o pré-processamento da requisição alvo. Essa tarefa consiste na aplicação de técnicas de PLN no texto encontrado em elementos da requisição (título, descrição e comentários). A tarefa visa eliminar ruídos textuais a fim de extrair características que ajudem a representar o contexto da requisição alvo, facilitando selecionar, por meio de técnicas de recuperação da informação, requisições históricas similares.

Após a aplicação das técnicas de PLN para o pré-processamento da requisição alvo,

Figura 14 – Inércia medido no repositório do *Guava*



Fonte: o autor 2020

é iniciada a realização da recuperação de desenvolvedores.

Durante o pré processamento será realizado, no texto da requisição de alteração alvo, a tokenização, que segmenta as palavras das sentenças, permitindo a remoção de palavras que não contribuem muito com o contexto de uma sentença (*stop words*). Como exemplo, pode-se levar em consideração os textos da requisição apresentada na Figura 8:

```
"Add Bulk Delete API Blob Store Interface

Continuing from #39656 here:

Currently deleting snapshots can be a very slow process if the delete
entails removing a large number of blobs from the repository (i.e.
when one or more indices that consistent of many files becomes
unreferenced and has to be deleted).
The current implementation of the blob store only offers synchronous
```

```
and sequential deletes of these blobs, which means that one needs at
least as many network calls as the number of to be deleted blobs.
In addition to being potentially very slow, this also might incur
higher than necessary cost for blob stores cloud solutions that
bill for each API request.

There are at least two possible improvements to be made
to performance:

Make the delete API non-blocking so blobs can be removed in parallel
where possible. (Suggested implementation in #40144)
Use bulk deletion APIs where possible (e.g. S3's bulk delete)
I think we should do the latter here and adjust
org.elasticsearch.common.blobstore.BlobContainer#deleteBlob to accept
a list of blobs to delete and leverage bulk delete APIs in
implementations where possible."
```

Após a aplicação de tokenização e remoção de *stop words*, o resultado é:

```
Add Bulk Delete API Blob Store Interface. Currently deleting
snapshots slow process delete entails removing large number
blobs repository. one indices consistent many files becomes
unreferenced deleted .The current implementation blob store
offers synchronous sequential deletes blobs, means one needs
least many network calls number deleted blobs. In addition
potentially slow, might incur higher necessary cost blob stores
cloud solutions bill API request. There least two possible improvements
made performance: Make delete API non-blocking blobs removed
parallel possible. Suggested implementation # 40144 Use bulk deletion
APIs possible S3 bulk delete I think latter adjust
org.elasticsearch.common.blobstore.BlobContainer deleteBlob accept
list blobs delete leverage bulk delete APIs implementations possible.
```

No intuito de eliminar o tempo verbal das palavras para analisá-las em sua raiz, a realização do processo de lematização ocorre:

```
Add Bulk Delete API Blob Store Interface. Currently deleting
snapshot slow process delete entail removing large number
blob repository. one index consistent many file becomes
unreferenced deleted. The current implementation blob store
```

```
offer synchronous sequential deletes blob, mean one need
least many network call number deleted blobs. In addition
potentially slow, might incur higher necessary cost blob store
cloud solution bill API request. There least two possible improvement
made performance: Make delete API non-blocking blob removed
parallel possible. Suggested implementation # 40144 Use bulk deletion
APIs possible S3 's bulk delete I think latter adjust
org.elasticsearch.common.blobstore.BlobContainer deleteBlob accept
list blob delete leverage bulk delete APIs implementation possible.
```

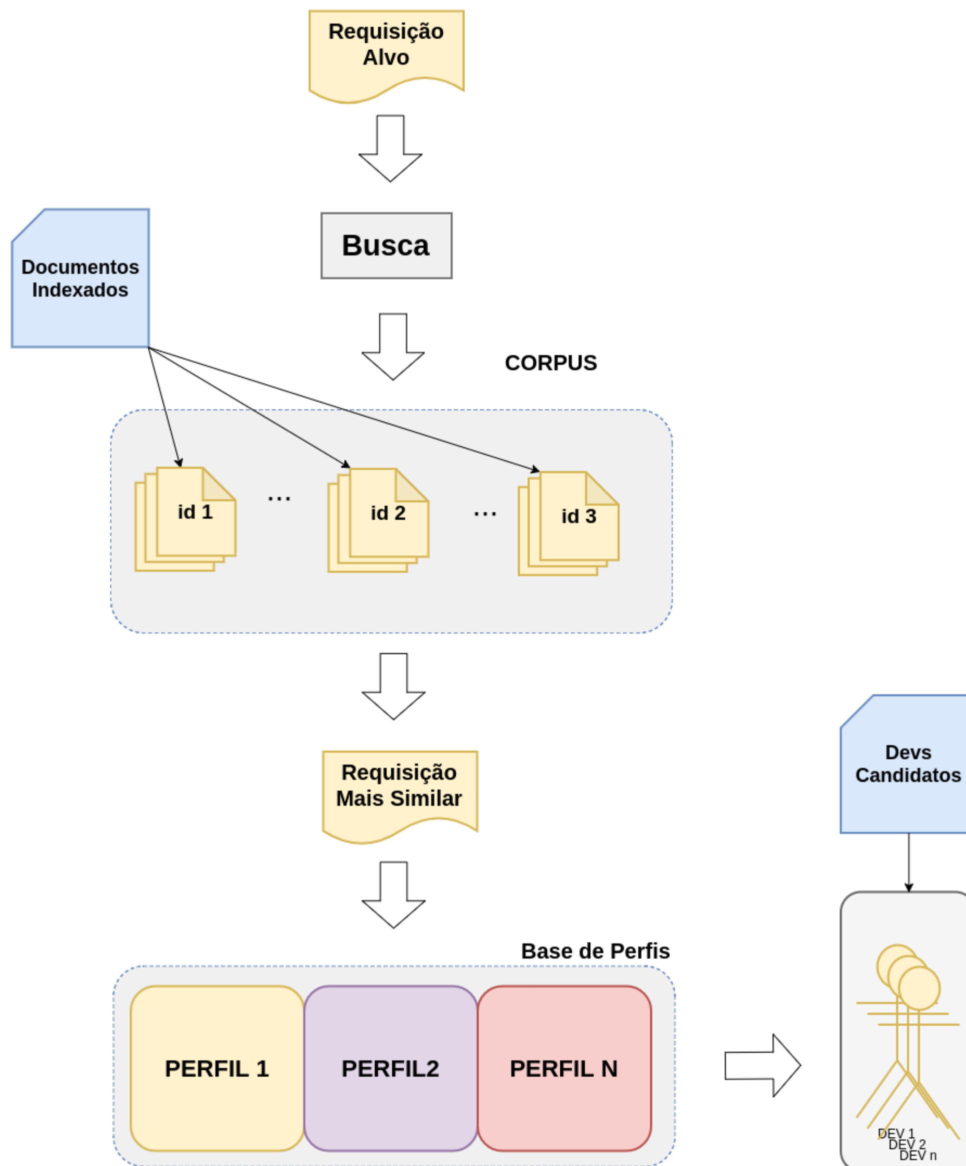
E por fim são obtidos listas de sinônimos das palavras do texto através do banco de dados léxico *Wordnet*, onde se encontra o termo e o grau de relação. Para a frase "Add Bulk Delete"do título, por exemplo. É obtido:

```
[('attention_deficit_disorder.n.01'),('add.v.01'),('add.v.02'),
('lend.v.01'),('add.v.04'),('total.v.02'),('add.v.06')],
[('majority.n.01'),('bulk.n.02'),('bulk.n.03'),
('bulk.v.01'),('bulge.v.04')],
[('delete.v.01'),('erase.v.03'),('edit.v.04')]]
```

Na abordagem proposta, os textos das requisições históricas são indexados e adicionados como documentos em um *corpus* criado na primeira etapa de Aquisição do método. Cada documento possui uma assinatura que representa o seu conteúdo textual. O texto da requisição alvo pré processado e indexado, também pela aplicação de LSI, é usado pelo método proposto para uma busca no corpus usando o texto da requisição alvo para selecionar a requisição resolvida com a maior similaridade semântica. Isso é possível pois os índices são usados para estabelecer a similaridade textual entre documentos, o que permite a realização de buscas por Similaridade de Cossenos. A busca funciona ao medir o espaço do produto interno entre dois vetores, no caso entre o vetor que representa a requisição alvo e todos os vetores que compõem a matriz LSI do *corpus*. A medida indica a direção dos vetores no espaço vetorial, representando a similaridade entre as suas respectivas requisições em meio às outras. O vetor com a maior Similaridade de Cosseno em relação ao vetor alvo é recuperado.

