



Universidade Federal do Piauí  
Centro de Ciências da Natureza  
Programa de Pós-Graduação em Ciência da Computação

# **Avaliação de Técnicas para a Identificação de Mantenedores de Código Fonte**

**Otávio Cury da Costa Castro**

**Teresina-PI, Fevereiro de 2021**



Otávio Cury da Costa Castro

## **Avaliação de Técnicas para a Identificação de Mantenedores de Código Fonte**

**Dissertação de Mestrado** apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Pedro de Alcântara dos Santos Neto

Coorientador: Guilherme Amaral Avelino

Teresina-PI

Fevereiro de 2021

---

Otávio Cury da Costa Castro

Avaliação de Técnicas para a Identificação de Mantenedores de Código Fonte/  
Otávio Cury da Costa Castro. – Teresina-PI, Fevereiro de 2021-  
76 p. : il. (algumas color.) ; 30 cm.

Orientador: Pedro de Alcântara dos Santos Neto

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação, Fevereiro de 2021.

1. Expertise de Código Fonte. 2. Mineração de Repositório de Software. 3. Aprendizagem de Máquina I. Pedro de Alcântara dos Santos Neto. II. Universidade Federal do Piauí. III. Programa de Pós-Graduação em Ciência da Computação. IV. Avaliação de Técnicas para a Identificação de Mantenedores de Código Fonte

CDU 02:141:005.7

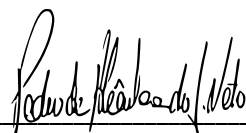
---

# **“Avaliação de Técnicas para a Identificação de Mantenedores de Código Fonte”**

**OTÁVIO CURY DA COSTA CASTRO**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

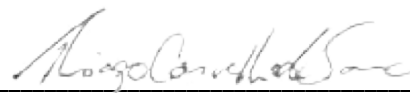
Aprovada por:



Prof. Pedro de Alcântara dos Santos Neto  
(Presidente da Banca Examinadora)



Prof. Marco Túlio Valente  
(Examinador Externo à Instituição)



Prof. Thiago Carvalho de Sousa  
(Examinador Externo à Instituição)



Prof. Guilherme Amaral Avelino  
(Examinador Interno)



Prof. Raimundo Santos Moura  
(Examinador Interno)

Teresina, 26 de fevereiro de 2021

*Ao meu irmão Arthur Cury (in memoriam), minha fonte de motivação.*

# Agradecimentos

Primeiramente, gostaria de agradecer a Deus pelas oportunidades e bênçãos postas em meu caminho. Agradeço a toda minha família. Agradeço aos meus pais, irmãos, tios e avós, pelo carinho e suporte essenciais em minha vida. Agradecimento especial à minha mãe Gemilia, minha fundação. Obrigado pela incansável dedicação à minha formação moral e intelectual.

Agradeço a todos meus professores que contribuíram e contribuem para o meu crescimento pessoal, acadêmico e profissional. Obrigado por todo o apoio e conhecimento transmitido durante minha vida de discente.

Em especial, agradeço aos meus orientadores Pedro de Alcantara e Guilherme Amaral. Obrigado pela paciência, confiança e companheirismo durante esses anos. Suas dedicações à minha orientação tornaram este trabalho possível.

Agradeço à CAPES pelo apoio financeiro destinado a realização desta pesquisa. Por fim, a todos meus amigos da Universidade Federal do Piauí. Obrigado pelos momentos de alegria e apoio por todos esses anos.

Muito Obrigado!





# Resumo

Durante o processo de evolução de um software, é comum que desenvolvedores acumulem conhecimento em determinados arquivos de código fonte que o compõe. Isso torna esses desenvolvedores os mais aptos entre os membros da equipe de desenvolvimento a realizarem futuras atividades que envolvam esses arquivos. Desta forma, a identificação desses mantenedores de código é uma informação importante nos ciclos de desenvolvimento de software. Essa informação pode ser utilizada em atividades como: implementação de novas funcionalidades, manutenção de código, assistência a novos membros da equipe, revisão de código, entre outros contextos da produção de software de qualidade. Porém, a identificação desses mantenedores de arquivo nem sempre é uma tarefa trivial, principalmente em projetos grandes onde um arquivo pode ter dezenas ou mesmo centenas de contribuidores durante sua história. Desse contexto, surge a necessidade da criação de técnicas cada vez mais eficazes na identificação automática desses mantenedores. Há, na literatura, estudos que têm como objetivo propor técnicas que utilizam de mineração de repositório de software para ajudar na identificação de mantenedores de arquivos. Porém, acredita-se que ainda existam lacunas nesta área de estudo que podem ser exploradas. Em trabalhos anteriores, foi identificado que informações tais como recência das modificações e o tamanho do arquivo em questão influenciam no conhecimento que determinado desenvolvedor tem com aquele arquivo. Neste trabalho, essas e outras informações extraídas do histórico de desenvolvimento foram combinadas em um modelo linear, e em classificadores de aprendizagem de máquina, para a identificação de mantenedores de arquivo. Essas novas técnicas foram comparadas com outras quatro existentes na literatura: (1) número de *commits*; (2) número de linhas adicionadas por um desenvolvedor na última versão do arquivo; (3 e 4) dois modelos lineares propostos em trabalhos da área. Os resultados mostram que os modelos propostos apresentam os melhores *F-Measures* nos cenários analisados, com os classificadores de aprendizagem de máquina atingindo os melhores *Precisions*. A partir dos resultados obtidos, conclui-se que a definição da melhor técnica depende do contexto da aplicação destas técnicas.

**Palavras-chaves:** Expertise de Código Fonte. Mineração de Repositório de Software. Aprendizagem de Máquina.



# Abstract

During the software evolution process, it is common for developers to accumulate knowledge in certain source code files that comprise it. This makes these developers the ablest among the members of the development team to carry out future activities involving these files. In this way, the identification of these code maintainers is important information in software development cycles. This information can be used in activities such as the implementation of new features, code maintenance, assistance to new team members, code review, among other contexts for the production of quality software. However, identifying these file maintainers is not always a trivial task, especially in large projects where a file may have dozens or even hundreds of contributors during its history. From this context, the need arises for the creation of increasingly effective techniques in the automatic identification of these maintainers. There are studies in the literature that aim to propose techniques that use software repository mining to assist in the identification of file maintainers. However, it is believed that there are still gaps in this area of study that can be explored. In previous works, it was identified that information such as recent changes and the size of the file in question influence the knowledge that a developer has with that file. In this work, this and other information extracted from the development history were combined in a linear model, and in machine learning classifiers, for the identification of file maintainers. These new techniques were compared with four others in the literature: (1) number of *commits*; (2) number of lines added by a developer in the latest version of the file; (3 and 4) two linear models proposed in works in the area. The results show that the proposed models present the best *F-Measures* in the analyzed scenarios, with the machine learning classifiers reaching the best *Precisions*.

**Keywords:** Source Code Expertise. Software Repository Mining. Machine Learning.



# Lista de ilustrações

Figura 1 – Histórico de <i>commits</i> do arquivo A. . . . .	12
Figura 2 – Histórico de <i>commits</i> do arquivo A. . . . .	13
Figura 3 – <i>Cross Validation K-Fold</i> com $K = 5$ . . . . .	19
Figura 4 – Distribuição das respostas no <i>dataset</i> público. . . . .	37
Figura 5 – Distribuição das respostas no <i>dataset</i> privado. . . . .	37
Figura 6 – Correlações entre as variáveis extraídas do histórico de desenvolvimento. . . . .	41
Figura 7 – Performance em cada <i>threshold</i> analisado usando dados dos projetos públicos. . . . .	52
Figura 8 – Performance em cada <i>threshold</i> analisado usando dados dos projetos privados. . . . .	53



# Lista de tabelas

Tabela 1	– Exemplo de matriz de confusão. . . . .	17
Tabela 2	– Resultados das buscas nas bases bibliográficas. . . . .	22
Tabela 3	– Estudos relacionados citados categorizados por objetivo. . . . .	27
Tabela 4	– Valores dos primeiros quartis das distribuições das métrica de filtragem para cada linguagem. . . . .	32
Tabela 5	– Características dos projetos <i>open-source</i> analisados. . . . .	33
Tabela 6	– Número de <i>commits</i> , arquivos e desenvolvedores dos dois projetos privados. . . . .	33
Tabela 7	– Quantidade de pares <i>desenvolvedor-arquivo</i> e desenvolvedores envolvidos na aplicação do <i>survey</i> , por tipo de projeto. . . . .	35
Tabela 8	– Informações sobre a quantidade de emails enviados e respostas recebidas. . . . .	36
Tabela 9	– Porcentagem de mantenedores e não-mantenedores. . . . .	36
Tabela 10	– Variáveis extraídas dos históricos de desenvolvimento. O significado das variáveis são apresentados considerando um desenvolvedor $d$ e um arquivo $f$ na sua versão $v$ . . . . .	38
Tabela 11	– Variáveis extraídas e trabalhos relacionados. . . . .	39
Tabela 12	– Correlação das variáveis extraídas com Conhecimento. . . . .	40
Tabela 13	– Performance das técnicas lineares nos seus melhores <i>thresholds</i> $k$ . . . . .	54
Tabela 14	– Performance dos classificadores de aprendizagem de máquina. . . . .	54
Tabela 15	– Correlações entre as 12 (doze) variáveis extraídas dos históricos de desenvolvimento. . . . .	76





# Lista de abreviaturas e siglas

DOA	Degree of Authorship
DOK	Degree of Knowledge
FA	First Authorship
FLOSS	Free, Libre and Open Source Software
FN	Falsos Negativos
FP	Falsos Positivos
FPS	File Path Similarity
GBM	Gradient Boosting Machine
IA	Inteligência Artificial
IIQ	Intervalo Interquartil
KNN	K Nearest Neighbors
LOC	Lines of Code
SCV	Sistema de Controle de Versão
SVM	Support Vector Machines
VP	Verdadeiros Positivos
VN	Verdadeiros Negativos
WRC	Weighted Review Count



# Lista de símbolos

$\cup$	União de conjuntos
$\cap$	Intersecção de conjuntos
$\sum$	Somatório de elementos
$\in$	Pertence
$\subseteq$	Está contido ou é igual a
$\geq$	Maior ou igual que
$\leq$	Menor ou igual que



# Sumário

<b>I</b>	<b>INTRODUÇÃO</b>	<b>1</b>
<b>II</b>	<b>ESTADO DA ARTE</b>	<b>9</b>
<b>1</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>11</b>
<b>1.1</b>	<b>Sistema de Controle de Versão</b>	<b>11</b>
<b>1.2</b>	<b>Expertise de Desenvolvedores</b>	<b>13</b>
<b>1.3</b>	<b>Aprendizagem de Máquina</b>	<b>14</b>
1.3.1	Algoritmos de aprendizagem de máquina	14
1.3.1.1	Random Forest	15
1.3.1.2	Gradient Boosting Machines	15
1.3.1.3	Support Vector Machines	15
1.3.1.4	K Nearest Neighbors	16
1.3.1.5	Logistic Regression	16
<b>1.4</b>	<b>Performance de Classificadores Binários</b>	<b>16</b>
1.4.1	Matriz de Confusão	17
1.4.2	Precision	17
1.4.3	Recall	17
1.4.4	F-Measure	18
<b>1.5</b>	<b>Validação Cruzada</b>	<b>18</b>
<b>2</b>	<b>ESTADO DA ARTE</b>	<b>21</b>
<b>2.1</b>	<b>Propostas de Novas Técnicas</b>	<b>22</b>
<b>2.2</b>	<b>Comparação de Técnicas</b>	<b>25</b>
<b>III</b>	<b>PROPOSTA</b>	<b>29</b>
<b>3</b>	<b>PROPOSTA</b>	<b>31</b>
<b>3.1</b>	<b>Coleta de Dados</b>	<b>31</b>
3.1.1	Projetos Open-Source	31
3.1.2	Projetos Privados	33
3.1.3	Extraindo Histórico de Desenvolvimento	33
3.1.4	Gerando Amostras para Aplicação do Survey	34
3.1.5	Aplicação do Survey	35
3.1.6	Processando as Respostas	36
<b>3.2</b>	<b>Extraindo e Analisando Variáveis de Desenvolvimento</b>	<b>36</b>

3.2.1	Variáveis . . . . .	37
3.2.2	Analizando as Variáveis Extraídas . . . . .	38
<b>3.3</b>	<b>Técnicas Comparadas . . . . .</b>	<b>40</b>
3.3.1	Commits . . . . .	41
3.3.2	Blame . . . . .	41
3.3.3	Degree of Authorship (DOA) . . . . .	42
3.3.4	CoDiVision . . . . .	42
3.3.5	Nível de Expertise . . . . .	44
3.3.6	Classificadores de Aprendizagem de Máquina . . . . .	45
<b>3.4</b>	<b>Metodologia de Avaliação . . . . .</b>	<b>46</b>
3.4.1	Utilizando Técnicas Lineares para Identificar Mantenedores . . . . .	46
3.4.2	Avaliando a Performance . . . . .	46

## **IV AVALIAÇÃO 49**

<b>4</b>	<b>RESULTADOS E DISCUSSÃO . . . . .</b>	<b>51</b>
4.1	Melhores Configurações das Técnicas Lineares . . . . .	51
4.2	Resultados . . . . .	52
4.3	Discussão . . . . .	53
4.4	Ameaças à Validade . . . . .	56
4.4.1	Validade de Construção . . . . .	56
4.4.2	Validade Interna . . . . .	57
4.4.3	Validade de Conclusão . . . . .	57
4.4.4	Validade Externa . . . . .	58
<b>5</b>	<b>CONCLUSÕES E TRABALHOS FUTUROS . . . . .</b>	<b>59</b>
5.1	Resumo . . . . .	59
5.2	Principais Achados . . . . .	59
5.3	Desafios e Limitações . . . . .	60
5.4	Trabalhos Futuros . . . . .	61

<b>REFERÊNCIAS . . . . .</b>	<b>63</b>
------------------------------	-----------

## **APÊNDICES 73**

<b>APÊNDICE A – CORRELAÇÕES ENTRE AS VARIÁVEIS EXTRAÍDAS . . . . .</b>	<b>75</b>
--	-----------

# Parte I

## Introdução





## Contexto e Motivação

Alterações no código fonte de software são atividades fundamentais durante o seu processo de evolução (RAJLICH, 2014). Essas alterações são realizadas em várias atividades relacionadas ao desenvolvimento. Entre essas atividades podemos citar: implementação de novas funcionalidades, resolução de *bugs*, adequações às novidades tecnológicas, entre outras. Tais atividades requerem um gerenciamento eficiente da equipe de desenvolvedores.