A partir da requisição encontrada, a abordagem proposta recupera da base de perfis a lista de desenvolvedores candidatos que tenham trabalhado historicamente em requisições pertencentes ao mesmo grupo da requisição encontrada, conforme é apresentado na figura Figura 15.

Figura 15 – Seleção de Desenvolvedores candidatos



Fonte: o autor 2020

5.2.3 Recomendação do Desenvolvedor

A última etapa do método tem como tarefa principal a recomendação do desenvolvedor mais apropriado para a atribuição da requisição alvo. Dada a lista de desenvolvedores candidatos para a resolução da alteração, é necessário selecionar o desenvolvedor ou os desenvolvedores mais apropriados. Tendo em vista a abordagem desse trabalho, é necessário levar em consideração o impacto na qualidade do código ao recomendar um desenvolvedor para a realização de uma alteração, logo a lista de candidatos é ordenada segundo as métricas de qualidade de código *commits* para as requisições do grupo da requisição buscada no *corpus* na etapa anterior. Seguindo a premissa de que o desenvolvedor mais

apropriado possui as métricas de qualidade mais positivas dentro de seu grupo.

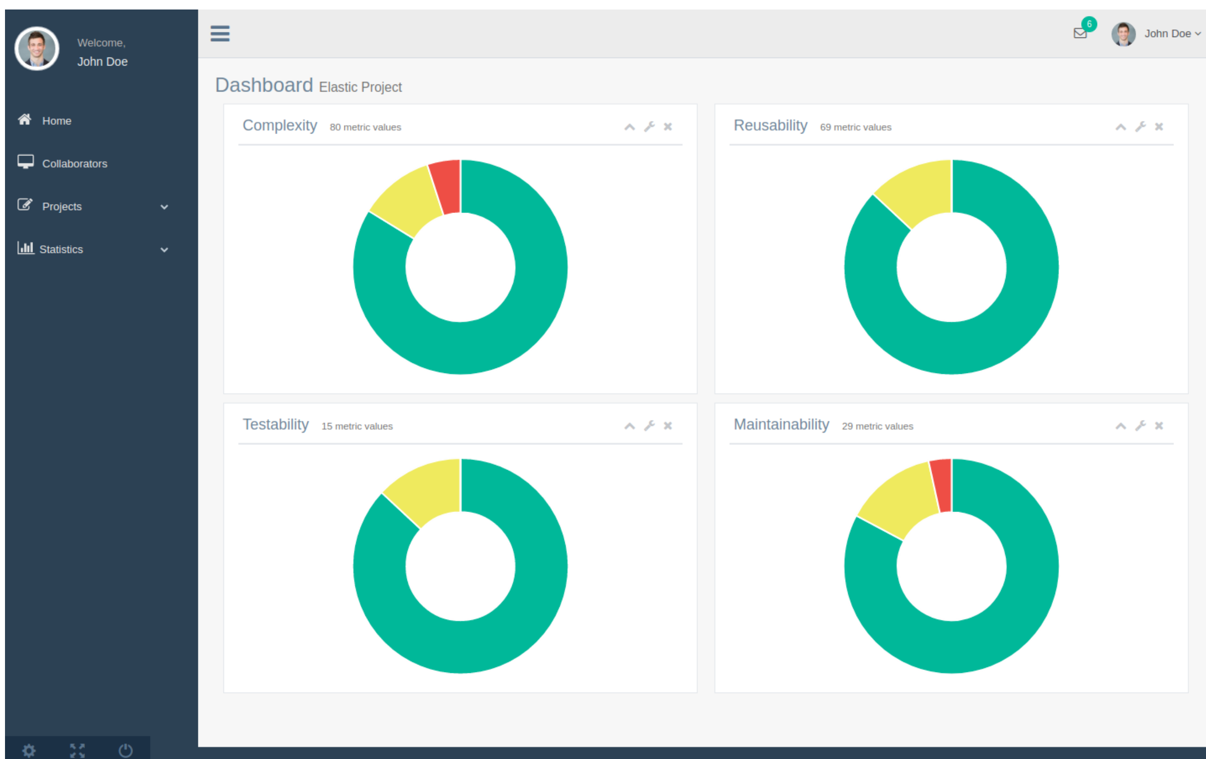
6 Resultados

Neste capítulo são apresentados os resultados experimentais desta pesquisa. Os principais resultados obtidos estão relacionados à construção do método de recomendação de desenvolvedores e à implementação de uma ferramenta visual para o suporte de qualidade de código colaborativo.

6.1 Método para o suporte de qualidade de código colaborativo

Um subproduto deste trabalho de pesquisa foi a criação de uma ferramenta visual (*Dashboard*), capaz de auxiliar desenvolvedores a terem uma melhor compreensão do impacto de seus *commits* na qualidade do código. A *Dashboard* apresenta diferentes aspectos das métricas de qualidade de software ao agrupar as métricas em grupos de influências. As métricas foram dividas em métricas de complexidade, manutenibilidade, reusabilidade e testabilidade, conforme mostra Figura 16.

Figura 16 – Captura de tela da Ferramenta Visual



Fonte: o autor 2020

A *Dashboard* também realiza recomendações de alterações em componentes de

código com baixa qualidade por meio de um Sistema Baseado em Regras. Com o auxílio de uma especialista, foi criado uma série de alterações que o sistema poderia recomendar a serem realizadas em um repositório de acordo com os intervalos dos valores de cada métrica calculada. Essas alterações podem ser realizadas em nível de Pacote, de Classe ou Método em repositório programados em Java. Esta parte do trabalho contribuiu para a publicação de (SILVA et al., 2019).

Abaixo segue as métricas avaliadas pela plataforma e as suas possíveis recomendações de alteração:

- **VG** - "O <método>, possui muitos pontos de decisão/seleção (if-then-else). Resultando em alta complexidade. Tente dividir o problema que o método resolve em problemas menores. Crie outros métodos e chame-os dentro do original."
- **PAR** - "O <método>, da Classe <classe>, possui muitos parâmetros. Significa que ele é muito especializado e pode ser difícil de ser reutilizado. Reduza o número de parâmetros, tente passar como parâmetro uma estruturada específica ou um objeto."
- **NBD** - "O <método>, da Classe <classe>, tem estruturas muito profundas. Aparentemente existem muitas estruturas dentro de outras, como loops, if-the-else, e ainda outras estruturas if-the-else dentro destas, e assim por diante. Isto significa que esta parte do código está muito especializada e pode ser difícil de ser reutilizada. Tente dividir o problema que o método resolve em problemas menores. Crie outros métodos e chame-os dentro do original."
- **CA** - "O <método> parece estar sendo utilizado por diversas classes que pertencem a outros pacotes. Estas classes podem ser muito dependentes destes pacotes. Isto aumenta o nível de acoplamento do código, dificultando seu reúso e manutenção."
- **CE** - "Muitas classes deste <pacote> aparentam estarem sendo utilizadas por outros pacotes. Isto aumenta o acoplamento do código, dificultando o reuso, manutenção e possíveis modificações futuras."
- **RMI** - "A instabilidade do <pacote> está baixa. Isto significa que esse pacote é muito dependente de outros pacotes. As alterações realizadas nos outros pacotes podem afetar este. Tente reduzir a dependência deste pacote em relação aos outros pacotes."
- **RMA** - "O número de classes abstratas (ou interfaces) neste <pacote> está baixo. Isto significa que é um pacote(projeto) concreto e não contém classes abstratas suficientes. Podendo ser difícil de ampliá-lo, reusá-lo ou mantê-lo. Seria melhor aumentar o número de classes abstratas neste pacote(projeto)."

- **DIT** - "A classe: <classe> está muito profundo na arvore de heranças das classes. Significa que que ela(ele) está herdando muitos métodos e atributos. Isto pode deixar muito complexo para uma manutenção futura. Tente reduzir a profundidade de herança nas classes."

- **WMC** - "O Método: <método> da Classe: <classe> aparentemente está muito complexo. Isto pode tornar esta classe difícil de se manter e reutilizar. Para cada método desta classe, tente criar outros métodos e chama-los dentro do original."

- **NSC** - "A Classe: <classe> tem muitos filhos. Quando uma classe tem muitos filhos pode ser bom para o reuso do código, mas muitos filhos podem indicar problemas de abstração nesta classe base. É melhor reduzir o numero de filhos nesta classe."

- **NORM** - "A Classe <classe> contém muitos métodos sobrescritos da classe pai. Muitos métodos redefinidos implicam em diferenças muito grandes entre a classe pai e a herança pode não fazer sentido. Tente criar novos métodos dentro da classe, ao invés de sobrescrever muitos métodos da classe pai."

- **LCOM** - "A Classe: <classe> tem problemas de coesão. As classes com problemas de coesão executam mais itens do que seus próprios objetivos, isto é, fazem outras tarefas. Falta de coesão implica em classes que provavelmente deveriam ter sido separadas em duas ou mais sub-classes."

- **NOF** - "A Classe: <classe> possui diversos atributos. Classes com este tipo de problema são difíceis de serem entendidas e mantidas. Tente reduzir o numero de atributos desta classe."

- **NSF** - "A Classe: <classe> possui muitos atributos estáticos. Um numero considerável de atributos estáticos pode dificultar a compreensão e manutenção do código. Tente criar variáveis locais com seus métodos setter e getter."

- **NOM** - "A Classe: <classe> possui muitos métodos. Isto pode indicar que a classe tem mais de um propósito além do seu próprio objetivo. Procure reduzir o número de métodos da classe."

- **NSM** - "A Classe: <classe> possui muitos atributos estáticos. Um numero considerável de atributos estáticos pode dificultar a compreensão e manutenção do código. Tente criar variáveis locais com seus métodos setter e getter."

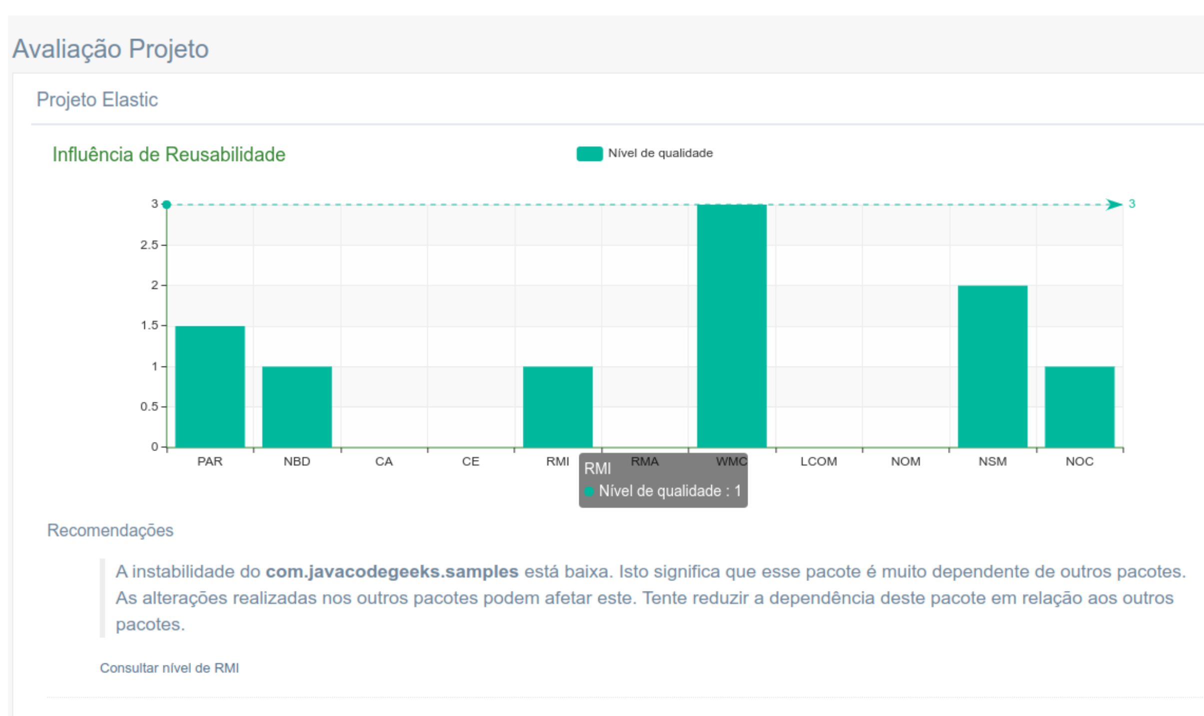
- **SIX** - "A Classe: <classe> tem profundidade de herança e sobrescreve muitos métodos da classe pai. Isto indica que o código é complexo e tende a ser difícil de manter. Pode ser um problema de abstração, as subclasses de uma herança estão sobrescrevendo diversos métodos da superclasse. É possível de ser considerado

um problema de abstração ou crie novos métodos dentro da classe, ao invés de sobrescrever diversos métodos da classe pai."

- **MLOC** - "O método <método> da Classe: <classe> tem muitas linhas de código. Métodos assim tornam-se complexos e difíceis de serem entendidos. Se possível, reduza o número de linhas do método."

A Figura 17 mostra um exemplo de recomendação de alteração para o repositório Elastic Search[1].

Figura 17 – Recomendação da Ferramenta Visual



Fonte: o autor 2020

6.2 Recomendação de Desenvolvedores

Após o término da Fase de Exploração, tratada no Capítulo 4, estavam definidos os recursos disponíveis em ambientes de desenvolvimento colaborativo. A partir desses recursos, foram estabelecidos os conjuntos de dados relevantes para a caracterização de desenvolvedores em relação ao seu comportamento durante o processo de desenvolvimento, mais especificamente a tarefa de resolução de Change Requests. O modelo de desenvolvedor proposto por este trabalho é composto por três conjuntos de características históricas

[1] https://github.com/elastic/elasticsearch

encontradas em repositórios de códigos: métricas de qualidade de código, dados do *commit* e atributos de Change Requests, como é descrito no Capítulo 5.

6.2.1 Ferramentas Utilizadas

As ferramentas que auxiliam na gestão de defeitos durante o desenvolvimento de software, tendem a representar as Change Requests de diferentes formas. No geral, as requisições apresentadas possuem certos elementos com o propósito de elucidar características da alteração à equipe de desenvolvimento, como é apresentado em Figura 7. Segundo o método proposto, para a atribuição de Change Requests, são considerados necessários somente o título, a descrição e os possíveis comentários exibidos em uma requisição, além do desenvolvedor a quem a requisição foi atribuída, ignorando elementos como imagens e rótulos.