Esse gerenciamento se torna particularmente complexo em projetos grandes e geograficamente distribuídos, em que um gerente de projeto necessita do máximo de informações sobre o desenvolvimento para coordenar de maneira eficiente o trabalho da equipe (HERBSLEB; GRINTER, 1999). Dentre as diferentes informações necessárias, saber quais membros da equipe de desenvolvimento possuem conhecimento em quais arquivos de código fonte que compõe o projeto mostra-se bastante útil, especialmente em um contexto em que o trabalho remoto encontra-se em grande crescimento e as interações presenciais foram reduzidas.

A informação sobre o conhecimento de desenvolvedores com arquivos de código fonte pode ser útil em vários cenários ligados ao desenvolvimento de software. Essa informação pode ser utilizada em diversas atividades de designação de tarefas, entre elas:

- **Resolução de *bugs*:** identificar quais desenvolvedores possuem maiores *expertises* em arquivos envolvidos na resolução de um *bug* (ANVIK; HIEW; MURPHY, 2006; KAGDI et al., 2012).
- **Implementação de mudanças:** identificar quais desenvolvedores possuem maiores conhecimentos em arquivos relacionados a uma requisição de mudança (KAGDI; POSHYVANYK, 2009; KAGDI et al., 2012; HOSSEN; KAGDI; POSHYVANYK, 2014).
- **Revisão de código:** identificar desenvolvedores mais apropriados a realizar revisão em determinados arquivos (HANNEBAUER et al., 2016; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020).
- **Operação de *merge*:** identificar o desenvolvedor mais apto a realizar operações de merge de *branches* que envolvem determinados arquivos (COSTA et al., 2016a; COSTA et al., 2019a).
- **Ajuda a novos membros:** identificar quais desenvolvedores possuem conhecimento para auxiliar novos membros da equipe na resolução de atividades (KAGDI; HAMMAD; MALETIC, 2008; FRITZ et al., 2010; FRITZ et al., 2014).

Além do contexto de designação de atividades, essa informação é útil no monitoramento de como o conhecimento sobre o código fonte do projeto está distribuído entre

os membros da equipe de desenvolvimento. Isso ajuda a identificar a concentração de conhecimento em partes do código por certos desenvolvedores, mensurada por uma métrica chamada de *Truck Factor* (AVELINO et al., 2016; FERREIRA; VALENTE; FERREIRA, 2017; FERREIRA et al., 2019). Essa situação traz risco de descontinuidade do projeto, no caso da saída de determinados desenvolvedores.

No entanto, devido à grande quantidade de informações sobre mudanças no código que desenvolvedores e gerentes de projeto lidam todos os dias (FRITZ et al., 2014), é difícil saber quem é familiar com quais arquivos do software. Para ajudar nessa tarefa, uma alternativa é recorrer à informações presentes no Sistema de Controle de Versão (SCV) utilizado no projeto. Esses sistemas são responsáveis por registrar grande parte do histórico de interações *desenvolvedor-arquivo* (LOELIGER; MCCULLOUGH, 2012).

Utilizando tais dados, diversas técnicas foram desenvolvidas objetivando automatizar a identificação de *experts* em arquivos de código fonte (FRITZ et al., 2014; MCDONALD; ACKERMAN, 2000; MOCKUS; HERBSLEB, 2002; MINTO; MURPHY, 2007; SILVA et al., 2015). Esses desenvolvedores podem ser chamados de mantenedores de arquivo, por serem os desenvolvedores mais aptos a realizarem atividades de manutenção em determinadas partes do software (AVELINO et al., 2018; HOSSEN; KAGDI; POSHY-VANYK, 2014). A identificação automatizada desses mantenedores proporciona economia de tempo aos gerentes de projeto nas diversas atividades citadas anteriormente, comuns em desenvolvimento de software.

Em um trabalho relacionado (AVELINO et al., 2018) foi feita uma comparação do desempenho de três técnicas na identificação de mantenedores de arquivo: número de *Commits* (BIRD et al., 2011; CASALNUOVO et al., 2015), número de linhas adicionadas por um desenvolvedor na versão do arquivo (*Blame*) (GIRBA et al., 2005; RAHMAN; DEVANBU, 2011), e *Degree of Authorship* (DOA) (FRITZ et al., 2010; FRITZ et al., 2014). Um dos achados principais desse estudo foi que informações sobre o *tamanho dos arquivos* e a *recência das modificações*, que não são consideradas pelas técnicas comparadas, influenciam nas suas performances. Controlando essas variáveis, foi possível melhorar o desempenho das técnicas avaliadas.

Esses resultados sugerem que técnicas que utilizam tais variáveis podem ser mais precisas na identificação de mantenedores de arquivos. Isso encoraja a investigação e criação de novos modelos que avaliem o uso dessas e outras variáveis que podem ser extraídas de históricos de desenvolvimento presentes em SCVs na identificação de mantenedores.

## Definição do Problema

Como citado anteriormente, várias atividades comuns ao dia a dia do desenvolvimento de software baseiam-se na identificação de *expertise* em arquivos de código fonte.

Desta forma, o desenvolvimento de técnicas cada vez mais acuradas que automatizem essa identificação, e conseqüentemente otimizem essas atividades, traz grandes benefícios aos processos de desenvolvimento.

Mesmo sendo uma área de estudo bastante investigada, há lacunas que podem ser exploradas no desenvolvimento de novas técnicas que automatizem a identificação de mantenedores de arquivo. Essas novas técnicas podem combinar variáveis presentes no histórico de desenvolvimento que estão relacionadas com *expertise* de código fonte em modelos de inferência de conhecimento.

Para isso, é necessário um estudo que investigue quais informações extraídas de um histórico de desenvolvimento podem compor esses novos modelos. Além disso, é preciso verificar o quanto essas variáveis estão relacionadas com o conhecimento de desenvolvedores com arquivos de código fonte.

Dessa forma, um estudo que pretende tirar essas conclusões deve utilizar dados sobre histórico de desenvolvimento de diversos projetos de software de diferentes linguagens de programação. A partir dessas análises é possível a proposição de novas técnicas de identificação de *expertise* e a realização de comparações de performance com técnicas existentes na literatura da área.

## Visão Geral da Proposta

Este trabalho propõe novas técnicas que utilizam informações do histórico de desenvolvimento para identificar mantenedores de arquivos. Estes modelos foram baseados em dados extraídos do histórico de desenvolvimento de projetos de software.

Para isso, foram selecionados 111 projetos públicos e 2 projetos privados, desenvolvidos em 6 linguagens de programação diferentes. Desses projetos, foram extraídas informações de seus Sistemas de Controle de Versão.

Após essa seleção e extração de informações, foi aplicado um *survey* às amostras de contribuidores desses projetos, com o objetivo de coletar informações sobre o conhecimento desses desenvolvedores com arquivos de código fonte. Baseando-se nos dados coletados, foram analisadas as correlações de 12 variáveis do histórico de desenvolvimento com o conhecimento de desenvolvedores em arquivos. Desta análise, foram identificadas quais variáveis são mais importantes para a composição de modelos de conhecimento.

A partir deste estudo de correlação, foi proposto um modelo linear para o cálculo de conhecimento que determinado desenvolvedor tem em um arquivo. Utilizando das mesmas variáveis que compuseram tal modelo, foi proposta a utilização de classificadores de aprendizagem de máquina para a identificação de mantenedores de arquivos.

Por fim, realizou-se uma comparação da performance das técnicas propostas com 4

(quatro) técnicas existentes na literatura na identificação de mantenedores de arquivos. Foram utilizadas as técnicas: *Commits*, *Blame*, *Degree of Authorship* (DOA), e um modelo linear proposto em um trabalho do nosso grupo, denominado aqui de *CoDiVision*, que é o nome da ferramenta que o implementa.

## Objetivos

O objetivo principal deste trabalho é investigar a proposição de novas técnicas para a identificação de mantenedores de arquivos de código fonte. Adicionalmente, os desempenhos dessas novas técnicas são comparados com os desempenhos de outras presentes na literatura da área. Pode-se enumerar formalmente os objetivos principais como:

1. Propor um modelo linear de inferência de conhecimento de desenvolvedores em arquivos.
2. Propor a utilização de classificadores de aprendizagem de máquina na identificação de mantenedores de arquivo.
3. Comparar os desempenhos dos modelos propostos com trabalhos de relevância da área. As performances dessas novas técnicas são comparadas com as de outras 4 presentes na literatura: *Commits*, *Blame*, *Degree of Authorship*, *CoDiVision*.

Para alcançar estes objetivos principais, é possível identificar objetivos secundários a serem alcançados ao longo do trabalho, como:

1. Construir um *dataset* de projetos para o estudo. Esses projetos devem ser selecionados cuidadosamente, utilizando de critérios de relevância dos projetos e garantindo uma variabilidade adequada da amostra.
2. Obter dados com o conhecimento dos desenvolvedores em arquivos dos projetos selecionados. Este *dataset* foi construído por meio da aplicação de um questionário aos desenvolvedores.
3. Identificar quais variáveis extraídas de SCVs estão mais correlacionadas com o conhecimento de desenvolvedores em arquivos de código fonte.

## Estrutura do Trabalho

Na Introdução deste trabalho foram apresentados contexto e motivações, o problema que foi atacado, e quais foram os objetivos deste trabalho. O restante deste estudo é dividido em 5 (cinco) Capítulos.

No Capítulo 1 são apresentados os principais conceitos utilizados neste trabalho. Neste Capítulo, é explicado o funcionamento de um Sistema de Controle de Versão, quais tipos de *expertise* de desenvolvedores são encontradas na literatura e qual a estudada nesta dissertação, alguns conceitos de aprendizagem de máquina, os algoritmos que foram escolhidos para uso, e quais foram as métricas de performance utilizadas para comparar as técnicas.

No Capítulo 2 são discutidos os principais estudos relacionados a este trabalho. Esse capítulo é dividido em duas partes. A primeira discute trabalhos que apresentaram novas técnicas para a identificação de mantenedores de código, e a segunda apresenta trabalhos que tiveram como foco a comparação de técnicas existentes.

O Capítulo 3 descreve todo o processo necessário para a comparação de desempenho entre as técnicas. Ele descreve como os projetos foram selecionados, quais dados foram coletados e analisados, quais as variáveis escolhidas, como foram criados os modelos, quais foram as técnicas comparadas, e como foi feita a avaliação de desempenho dessas técnicas.

No Capítulo 4 são apresentadas as análises e comparações feitas, e discutidos os resultados e ameaças à validade deste estudo. Finalmente, o Capítulo 5 conclui o trabalho apresentando um resumo, os principais achados, as limitações identificados, para enfim, apontar direções para trabalhos futuros.



## Parte II

### Estado da arte





# 1 Fundamentação teórica

Este Capítulo apresenta conceitos necessários para o entendimento do trabalho realizado. A Seção 1.1 explica de forma resumida o que são e como funcionam Sistemas de Controle de Versão. A Seção 1.2 descreve diferentes tipos de *expertise* consideradas na literatura. A Seção 1.3 fala sobre aprendizagem de máquina, seus tipos, e os classificadores utilizados neste trabalho. A Seção 1.4 apresenta as métricas de performance de classificadores binários utilizadas. Finalizando, a Seção 1.5 explica o conceito de validação cruzada e seu funcionamento.

## 1.1 Sistema de Controle de Versão

Sistema de Controle de Versão (SCV) é um termo utilizado para descrever softwares responsáveis por manter o registro de diferentes versões de um arquivo (LOELIGER; MCCULLOUGH, 2012). Esses sistemas também podem ser referidos por outros nomes como: *Source Code Manager* (SCM), *Revision Control System* (RCS). Porém, neste trabalho esses sistemas vão ser referenciados como Sistemas de Controle de Versão.

Os benefícios da utilização de SCVs no processo de desenvolvimento de software são muitos, o que os torna quase indispensáveis na produção de software atualmente. Durante o desenvolvimento, diversas alterações nos arquivos que compõe o projeto são realizadas por diferentes desenvolvedores (OTTE, 2009). Desse fato surge a utilidade de um sistema que mantenha o histórico de quando, por quem, e quais alterações foram feitas nesses arquivos (SPINELLIS, 2005).

SCVs facilitam o compartilhamento dos arquivos de um projeto, acelerando e simplificando seu desenvolvimento. Desenvolvedores podem trabalhar em paralelo no mesmo código fonte e depois realizar *commits* das suas alterações. Se ocorrerem conflitos nas alterações realizadas em um arquivo, SCVs possuem mecanismos para lidar com isso, como operações de *merge*.

*Commits* possuem um papel fundamental no funcionamento de um SCV, sendo as operações que registram alterações nos arquivos de um projeto. Cada *commit* gera uma nova *versão* de cada arquivo alterado (SPINELLIS, 2005). A partir dessas versões, pode-se analisar o histórico de alterações realizadas nos arquivos do projeto em questão. É possível identificar quais as alterações feitas em cada linha que compõe o arquivo.

Observa-se na Figura 1 um exemplo de histórico de *commits* de um arquivo. Na figura, o arquivo *A* foi adicionado ao projeto no 1ª *Commit* pelo desenvolvedor *A*. Um segundo *commit* feito pelo desenvolvedor *B* modificou o arquivo *A*, e em um *commit*

subsequente, o arquivo *A* foi removido pelo desenvolvedor *B*. Adicionar (ADD), modificar (MOD) e remover (DEL) são tipos de operações que podem ser realizadas em um arquivo, referentes a inclusão do arquivo no sistema, a modificação do seu conteúdo, e a remoção do arquivo, respectivamente.