Para a criação do modelo do desenvolvedor é necessário a mineração de repositórios de código. Porém, como o modelo proposto é baseado em três grupos de características, a aquisição da base de dados deve ser realizada em repositórios que possibilitem a extração de características de qualidade de código, de *commits* e da requisição de alteração relacionada ao *commits*.

Nesta pesquisa, foram utilizados projetos desenvolvidos na plataforma de controle de versionamento GitHub[2]. Os critérios para a seleção dos projetos foram: alto número de contribuidores; grande volume de *commits* e linguagem de programação Java. Esses critérios foram usados para estabelecer parâmetros que permitissem uma aquisição volumosa de dados em diferentes cenários. A mineração de repositórios de código é realizada em dois projetos de código-livre: Google[3] e Mockito[4]. O Quadro 3 detalha melhor os repositórios.

Quadro 3 – Repositórios selecionados para construção da base

Projeto	**Repositório**	**Desenvolvedores**	**Commits**	**Requisições Resolvidas**
Mockito	mockito	177	5.200	750
Google	guava	218	5.000	2.500
Google	closure-compiler	453	15.000	1.700

O GitHub possui uma ferramenta de gestão de erros chamada Issues, disponível em todos os projetos desenvolvidos em sua plataforma (GITHUBINC, 2008).

As requisições apresentadas na ferramenta contam com os seguintes elementos:

- Autor da requisição

[2] https://github.com/
[3] https://github.com/google/
[4] https://github.com/mockito/

- Título e Descrição
- Número da requisição
- Status da requisição (Resolvido ou em Aberto)
- Rótulos específicos para cada projeto
- Milestones (Etapas do projeto. Beta Launch por exemplo.)
- Assignee (Desenvolvedor Atribuído)
- Comentários de participantes

A plataforma GitHub também permite a mineração de repositórios de código-fonte, de *commits* e de requisições através de sua API[5], possibilitando a aquisição da base de dados para a criação do modelo de desenvolvedor. A API disponibiliza os dados no formato JSON, e tem um limite de 5.000 requisições de dados por hora.

Após a aquisição dos arquivos de códigos dos repositórios que sigam estritamente o paradigma de orientação a objetos, é iniciada a extração de suas métricas de qualidade. As métricas de qualidade de código que compõem o modelo de desenvolvedor deste trabalho são referentes à características que podem ser observadas no código, entretanto é necessário o uso de ferramentas para uma extração precisa das métricas em repositórios extensos.

Este trabalho utilizou uma ferramenta chamada CK(ANICHE, 2015) para extrair as métricas de qualidade de código propostas por Chidamber e Kemerer (CHIDAMBER; KEMERER, 1991) em repositórios desenvolvidos em Java sem precisar compilar (ANICHE, 2015). Originalmente, o CK não contemplava a extração de métricas em diferentes versões do mesmo repositório, portanto, a ferramenta teve que ser adaptada para comportar esse caso de uso. Por fim, a ferramenta extrai um conjunto de métricas de qualidade por *commit*, gerando um arquivo csv por *commit* correspondente, conforme indicado na Figura 18.

6.2.2 Experimentos

Atualmente, até onde vai nosso conhecimento, não existem bases de dados de código-fonte que realizaram seus processos de triagem, conforme proposto por este trabalho. Por consequência, isto dificulta a etapa de avaliação do método proposto, uma vez que não há um modelo-ouro para ser comparado. Portanto, avaliar somente a precisão do método proposto não é suficiente, pois os desenvolvedores de um conjunto de testes podem ter sido escolhidos por características subjetivas ou mesmo aleatoriamente (HERLOCKER et al., 2004).

[5] https://developer.github.com/v3/repos/commits/

Figura 18 – Exemplo de Arquivo de Extração da Ferramenta CK

class	type	cbo	wmc	dit	rfc	lcom	nosi	loc	nbd
com.google.thirdparty.publicsuffix.PublicSuffixPatterns	class	5	1	1	1	0	3	21	0
com.google.thirdparty.publicsuffix.TrieParser	class	8	21	1	16	3	7	60	3
com.google.thirdparty.publicsuffix.PublicSuffixType	enum	3	9	1	3	6	1	44	2
com.google.common.base.Functions$FunctionComposition	subclass	11	6	1	7	0	2	26	1
com.google.common.base.Functions$ForMapWithDefault	subclass	8	7	1	6	0	3	27	1
com.google.common.base.Functions$SupplierFunction	subclass	7	6	1	4	0	1	27	1
com.google.common.base.Functions$ToStringFunction	enum	2	2	1	2	1	1	9	0
com.google.common.base.Functions$FunctionForMapNoDefault	subclass	7	6	1	6	0	2	26	1
com.google.common.base.Functions$ConstantFunction	subclass	6	7	1	2	0	1	24	1
com.google.common.base.Functions$PredicateFunction	subclass	7	6	1	4	0	1	27	1
com.google.common.base.Functions	class	27	9	1	0	36	0	235	0
com.google.common.base.Functions$IdentityFunction	enum	2	2	1	0	1	0	8	0
com.google.common.base.Throwables$Anonymous1	anonymous	1	2	1	1	1	2	9	0

Fonte: o autor 2020

A métrica proposta em (BASSI et al., 2018) foi escolhida para avaliar se os desenvolvedores recomendados pelo método proposto teriam um impacto positivo na qualidade do código de uma alteração em comparação com o desenvolvedor que realmente resolveu a requisição. Essa métrica tem como objetivo medir o *Grau de Contribuição* de desenvolvedores em ambientes colaborativos. A métrica pode ser descrito em Equação 6.1 onde X é o valor de uma métrica e i é o *commit*:

$$\sum_{i=0}^{n} X_i - X_{i-1} \tag{6.1}$$

Se o resultado da diferença entre *commits* em sequência for alto, isso significa que existe um aumento durante um período no valor da métrica sendo avaliada e consequentemente uma influência negativa na saúde do repositório e um Grau ruim de Contribuição. Como por exemplo, se ocorrer um aumento de Acoplamento Entre Objetos ou, em inglês *Coupling Between Objects* (CBO), entre um intervalo de dois *commits* pode-se assumir que que houve um impacto negativo com a alteração realizada. Logo, os desenvolvedores recomendados pelo método devem possuir um Grau de Contribuição melhor ou igual em relação aos desenvolvedores que originalmente realizaram a alteração avaliada. Esse grau é aplicado para todas as métricas de cada instância e a média dos resultados é usada como comparativo.

Entretanto, também é preciso avaliar a relevância das recomendações do método proposto afim de validar os perfis dos desenvolvedores. Uma vez que a base de perfis é criada a partir do histórico de requisições resolvidas por colaboradores de um repositório, o desenvolvedor que originalmente resolveu a requisição de alteração avaliada deve estar pelo menos na lista de desenvolvedores candidatos, mesmo que ele não seja o mais apropriado. Para avaliar essa parte do método foi utilizada a métrica *recall@k*, conforme indica a

literatura(HOSSEN; KAGDI; POSHYVANYK, 2014). Segundo (HERLOCKER et al., 2004), *recall* é a proporção em que itens relevantes são encontrados nos itens recomendados e a sua modificação "@k"é referente à aplicação da métrica para conjuntos de itens recomendados com diferentes tamanhos.