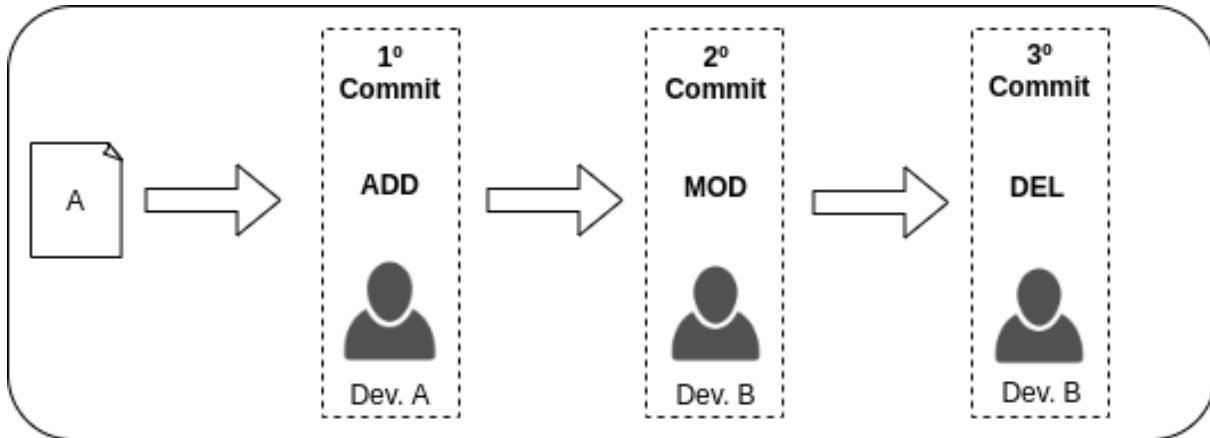


Figura 1 – Histórico de *commits* do arquivo *A*.

Entre o 1º *commit* feito pelo desenvolvedor *A* e 2º *commit* pelo desenvolvedor *B* representados na Figura 1, é executado *checkout* pelo desenvolvedor *B*. *Checkout* é a obtenção de uma versão dos arquivos de um sistema. Essa operação é bastante utilizada por desenvolvedores para manter os arquivos atualizados com as últimas alterações realizadas por outros membros da equipe de desenvolvimento.

Outro conceito importante no funcionamento de um SCV é o *branch*. *Branches* são linhas independentes de desenvolvimento de um mesmo software (PILATO; COLLINS-SUSSMAN; FITZPATRICK, 2008). Eles podem ser vistos como ramificações da linha principal de desenvolvimento chamada de *trunk/master*, da qual compartilham um histórico de *commits* em comum. Essas ramificações são úteis quando deseja-se realizar alterações no código de um software sem correr o risco de desestabilizar o código da linha principal, como por exemplo: criação de um software derivado, implementação de uma funcionalidade complexa.

A Figura 2 exemplifica uma situação onde o *branch* principal *Master* é ramificado em outros dois durante a sua história. Novos *branches* podem unir-se aos seus *branches* de origem por meio de uma operação chamada *merge*.

Todas as técnicas apresentadas e comparadas neste estudo baseiam-se em informações contidas em SCVs para identificar mantenedores de arquivos. O processo executado para extrair e processar as informações presentes nos SCVs dos projetos selecionados será detalhado no Capítulo 3.

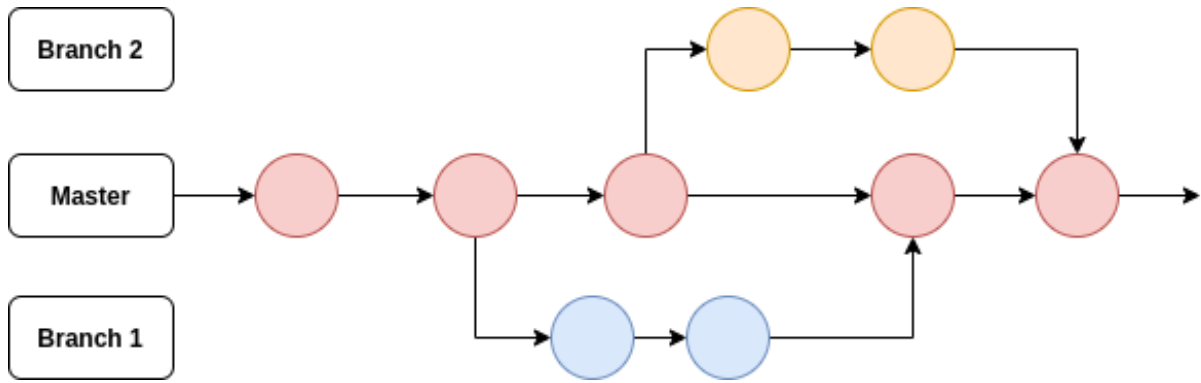


Figura 2 – Histórico de *commits* do arquivo A.

## 1.2 Expertise de Desenvolvedores

O termo *expertise* tem vários sinônimos utilizados na literatura da área como: experiência, conhecimento, autoria, familiaridade, propriedade (KRÜGER et al., 2018). Por ser um termo largamente adotado, decidiu-se por utilizá-lo em inglês neste trabalho.

Na literatura existem diferentes tipo de *expertise* que podem ser consideradas no contexto de desenvolvimento de software. Essas *expertises* diferenciam-se pelo tipo de conhecimento que uma pessoa acumula do artefato/processo em questão.

Alguns trabalhos focam na identificação de *Expertise de Revisão* (*Review Expertise*) (HANNEBAUER et al., 2016; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020). Ou seja, esses estudos focam na identificação de quais desenvolvedores em um projeto possuem mais experiência em revisar determinado código fonte. Dessa forma, esses estudos buscam automatizar o processo de designação de revisão (KIM; LEE, 2018). A automatização deste processo é um dos principais focos em pesquisas relacionadas à revisão de código (BADAMPUDI; BRITTO; UNTERKALMSTEINER, 2019).

Outros trabalhos propõe técnicas para a inferência de *Expertise de Uso* (*Usage Expertise*) (SCHULER; ZIMMERMANN, 2008; MANI; PADHYE; SINHA, 2016). Esta *expertise* está relacionada ao quanto determinado desenvolvedor sabe usar um artefato. Ou seja, o quanto ele entende do seu propósito e de como seus serviços podem ser utilizados para resolver problemas. Essa *expertise* está relacionada ao perfil de habilidades (*skill profiling*) de desenvolvedores (OLIVEIRA; VIGGIATO; FIGUEIREDO, 2019; MONTANDON; SILVA; VALENTE, 2019).

Porém, o trabalho apresentado nesta dissertação foca na identificação de *Expertise de Implementação* (*Implementation Expertise*) (ANVIK; MURPHY, 2007) ou *Expertise de Modificação* (*Modification Expertise*) (HANNEBAUER et al., 2016). Essa *expertise* diz respeito ao quanto um desenvolvedor está familiarizado com determinado código fonte, entendendo suas estruturas e funcionamento. Normalmente, esse tipo de conhecimento é adquirido a medida que desenvolvedores modificam o código fonte (ANVIK; MURPHY,

2007). A identificação de desenvolvedores com esse tipo de conhecimento pode ser feita em diferentes níveis de granularidade como: linhas, métodos, arquivos, pacotes. Este trabalho foca especificamente na identificação de *expertise* em nível de arquivo, ou seja, o quanto um desenvolvedor conhece, de maneira geral, o conteúdo de um arquivo de código fonte.

## 1.3 Aprendizagem de Máquina

Aprendizagem de máquina é uma subárea da Inteligência Artificial (IA) (RUSSELL; NORVIG, 2009). Ela tem como foco a construção de métodos que utilizam *experiência* para melhorar sua performance, ou fazer previsões mais acuradas em determinado domínio de conhecimento (LUGER, 2004; KULKARNI, 2012). Neste contexto, *experiência* significa informações sobre o passado disponíveis ao método.

Há basicamente três tipos de aprendizagem: supervisionada, não-supervisionada e semi-supervisionada. Na aprendizagem supervisionada, cada instância da base de dados utilizada pelo algoritmo durante o treinamento é fornecida com sua saída correta correspondente (rótulo), diferente da não-supervisionada onde essas saídas corretas não são fornecidas (KULKARNI, 2012). A aprendizagem semi-supervisionada utiliza uma pequena quantidade de dados rotulados com uma grande quantidade de dados não rotulados durante a fase de treinamento.

Se a saída pertencer a um conjunto finito de valores, o problema de aprendizagem será chamado de classificação. O problema é categorizado como classificação binária se houver apenas dois valores de saída possíveis (RUSSELL; NORVIG, 2009). Trazendo para o contexto deste estudo, classificar desenvolvedores em mantenedores e não mantenedores é um problema de classificação binária.

Aprendizagem de máquina é uma área de estudo em expansão, com aplicações em várias áreas do conhecimento humano (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018). Na engenharia de software, há utilização de aprendizagem de máquina em áreas como: qualidade de software (AL-JAMIMI; AHMED, 2013), previsão de custo de desenvolvimento (WEN et al., 2012), teste de software (DURELLI et al., 2019), previsão de defeitos (SHEPPERD; BOWES; HALL, 2014), entre outras. Dentre essas áreas, a utilização de algoritmos de aprendizagem de máquina em conjunto com mineração de repositório de software é um tópico emergente em engenharia de software (GÜEMES-PEÑA et al., 2018).

### 1.3.1 Algoritmos de aprendizagem de máquina

Esta Seção descreve de forma resumida os 5 algoritmos de aprendizagem supervisionada utilizados neste trabalho. Foram escolhidos 5 algoritmos conhecidos e aplicados em contextos semelhantes ao deste trabalho (MONTANDON; SILVA; VALENTE, 2019).

### 1.3.1.1 Random Forest

*Random Forest* é um algoritmo de aprendizagem de máquina baseado em árvores de decisão. Este algoritmo pode ser utilizado tanto para tarefas de regressão quanto para classificação. É um dos algoritmos mais utilizados devido a sua performance e robustez (ALI et al., 2012).

Resumidamente, o *Random Forest* cria várias árvores de decisão individuais e as combina para obter um maior poder preditivo (BREIMAN, 2001). Tanto as amostras dos dados quanto as variáveis que compõem cada árvore são escolhidas de forma aleatória. Quando apresentados novos dados, o algoritmo irá combinar as respostas dessas árvores individuais em um único resultado. A forma de combinar os resultados dependerá da natureza do problema.

Há várias aplicação deste algoritmo na engenharia de software como: predição de falhas (KAUR; MALHOTRA, 2008), qualidade de software (FOLLECO et al., 2008), estimativa de esforço (SATAPATHY; ACHARYA; RATH, 2016), *expertise* de desenvolvedores (MONTANDON; SILVA; VALENTE, 2019).

### 1.3.1.2 Gradient Boosting Machines

*Gradient Boosting Machine* (GBM) é um poderoso algoritmo de aprendizagem de máquina que vem demonstrando uma boa performance em diversas áreas de aplicação (NATEKIN; KNOLL, 2013). Pode-se encontrar algumas aplicações desse algoritmo na área de engenharia de software (SATAPATHY; ACHARYA; RATH, 2014; NASSIF et al., 2012; SATAPATHY; RATH, 2017).

Assim como o *Random Forest*, esse algoritmo combina o poder preditivo de vários modelos menores em um modelo robusto. Porém, diferente do *Random Forest* que combina vários modelos construídos de forma aleatória (*bagging*), esse algoritmo constrói vários de forma sequencial para que cada modelo melhore a performance dos anteriores (*boosting*). Normalmente os modelos utilizados são árvores de decisão.

### 1.3.1.3 Support Vector Machines

Máquinas de Vetores de Suporte (*Support Vector Machines*) é um algoritmo de aprendizagem supervisionado utilizado em problemas de classificação ou regressão (GUNN et al., 1998). É um algoritmo amplamente utilizado, em especial pela a sua performance em problemas que envolvem muitos atributos (*features*), e a sua robustez a *outliers*.

De maneira geral, este algoritmo se baseia em achar um hiperplano (função) que melhor separa os dados de duas classes. Isso é feito calculando a distância (margem) dos pontos mais próximos de ambas as classes ao hiperplano. O hiperplano ótimo é aquele que maximiza essa distância. Em outras palavras, o algoritmo acha uma fronteira que

maximiza a distinção entre duas classes. Para lidar com dados que não sejam linearmente separáveis, o algoritmo utiliza um *Kernel trick* para mapear matematicamente os dados em espaços de dimensões maiores e conseguir separá-los de forma mais fácil.

Esse algoritmo é bastante utilizado em pesquisas em engenharia de software. Pode-se encontrar aplicações dele em: predição de falhas (ELISH; ELISH, 2008), qualidade de software (XING; GUO; LYU, 2005), triagem de *bugs* (AHSAN; FERZUND; WOTAWA, 2009), *expertise* de desenvolvedores (MONTANDON; SILVA; VALENTE, 2019).

#### 1.3.1.4 K Nearest Neighbors

O *K Vizinhos Mais Próximos* (*K Nearest Neighbor*) é um algoritmo de aprendizagem de máquina supervisionado simples, que pode ser utilizado em problemas de classificação e regressão (PETERSON, 2009). Porém, esse algoritmo é comumente utilizado em problemas de classificação, tendo várias aplicações em engenharia de software (LUCCA; PENTA; GRADARA, 2002; ALKHALID; ALSHAYEB; MAHMOUD, 2010; HASANLUO; GHAREHCHOPOGH, 2016; ALKHALID; LUNG; AJILA, 2013).

O algoritmo baseia-se na similaridade entre os dados para realizar a classificação. Dada uma nova entrada que deve ser classificada, o algoritmo calcula a distância dessa entrada para todos os dados previamente classificados (rotulados). Essa distância pode ser calculada utilizando a distância euclidiana dos valores das *features* de cada dado (CHOMBOON et al., 2015). Após os cálculos das distâncias, o novo dado é predito como da classe majoritária nos *K* dados mais próximos.

#### 1.3.1.5 Logistic Regression

Regressão Logística (*Logistic Regression*) é uma técnica estatística que pode ser aplicada no contexto de aprendizagem de máquina. Essa técnica é normalmente utilizada em problemas de classificação binária (BISHOP, 2006).

De forma bem resumida, essa técnica utiliza uma função logística ajustada aos dados de treinamento, que retorna a probabilidade de determinado dado pertencer à classe principal do problema. Com a probabilidade associada a novos dados, é utilizado um *threshold* que determina se o dado pertence ou não a classe principal.

## 1.4 Performance de Classificadores Binários

Existem várias medidas que podem ser utilizadas na análise de performance de classificadores binários, como os utilizados neste trabalho. Nesta Seção, são explicadas algumas delas, mostrando como são obtidas, e em quais contextos são aplicadas.

### 1.4.1 Matriz de Confusão

Matriz de Confusão é uma maneira de organizar as informações sobre classificação feita por um algoritmo, de modo que facilite a análise da sua performance (POWERS, 2011). Em classificações binárias, essa tabela é formada por duas linhas e duas colunas que apresentam o número de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos. Na Tabela 1 é apresentado um exemplo de matriz de confusão para classificações binárias.

		Valores Reais	
		x	y
Valores Preditos	x	Verdadeiros Positivos	Falsos Positivos
	y	Falsos Negativos	Verdadeiros Negativos

Tabela 1 – Exemplo de matriz de confusão.