A equação usada para avaliar *recall* em diferentes tamanhos de conjuntos é apresentada em Equação 6.2, onde N_{rs} é referente ao número de desenvolvedores recomendados que são os autores originais de alterações e N_r, que refere-se ao número de desenvolvedores recomendados:

$$Recall@k = \frac{N_{rs}}{N_r} \tag{6.2}$$

Primeiramente, a abordagem proposta foi avaliada em 456 Change Requests de três repositórios diferentes (de código-fonte escrito em Java), conforme apresentado em Quadro 3. Como o número de solicitações de mudança é relativamente pequeno, foi empregado um método exaustivo de validação cruzada, que usa todas as instâncias de uma base de dados como parte do conjunto de validação (*leaving-one-out*), para cada repositório separadamente. O modelo foi treinado usando todas as instâncias do conjunto de dados, exceto uma, e validado usando a instância específica deixada de fora. Esse processo é repetido para cada instância até que todas as possibilidades de divisão do conjunto de dados (conjunto de treinamento e conjunto de validação) tenham sido alcançadas. O método foi avaliado usando as métricas *Grau de Contribuição* (Equação 6.1) e *Recall @k* (Equação 6.2).

Para melhor avaliar o método proposto, seus resultados devem ser comparados com um método de recomendação do estado-da-arte. Para esta tarefa, selecionamos o método de recomendação de desenvolvedores *iMacPro*, pois ele requer apenas acesso ao código-fonte de um repositório e seu histórico de alterações. A abordagem *iMacPro* é um método de recomendação análogo ao método proposta por este trabalho, com a exceção mais notável sendo o uso de Propensão à Mudança (*Change Proneness*) de uma unidade relevante de código-fonte como um fator-chave para a recomendação de desenvolvedores em vez de métricas de qualidade.

O Quadro 4 apresenta os valores de *Recall @k* do método proposto, em comparação com o *iMacPro*. A medida *Recall @k* no caso do experimento realizado mede a proporção dos desenvolvedores relevantes (que realmente resolveram a requisição alvo) encontrados nos conjuntos de tamanho k recomendados. O valor da métrica *Recall* tende a aumentar à medida que o número de k aumenta para os dois métodos. O teste estatístico *t de Student* foi aplicado para determinar se as médias dos valores de *recall* produzidos por ambos os métodos são significativamente diferentes entre si. O *teste-t* realizado resultou em um valor-t de 0.30537 e valor-p de 0.382012. Como o valor de p obtido é superior a 0.05, não

há diferença estatisticamente significante entre os métodos comparados, o que indica que a abordagem proposta pode recomendar desenvolvedores, tão bem quanto uma abordagem bem estabelecida na literatura (*iMacPro*).

Ao considerar-se que não há uma diferença estatisticamente relevante das medidas de relevância entre os conjuntos de desenvolvedores recomendados pelo método proposto e pelo método encontrado na literatura *iMacPro*, pode-se concluir como aceita a Hipótese 1: "O método proposto recomenda desenvolvedores tão efetivamente quanto métodos presentes na literatura quando aplicados em bases de código-livre".

Quadro 4 – Comparação de *Recall@k* entre os métodos avaliados

	Recall@k	**Método Proposto**	**iMacPro**
Mockito	**1**	11%	13%
	3	40%	22%
	5	81%	29%
Closure-Compiler	**1**	13%	15%
	3	28%	48%
	5	50%	54%
Guava	**1**	11%	13%
	3	22%	28%
	5	25%	34%

Após a avaliação da efetividade do método proposto em recomendar conjuntos de desenvolvedores para resolver Change Requests, pode-se agora avaliar o impacto na qualidade do software que as recomendações em si podem ter. A Quadro 5 apresenta os valores da métrica *Grau de Contribuição* descrita em (BASSI et al., 2018) para os dois métodos de recomendação de desenvolvedores. Nesta avaliação, todos os desenvolvedores em um repositório tiveram seu *Grau de Contribuição* calculada. Em seguida, o *Grau de Contribuição* do desenvolvedor recomendado foi comparada ao responsável pela Change Request, se o *Grau de Contribuição* do desenvolvedor recomendado por um método for maior que o desenvolvedor que resolveu a requisição originalmente, o experimento marca a recomendação como "Positiva". Caso o *GC* seja menor, o experimento marca como "Negativa"e se não houver diferença marca como "Igual".

O teste estatístico *t de Student* foi aplicado em um subconjunto composto apenas por valores positivos, a fim de determinar se há uma diferença significativa entre as recomendações feitas pelos métodos avaliados em relação à qualidade do software. Os resultados obtidos no teste levam à conclusão de que há uma diferença significativa no $p < 0.05$, com um valor-t de aproximadamente 3.35 e um valor-p de aproximadamente 0.014.

Quadro 5 – Grau de Contribuição de Desenvolvedores

	Contribuição	**Método Proposto**	**iMacPro**
Mockito	**Igual**	17	20
150	**Negativa**	8	71
Change Requests	**Positiva**	125	59
Closure-Compiler	**Igual**	25	28
188	**Negativa**	3	96
Change Requests	**Positiva**	160	64
Guava	**Igual**	12	15
118	**Negativa**	15	62
Change Requests	**Positiva**	91	41

Os resultados da avaliação experimental dado no Quadro 5 mostram que 94.3% dos desenvolvedores recomendados têm níveis iguais ou superiores de *Grau de Contribuição* em relação aos desenvolvedores que resolveram as Change Requests avaliadas, em contraste com apenas 49% obtidos pelo método *iMacPro*. A diferença estatisticamente relevante entre esses dois conjuntos permite a aceitação da Hipótese 2 do trabalho proposto: "O método proposto recomenda mais desenvolvedores com impacto positivo na qualidade do código de um repositório comparado com métodos presentes na literatura".

O resultado dos experimentos realizados em repositórios de código aberto demonstra que a abordagem proposta pode ser benéfica para a saúde de um repositório, pois leva em consideração a qualidade das alterações anteriores de um desenvolvedor. Além disso, apresentou um *recall @k* tão bom quanto o do *iMacPro*, aludindo a um desempenho fundamentalmente semelhante ao obtido com métodos de recomendação de desenvolvedores do estado-da-arte, com o bônus de recomendar desenvolvedores com um histórico positivo em termos de qualidade do código para a resolução de uma alteração.

6.3 Aplicação do método proposto em ambiente privado

Com o interesse de continuar explorando a aplicabilidade do método proposto em um ambiente de desenvolvimento privado, despertando o interesse científico e econômico ao redor da proposta, foi realizado uma análise do método em um repositório fornecido por meio de parceria com uma empresa que atua na área de desenvolvimento de software para a geração, transmissão e distribuição de energia elétrica.

O repositório fornecido cumpre todos os requisitos dos pressupostos do métodos seção 5.1, entretanto é importante ressaltar que o repositório havia sido desenvolvido

utilizando o sistema de versionamento de código *Clearcase*, que possui uma arquitetura diferente do *Git*. Isso fez com que fosse possível somente a coleta das Change Request criadas após o repositório ter migrado para o *Git*.

O número total de Change Requests coletadas foi de 61 e os detalhes são apresentados abaixo:

- **Requisições sem alteração: 34**
- **Requisições não-resolvidas:17**
- **Requisições resolvidas:10**
- **Total de Requisições:61**

Durante a tarefa de Aprendizagem de Máquina do método, foi realizada uma análise em relação ao número mínimo de Change Request necessárias para a viabilidade do método. Para isso foram avaliados os valores da métrica *Recall@k* para aplicações do método em subconjuntos de diferentes tamanhos dos repositórios de código-livre.

Para valores abaixo de 50 requisições, notou-se um aumento anormal nos valores de **Recall@k**, conforme apresentado em Figura 19.

Os resultados da análise indicam que as recomendações do método proposto não são confiáveis para repositórios com números tão baixos de requisições resolvidas quanto aos do repositório disponibilizado pela empresa parceira. Isso se deve ao fato de que não exite uma distribuição de dados grande o suficiente para representar o contexto das alterações realizadas, fazendo com que o método recomende uma combinação dos mesmos desenvolvedores, sofrendo de *overfitting*.