Na tabela apresentada, Verdadeiros Positivos (**VP**) indicam a quantidade de elementos da classe x que foram classificados como x. Os Verdadeiros Negativos (**VN**) indicam a quantidade de elementos da classe y que foram classificados como y. Falsos Positivos (**FP**) indicam a quantidade de elementos da classe y que foram classificados como elementos da classe x, e Falsos Negativos (**FN**) indica a quantidade de elementos da classe x que foram classificados como y. Fazendo o paralelo com a classificação binária realizada neste trabalho, as classes x e y são equivalentes a *mantenedores* e *não-mantenedores* respectivamente.

### 1.4.2 Precision

A Precisão (*Precision*) de um classificador diz respeito a proporção de predições positivas que estão de fato corretas (POWERS, 2011). Utilizando a Tabela 1, a Precisão calcula a proporção de elementos preditos como x (VP + FP), que de fato pertencem à classe x (VP). A Precisão é calculada da seguinte maneira:

$$Precision = \frac{VP}{VP + FP} \quad (1.1)$$

Precisão é uma boa métrica de performance a ser utilizada quando o custo de Falsos Positivos é alto no contexto de aplicação. Neste trabalho, essa métrica será referida apenas como *Precision*.

### 1.4.3 Recall

O *Recall* de um classificador é o valor que representa a proporção de predições positivas da quantidade total de elementos da classe (POWERS, 2011). Essa métrica serve para verificar o quanto determinado classificador consegue identificar elementos da classe



positiva. Utilizando a Tabela 1, o *Recall* calcula a proporção de elementos de  $x$  ( $VP + FN$ ), que foram preditos como  $x$  ( $VP$ ). O *Recall* é calculado da seguinte maneira:

$$Recall = \frac{VP}{VP + FN} \quad (1.2)$$

*Recall* é uma boa métrica de performance a ser utilizada quando o custo de Falsos Negativos é alto no contexto de aplicação.

#### 1.4.4 F-Measure

*F-Measure* (FM), também conhecida como *F1-Score*, é uma medida que busca informar o balanceamento entre *Precision* e *Recall* (POWERS, 2011). *F-Measure* é calculada com a seguinte média harmônica:

$$FM = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (1.3)$$

Essa métrica foi utilizada neste trabalho para avaliar a performance das técnicas comparadas na identificação de mantenedores de arquivo.

### 1.5 Validação Cruzada

Validação Cruzada (*Cross Validation*) é uma técnica utilizada para avaliar a habilidade de generalização de um modelo preditivo (BERRAR, 2019). Esta técnica é aplicada para evitar um problema conhecido como superadaptação (*overfitting*), situação em que um modelo performa muito bem para determinada base de dados, porém é ineficaz na predição de novos resultados (HAWKINS, 2004).

A Validação Cruzada se baseia na partição do conjunto de dados disponíveis em conjuntos complementares, de modo que se possa realizar a análise e estimação de modelos em um conjunto, chamado de *conjunto de treinamento*, e a validação dos modelos encontrados no conjunto restante, conhecido como *conjunto de teste* ou *conjunto de validação*.

Na Validação Cruzada K-Fold, a divisão do conjunto de dados é controlada por um parâmetro  $K$ . Este parâmetro especifica em quantos subconjuntos mutuamente exclusivos a divisão dos dados será feita. Realizada a divisão, um subconjunto será utilizado para estimar os modelos, e os  $K-1$  subconjuntos restantes serão utilizados como conjunto de validação. Esse processo é repetido  $K$  vezes, e em cada iteração um subconjunto diferente é escolhido para a estimação de modelos (HASTIE; TIBSHIRANI; FRIEDMAN, 2009). Ao final do processo, pode-se tirar uma média das  $K$  performances como performance final. A Figura 3 representa esse processo utilizando um  $K = 5$ .



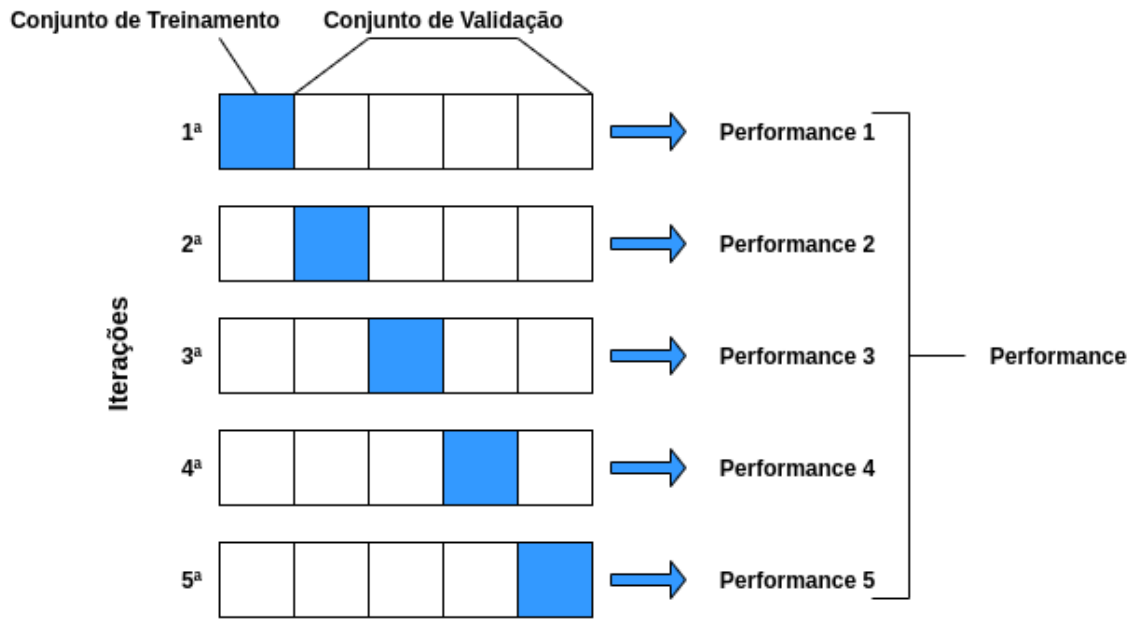


Figura 3 – *Cross Validation K-Fold* com  $K = 5$ .

Quando  $K = N$ , sendo  $N$  o número de instâncias do conjunto de dados, o método é conhecido como *Leave-One-Out Cross Validation*. Porém, as escolhas típicas de  $K$  são 5 ou 10, sendo os métodos conhecidos como *5-Fold Cross Validation* e *10-Fold Cross Validation*. De maneira geral,  $K = 5$  ou  $K = 10$  são recomendados como bons parâmetros (HASTIE; TIBSHIRANI; FRIEDMAN, 2009), sendo *10-Fold Cross Validation* utilizado nesta dissertação.



## 2 Estado da Arte

Esta dissertação propõe novas técnicas para a identificação de mantenedores de código fonte, e compara seus desempenhos com técnicas existentes na literatura. Dessa forma, este Capítulo foi dividido em duas Seções. A Seção 2.1 discute trabalhos que propõem técnicas que buscam mensurar o conhecimento de desenvolvedores com artefatos de código fonte. Na Seção 2.2 são descritos trabalhos voltados para a comparação de técnicas existentes na literatura. A Tabela 3 apresenta todos os estudos citados nessa Seção dividindo-os entres esses dois objetivos.