Figura 19 – Subconjunto de 25% Repositório *Mockito*

```
==== MOCKITO ====
==> 150 Requisicoes
('Author Not Suited', 14)
('Equal', 17)
('Positive', 119)

('Recall@1', 17) --> 11%
('Recall@3', 60) --> 40%
('Recall@5', 122) --> 81%

===== 25% SAMPLE ======

===> 37 Requisicoes
('Author Not Suited', 8)
('Equal', 13)
('Positive', 16)
('Negative', 0)

('Recall@1', 13) -->  35%
('Recall@3', 26) -->  70%
('Recall@5', 29) -->  78%

=====
```

Fonte: o autor 2020

7 Conclusão

No contexto de ambientes de desenvolvimento colaborativo, o gerenciamento de artefatos relacionados à mudanças de requisitos é essencial para a manutenção da qualidade do código-fonte durante o ciclo de desenvolvimento de um projeto. Este trabalho se dispõe a criar um método de recomendação de desenvolvedores para a tarefa de resolução de Change Requests, baseado em perfis de desenvolvedores.

Além do método proposto, o trabalho obteve êxito na construção de uma base de dados referentes à *commits*, qualidade de código e Change Requests. Essa base conta com dados de 3 repositórios de código-livre. A construção da base e a implementação do método proposto permitiu a avaliação do método proposto ao compará-lo com um método de recomendação de desenvolvedores estabelecido da literatura chamado de *iMacPro*.

O resultado da fase de avaliação indica que o método possui fatores como por exemplo linguagem de programação e arquitetura do sistema de versionamento do repositório, que restringem o número de requisições que podem ser utilizadas. Entretanto, em repositórios com as condições específicas atingidas, foi mostrado que a abordagem proposta pode ser benéfica para a saúde de um repositório, pois leva em consideração a qualidade das alterações anteriores de um desenvolvedor. Além disso, apresentou um *recall @k* tão bom quanto o do *iMacPro*, aludindo a um desempenho fundamentalmente semelhante ao obtido com métodos de recomendação de desenvolvedores do estado-da-arte, com o bônus de recomendar desenvolvedores com um histórico positivo em termos de qualidade do código para a resolução de uma alteração.

Logo, conclui-se como comprovada a Hipótese H1: "O uso de modelagem objetiva de desenvolvedores para a tarefa de atribuição automatizada de Change Requests, fornece resultados superiores a métodos presentes na literatura quando aplicados em bases de código-livre". E como comprovada a Hipótese H2: "O método proposto recomenda mais desenvolvedores com impacto positivo na qualidade do código de um repositório comparado com métodos presentes na literatura".

Referências

ABDEL-HAFEZ, Ahmad; XU, Yue. A survey of user modelling in social media websites. *Computer and Information Science*, v. 6, p. 59–71, 09 2013. Citado na página 17.

AKTAş, Ethem; YILMAZ, Cemal. Automated issue assignment: results and insights from an industrial case. *Empirical Software Engineering*, 07 2020. Citado na página 16.

ALI, Naveed; LAI, Richard. A method of requirements change management for global software development. *Information and Software Technology*, Elsevier, v. 70, p. 49–67, 2016. Citado na página 18.

ALNANIH, Reem; ORMANDJIEVA, Olga; RADHAKRISHNAN, T. Context-based user stereotype model for mobile user interfaces in health care applications. *Procedia Computer Science*, v. 19, p. 1020 – 1027, 2013. ISSN 1877-0509. The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), the 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050913007503>. Citado na página 22.

ANDERSON, JL. Using software tools and metrics to produce better quality test software. In: IEEE. *AUTOTESTCON 2004. Proceedings*. [S.l.], 2004. p. 293–297. Citado 3 vezes nas páginas 18, 29 e 30.

ANICHE, Maurício. *Java code metrics calculator (CK)*. [S.l.], 2015. Available in https://github.com/mauricioaniche/ck/. Citado 2 vezes nas páginas 47 e 67.

BARTH, IJ. Modelando o perfil do usuário para a construção de sistemas de recomendação: um estudo teórico e estado da arte. *Revista de Sistemas de Informação da FSMA*, v. 6, p. 59–71, 2010. Citado 4 vezes nas páginas 23, 46, 53 e 56.

BASSI, P. R.; Wanderley, G. M. P.; Banali, P. H.; Paraiso, E. C. Measuring developers' contribution in source code using quality metrics. In: *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))*. [S.l.: s.n.], 2018. p. 39–44. Citado 4 vezes nas páginas 18, 44, 68 e 70.

BEAL, Franciele. *UM MÉTODO PARA A CONSTRUÇÃO DO PERFIL DINÂMICO DO DESENVOLVEDOR NO DESENVOLVIMENTO COLABORATIVO DE SOFTWARE*. Tese (Doutorado) — Pontifícia Universidade Católica do Paraná, 2014. Citado 2 vezes nas páginas 18 e 44.

BEAL, Franciele; BASSI, Patricia Rucker de; PARAISO, Emerson Cabrera. Developer modelling using software quality metrics and machine learning. In: HAMMOUDI, Slimane; SMIALEK, Michal; CAMP, Olivier; FILIPE, Joaquim (Ed.). *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Volume 1, Porto, Portugal, April 26-29, 2017*. SciTePress, 2017. p. 424–432. Disponível em: <https://doi.org/10.5220/0006327104240432>. Citado 4 vezes nas páginas 24, 35, 39 e 42.

BHATT, Pankaj; K, Williams; SHROFF, Gautam; MISRA, Arun Kumar. Influencing factors in outsourced software maintenance. *ACM SIGSOFT Software Engineering Notes*, v. 31, p. 1–6, 05 2006. Citado na página 29.

BISWAS, Pradipta. A brief survey on user modelling in human computer interaction. In: *Speech, Image, and Language Processing for Human Computer Interaction: Multi-Modal Advancements*. [S.l.]: IGI Global, 2012. p. 1–19. Citado na página 22.

BRUNTINK, Magiel; DEURSEN, Arie Van. Predicting class testability using object-oriented metrics. In: IEEE. *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. [S.l.], 2004. p. 136–145. Citado na página 29.

CAVALCANTI, yguaratã; MACHADO, Ivan; NETO, Paulo Anselmo da Motal S.; ALMEIDA, Eduardo Santana de. Towards semi-automated assignment of software change requests. *Journal of Systems and Software*, v. 115, 02 2016. Citado 5 vezes nas páginas 16, 17, 19, 37 e 42.

CHATZIGEORGIOU, Alexander; STEPHANIDES, George. Evaluating performance and power of object-oriented vs. procedural programming in embedded processors. In: SPRINGER. *International Conference on Reliable Software Technologies*. [S.l.], 2002. p. 65–75. Citado na página 20.

CHEN, Qiyang; NORCIO, AF. A neural network approach for user modeling. In: IEEE. *Conference Proceedings 1991 IEEE International Conference on Systems, Man, and Cybernetics*. [S.l.], 1991. p. 1429–1434. Citado na página 25.

CHIDAMBER, Shyam R; KEMERER, Chris F. Towards a metrics suite for object oriented design. Cambridge, Mass.: Center for Information Systems Research, Slan School of . . . , 1991. Citado 3 vezes nas páginas 30, 47 e 67.

CHIDAMBER, Shyam R; KEMERER, Chris F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 29.

CHOMSKY, Noam. *Syntactic structures*. [S.l.]: Walter de Gruyter, 2002. Citado na página 33.

CILOGLUGIL, Birol; INCEOGLU, Mustafa Murat. User modeling for adaptive e-learning systems. In: MURGANTE, Beniamino; GERVASI, Osvaldo; MISRA, Sanjay; NEDJAH, Nadia; ROCHA, Ana Maria A. C.; TANIAR, David; APDUHAN, Bernady O. (Ed.). *Computational Science and Its Applications – ICCSA 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 550–561. ISBN 978-3-642-31137-6. Citado na página 17.

CONSTANTINOU, Eleni; KAPITSAKI, Georgia M. Identifying developers' expertise in social coding platforms. In: IEEE. *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.], 2016. p. 63–67. Citado 3 vezes nas páginas 36, 39 e 44.

DEERWESTER, Scott; DUMAIS, Susan T; FURNAS, George W; LANDAUER, Thomas K; HARSHMAN, Richard. Indexing by latent semantic analysis. *Journal of the American society for information science*, Wiley Online Library, v. 41, n. 6, p. 391–407, 1990. Citado na página 52.

FENTON, Norman E; NEIL, Martin. Software metrics: roadmap. In: ACM. *Proceedings of the Conference on the Future of Software Engineering*. [S.l.], 2000. p. 357–370. Citado na página 18.

FILÓ, Tarcısio GS; BIGONHA, M; FERREIRA, K. A catalogue of thresholds for object-oriented software metrics. *Proc. of the 1st SOFTENG*, p. 48–55, 2015. Citado 3 vezes nas páginas 26, 29 e 30.

FISCHER, Gerhard. User modeling in human–computer interaction. *User Modeling and User-Adapted Interaction*, v. 11, n. 1, p. 65–86, Mar 2001. ISSN 1573-1391. Disponível em: <https://doi.org/10.1023/A:1011145532042>. Citado na página 22.

GAROUSI, Vahid; FELDERER, Michael; KıLıçASLAN, Feyza Nur. A survey on software testability. *Information and Software Technology*, v. 108, p. 35 – 64, 2019. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584918302490>. Citado na página 29.

GITHUBINC. *Mastering Issues*. 2008. Disponível em: <https://guides.github.com/features/issues/>. Citado 2 vezes nas páginas 49 e 66.

HALL, Mark A. Practical machine learning tools and techniques. 2005. Citado na página 23.

HENDERSON-SELLERS, Brian. *Object-oriented metrics: measures of complexity*. [S.l.]: Prentice-Hall, Inc., 1995. Citado 2 vezes nas páginas 29 e 30.

HERLOCKER, Jonathan L; KONSTAN, Joseph A; TERVEEN, Loren G; RIEDL, John T. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, ACM, v. 22, n. 1, p. 5–53, 2004. Citado 2 vezes nas páginas 67 e 69.

HJøRLAND, Birger. The foundation of the concept of relevance. *JASIST*, v. 61, p. 217–237, 02 2010. Citado na página 31.

HORSTMAN, C. *Big Java: Programming and Practice*. [S.l.]: New York, 2002. Citado 2 vezes nas páginas 29 e 30.

HOSSEN, Md; KAGDI, Huzefa; POSHYVANYK, Denys. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In: . [S.l.: s.n.], 2014. p. 130–141. Citado 4 vezes nas páginas 20, 38, 39 e 69.

HWANG, Wen-Jyi; WEN, Kuo-Wei. Fast knn classification algorithm based on partial distance search. *Electronics letters*, IET, v. 34, n. 21, p. 2062–2063, 1998. Citado na página 25.

IHARA, Akinori; OHIRA, Masao; MATSUMOTO, Ken-ichi. An analysis method for improving a bug modification process in open source software development. In: *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*. New York, NY, USA: ACM, 2009. (IWPSE-Evol '09), p. 135–144. ISBN 978-1-60558-678-6. Disponível em: <http://doi.acm.org/10.1145/1595808.1595833>. Citado na página 18.

INZUNZA, Sergio; JUÁREZ-RAMÍREZ, Reyes; JIMÉNEZ, Samantha. User modeling framework for context-aware recommender systems. In: ROCHA, Álvaro; CORREIA, Ana Maria; ADELI, Hojjat; REIS, Luís Paulo; COSTANZO, Sandra (Ed.). *Recent Advances in Information Systems and Technologies*. Cham: Springer International Publishing, 2017. p. 899–908. ISBN 978-3-319-56535-4. Citado na página 17.

ISO/IEC. *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): Measurement of System and Software Product Quality*. [S.l.]: ISO, 2016. Citado na página 25.

KAGDI, Huzefa; GETHERS, Malcom; POSHYVANYK, Denys; HAMMAD, Maen. Assigning change requests to software developers. *Journal of Software Maintenance*, v. 24, p. 3–33, 01 2012. Citado 4 vezes nas páginas 17, 19, 37 e 42.

LI, Huan. A novel coupling metric for object-oriented software systems. In: IEEE. *Knowledge Acquisition and Modeling Workshop, 2008. KAM Workshop 2008. IEEE International Symposium on*. [S.l.], 2008. p. 609–612. Citado 2 vezes nas páginas 29 e 30.

LOCK, Simon; KOTONYA, Gerald. An integrated, probabilistic framework for requirement change impact analysis. *Australasian Journal of Information Systems*, v. 6, n. 2, 1999. Citado na página 18.

LU, Qibei; GUO, Feipeng. Personalized information recommendation model based on context contribution and item correlation. *Measurement*, 2018. ISSN 0263-2241. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0263224118311497>. Citado na página 31.

MARCUS, Andrian; MALETIC, Jonathan I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: IEEE. *25th International Conference on Software Engineering, 2003. Proceedings*. [S.l.], 2003. p. 125–135. Citado na página 34.

MCCABE, Thomas J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976. Citado na página 26.

MCCONNELL, Steve. *Code complete*. [S.l.]: Pearson Education, 2004. Citado na página 25.

MCNUTT, Kathleen. Public engagement in the w eb 2.0 era: Social collaborative technologies in a public sector context. *Canadian Public Administration*, Wiley Online Library, v. 57, n. 1, p. 49–70, 2014. Citado na página 15.

MEYER, André N; ZIMMERMANN, Thomas; FRITZ, Thomas. Characterizing software developers by perceptions of productivity. In: IEEE PRESS. *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.], 2017. p. 105–110. Citado 3 vezes nas páginas 36, 39 e 42.

MISRA, Sanjay; AKMAN, Ibrahim. Weighted class complexity: a measure of complexity for object oriented system. *Journal of Information Science and Engineering*, v. 24, p. 1689–1708, 2008. Citado na página 29.

MISTRÍK, Ivan; GRUNDY, John; HOEK, André van der; WHITEHEAD, Jim. Collaborative software engineering: Challenges and prospects. In: ______. *Collaborative Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 389–403.

ISBN 978-3-642-10294-3. Disponível em: <https://doi.org/10.1007/978-3-642-10294-3_19>. Citado na página 42.

MITCHELL, William. A paradigm shift to oop has occurred... implementation to follow. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 16, n. 2, p. 94–105, 2001. Citado na página 20.

MOHTASHAMI, Mojgan; MARLOWE, Thomas J; KU, Cyril S. Metrics are needed for collaborative software development. *Journal of Systemics, Cybernetics, and Informatics*, v. 9, n. 5, p. 41–47, 2011. Citado na página 15.

NAGUIB, Hoda; NARAYAN, Nitesh; BRUEGGE, Bernd; HELAL, Dina. Bug report assignee recommendation using activity profiles. In: . [S.l.: s.n.], 2013. p. 22–30. ISBN 9781467329361. Citado 3 vezes nas páginas 16, 36 e 39.

NGUYEN, Loc. A proposal of discovering user interest by support vector machine and decision tree on document classification. In: IEEE. *2009 International Conference on Computational Science and Engineering*. [S.l.], 2009. v. 4, p. 809–814. Citado na página 24.

OLAGUE, Hector M; ETZKORN, Letha H; COX, Glenn W. An entropy-based approach to assessing object-oriented software maintainability and degradation-a method and case study. In: *Software Engineering Research and Practice*. [S.l.: s.n.], 2006. p. 442–452. Citado 2 vezes nas páginas 29 e 30.

OLIVEIRA, Marcio FS; REDIN, Ricardo Miotto; CARRO, Luigi; LAMB, Luís da Cunha; WAGNER, Flávio Rech. Software quality metrics and their impact on embedded software. In: IEEE. *Model-based Methodologies for Pervasive and Embedded Software, 2008. MOMPES 2008. 5th International Workshop on*. [S.l.], 2008. p. 68–77. Citado na página 29.

PAPATHEODOROU, Christos. Machine learning in user modeling. In: SPRINGER. *Advanced Course on Artificial Intelligence*. [S.l.], 1999. p. 286–294. Citado na página 24.

PAZZANI, Michael J.; BILLSUS, Daniel. Content-based recommendation systems. In: ______. *The Adaptive Web: Methods and Strategies of Web Personalization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 325–341. ISBN 978-3-540-72079-9. Disponível em: <https://doi.org/10.1007/978-3-540-72079-9_10>. Citado na página 31.

PIAO, Guangyuan; BRESLIN, John G. Inferring user interests in microblogging social networks: a survey. *User Modeling and User-Adapted Interaction*, Springer, v. 28, n. 3, p. 277–329, 2018. Citado na página 22.

PLOSCH, Reinhold; GRUBER, Harald; KORNER, Christian; SAFT, Matthias. A method for continuous code quality management using static analysis. In: IEEE. *2010 Seventh International Conference on the Quality of Information and Communications Technology*. [S.l.], 2010. p. 370–375. Citado na página 47.

POUSHTER, Jacob et al. Smartphone ownership and internet usage continues to climb in emerging economies. *Pew Research Center*, Smartphone ownership and internet usage continues to climb in emerging economies, v. 22, p. 1–44, 2016. Citado na página 15.

RAHMAN, M. M.; RUHE, G.; ZIMMERMANN, T. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2009. p. 439–442. ISSN 1949-3770. Citado na página 17.

RAKHA, Mohamed; BEZEMER, Cor-Paul; HASSAN, Ahmed E. Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval. *Empirical Software Engineering*, 12 2017. Citado na página 16.

RAWAT, Mrinal Singh; MITTAL, Arpita; DUBEY, Sanjay Kumar. Survey on impact of software metrics on software quality. *IJACSA) International Journal of Advanced Computer Science and Applications*, Citeseer, v. 3, n. 1, 2012. Citado 2 vezes nas páginas 18 e 26.

RICH, Elaine. Users are individuals: individualizing user models. *International Journal of Man-Machine Studies*, v. 18, n. 3, p. 199 – 214, 1983. ISSN 0020-7373. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020737383800078>. Citado na página 22.

RICHARDSON, Ita; CASEY, Valentine; BURTON, John; MCCAFFERY, Fergal. Global software engineering: A software process approach. In: *Collaborative software engineering*. [S.l.]: Springer, 2010. p. 35–56. Citado na página 42.

ROBINSON, Hugh; SHARP, Helen. Collaboration, communication and co-ordination in agile software development practice. In: *Collaborative software engineering*. [S.l.]: Springer, 2010. p. 93–108. Citado na página 42.

ROBU, R.; STOICU-TIVADAR, V. Arff convertor tool for weka data mining software. In: *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. [S.l.: s.n.], 2010. p. 247–251. Citado na página 47.

RUSSELL, Stuart J; NORVIG, Peter. *Artificial intelligence: a modern approach*. [S.l.]: Malaysia; Pearson Education Limited,, 2016. Citado na página 33.

SALTON, Gerard; MICHAEL, J. Mcgill. 1983. *Introduction to modern information retrieval*, 1983. Citado na página 34.

SARWAR, Badrul Munir; KARYPIS, George; KONSTAN, Joseph A; RIEDL, John et al. Item-based collaborative filtering recommendation algorithms. *Www*, v. 1, p. 285–295, 2001. Citado na página 31.

SCHWAB, Ingo; KOBSA, Alfred. Adaptivity through unobstrusive learning. *KI*, v. 16, n. 3, p. 5–9, 2002. Citado na página 24.

SILVA, DI De; KODAGODA, N; PERERA, H. Applicability of three complexity metrics. In: IEEE. *Advances in ICT for Emerging Regions (ICTer), 2012 International Conference on*. [S.l.], 2012. p. 82–88. Citado na página 29.

SILVA, Matheus Camilo da; BASSI, Patricia Rucker de; WANDERLEY, Gregory Moro Puppi; TACLA, Cesar Augusto; PARAISO, Emerson Cabrera. A visual tool for supporting collaborative codequality. In: HAMMOUDI, Slimane; SMIALEK, Michal; CAMP, Olivier; FILIPE, Joaquim (Ed.). *CSCWD 2019 - The 23nd IEEE International*

Conference on Computer Supported Cooperative Work in Design, Porto, Portugal, May 6-8, 2018. [S.l.]: SciTePress, 2019. Citado 2 vezes nas páginas 20 e 63.

SINGH, Gagandeep. Metrics for measuring the quality of object-oriented software. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 38, n. 5, p. 1–5, 2013. Citado na página 26.

SOLIMAN, Taysir Hassan A; EL-SWESY, Adel; AHMED, Saddam Hussein. Utilizing ck metrics suite to uml models: A case study of microarray midas software. In: IEEE. *2010 The 7th International Conference on Informatics and Systems (INFOS)*. [S.l.], 2010. p. 1–6. Citado na página 47.

STERLING, Leon; SHAPIRO, Ehud Y. *The art of Prolog: advanced programming techniques*. [S.l.]: MIT press, 1994. Citado na página 23.

SUN, Xiaobing; YANG, Hui; XIA, Xin; LI, Bin. Enhancing developer recommendation with supplementary information via mining historical commits. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 134, n. C, p. 355–368, dez. 2017. ISSN 0164-1212. Disponível em: <https://doi.org/10.1016/j.jss.2017.09.021>. Citado 3 vezes nas páginas 17, 37 e 44.

TURING, Alan M. Computing machinery and intelligence (1950). *The Essential Turing: The Ideas that Gave Birth to the Computer Age. Ed. B. Jack Copeland. Oxford: Oxford UP*, p. 433–64, 2004. Citado na página 33.

WANDERLEY, Gregory; RAMOS, Milton; TACLA, Cesar; SATO, Gilson Yukio; SILVA, Edenilson Da; PARAISO, Emerson. Modus-sd: User modeling in collaborative software development. In: . [S.l.: s.n.], 2012. p. 372–378. ISBN 978-1-4673-1211-0. Citado na página 22.

WASHIZAKI, Hironori; YAMAMOTO, Hirokazu; FUKAZAWA, Yoshiaki. A metrics suite for measuring reusability of software components. In: IEEE. *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*. [S.l.], 2003. p. 211–223. Citado na página 29.

WEN, Hao; FANG, Liping; GUAN, Ling. Modelling an individual's web search interests by utilizing navigational data. In: IEEE. *2008 IEEE 10th Workshop on Multimedia Signal Processing*. [S.l.], 2008. p. 691–695. Citado na página 25.

YU, Sheng; ZHOU, Shijie. A survey on metric of software complexity. In: IEEE. *2010 2nd IEEE International Conference on Information Management and Engineering*. [S.l.], 2010. p. 352–356. Citado na página 26.

ZHANG, Tao; CHEN, Jiachi; YANG, Geunseok; LEE, Byungjeong; LUO, Xiapu. Towards more accurate severity prediction and fixer recommendation of software bugs. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 117, n. C, p. 166–184, jul. 2016. ISSN 0164-1212. Disponível em: <https://doi.org/10.1016/j.jss.2016.02.034>. Citado 3 vezes nas páginas 17, 19 e 37.

ZHANG, Tao; LEE, Byungjeong. How to recommend appropriate developers for bug fixing? In: . [S.l.: s.n.], 2012. Citado na página 38.