Para realizar a análise do estado da arte na utilização de técnicas para a inferência de conhecimento em código fonte a partir de informações contidas em repositório de código, foi feito um processo de busca de estudos com uma metodologia similar à de Mapeamentos Sistemáticos da Literatura (MSL). Um MSL é um método capaz de identificar e sumarizar trabalhos relevantes de uma determinada área de pesquisa (KEELE et al., 2007). A ideia é que a condução de determinado MSL seja não viesada, e seus resultados sejam replicáveis.

A busca foi realizada em 3 (três) bases de dados bibliográficas: Scopus<sup>1</sup>, Web of Science<sup>2</sup>, e IEEE Xplore<sup>3</sup>. A busca nestas bases é feita por meio de uma *string* de pesquisa. Essa *string* é composta por termos relacionados ao foco de pesquisa a ser explorado.

Inicialmente foram escolhidos termos principais a serem colocados na *string*. Foram escolhidos os termos: "developer", "knowledge", "code", "repository". Após a definição dos termos principais, foram incluídos sinônimos e termos associados encontrados na literatura da área. Foram adicionados os seguintes termos associados:

- **developer** - programmer;
- **knowledge** - familiarity, expertise, ownership, authorship, maintenance;
- **code** - file, class, package, module;
- **repository** - version control, revision control;

Os termos associados foram ligados por conectivos *OR*, e as *string* resultantes foram ligados por *AND*. A *string* de pesquisa final é apresentada a seguir:

*(developer\* OR programmer\*) AND (familiarity OR knowledge OR expertise OR ownership OR authorship OR maintenance) AND (code OR file OR class OR package OR module) AND (version control OR revision control OR repositior\*)*

<sup>1</sup> [www.scopus.com](http://www.scopus.com)

<sup>2</sup> <https://www.webofknowledge.com/>

<sup>3</sup> <https://ieeexplore.ieee.org/Xplore>

Essa *string* foi aplicada na busca por metadados (título, resumo e palavras chave) nas 3 (três) bases de dados bibliográficas mencionadas. As buscas foram limitadas a trabalhos na língua inglesa, e publicados a partir do ano 2000. Os estudos duplicados encontrados nas diferentes bases foram removidos com auxílio da ferramenta TheEnd<sup>4</sup>.

Base de Dados	Resultados
<i>Scopus</i>	475
<i>Web of Science</i>	369
<i>IEEE Xplore</i>	269
<b>Total</b>	1113
<b>Total sem duplicação</b>	663

Tabela 2 – Resultados das buscas nas bases bibliográficas.

A partir da busca realizada, foi feito um processo de busca dos estudos de interesse no conjunto resultante de 663 trabalhos. Este processo foi realizado em duas etapas. Primeiramente, foram lidos títulos e resumos dos trabalhos por dois pesquisadores, aplicando os seguintes critérios de inclusão:

- Critérios de inclusão:
  1. Utiliza alguma técnica para a inferência de conhecimento em código fonte a partir de informações de SCV **AND**;
  2. Publicado em revistas, jornais, ou trilhas de conferências **AND**;
  3. É um estudo completo;

Ao final da primeira etapa foram removidos 549 estudos que não atenderam a algum critério de inclusão, restando 114 estudos para serem lidos de forma mais detalhada. Na segunda etapa, foi realizada a leitura completa dos 114 estudos. Nesta etapa, foram aplicados os mesmos critérios de inclusão, onde foram removidos 83 estudos, restando 31 trabalhos, cujos principais serão discutidos nas Seções 2.1 e 2.2.

## 2.1 Propostas de Novas Técnicas

McDonald e Ackerman (2000) utilizam a heurística chamada *Line 10 rule* para identificar o desenvolvedor responsável por determinado *commit* e partir disso definir *expertise*. Eles baseiam-se na ideia que o desenvolvedor que alterou (*commit*) recentemente um artefato possui o maior conhecimento naquele artefato. Nesse trabalho, os autores propõem uma arquitetura chamada *Expertise Recommender* para indicar o desenvolvedor com o maior conhecimento em determinado arquivo. De maneira similar, Hossen, Kagdi

<sup>4</sup> <https://easii.ufpi.br/theend>

e [Poshyvanyk \(2014\)](#) apresentaram uma abordagem chamada *iMacPro* para recomendar desenvolvedores apropriados para resolver requisições de mudanças em software (*incoming change requests*). Parte dessa abordagem é responsável por determinar quem são mantenedores de arquivos associados a uma requisição de mudança. Para identificar mantenedores, os autores também utilizaram a premissa que desenvolvedores que alteraram recentemente são mais familiares com um arquivo.

Outros trabalhos levam em consideração granularidades menores de alterações ao identificar *expertise*. [Mockus e Herbsleb \(2002\)](#) consideram a quantidade de mudanças em nível de linha para rankear desenvolvedores por *expertise* utilizando uma ferramenta chamada *Expertise Browser*. [Girba et al. \(2005\)](#) e [Rahman e Devanbu \(2011\)](#) consideram um desenvolvedor possuidor de maior conhecimento em um arquivo se ele for o autor da maior porcentagem de linhas do arquivo. Já a ferramenta *Syde* proposta por [Hattori e Lanza \(2009\)](#), [Hattori e Lanza \(2010\)](#) cria um repositório de código que registra cada alteração feita por um desenvolvedor. Uma alteração é registrada a cada momento em que o desenvolvedor salva o arquivo com modificações, não precisando realizar um *commit* para isso. Essa ferramenta pode ser utilizada para identificar *experts* em um arquivo baseando-se na quantidade de pequenas mudanças feitas. Novamente, o desenvolvedor com maior conhecimento é aquele que fez o maior número de alterações em determinado arquivo.

Há estudos que combinam o histórico de *commits* com outras informações suplementares na identificação de *expertise* para diferentes propósitos. [Minto e Murphy \(2007\)](#) utiliza a quantidade de mudanças feitas por um desenvolvedor em um arquivo como medida de conhecimento, mas leva em consideração o relacionamento entre arquivos que foram alterados em conjunto. [Falcão et al. \(2020\)](#), em um estudo de fatores técnicos e sociais que influem na introdução de *bugs*, considera a quantidade de *commits* feitos pelo desenvolvedor e por outros desenvolvedores na inferência de conhecimento em arquivos. [Costa et al. \(2016b\)](#) e [Costa et al. \(2019b\)](#) consideram os *commits* feitos em diferentes *branches* para identificar familiaridade em arquivos de código fonte e recomendar desenvolvedores para a realização de operações de *merge*. Já outros trabalhos, como o proposto por [Sülün, Tüzün e Doğrusöz \(2019\)](#), utilizam a quantidade de *commits* no artefato de interesse e em outros artefatos relacionados para o cálculo do conhecimento, com o objetivo de recomendar revisores de código.

Todos os estudos citados nos parágrafos anteriores baseiam-se principalmente em informações sobre alterações como o número de *commits* e quem realizou a última mudança para identificar *expertise*. Acredita-se que apenas essas variáveis não são o suficiente para identificar *expertise*. Por esta razão, nesse trabalho foram analisadas mais variáveis e seus relacionamentos com o conhecimento de desenvolvedores.

[Fritz et al. \(2014\)](#) propuseram o modelo *Degree of Knowledge* (DOK). Esse modelo

calcula um valor real que representa o conhecimento de um desenvolvedor com um arquivo de código fonte (FRITZ et al., 2010). O DOK utiliza de informações relacionadas à autoria (*Degree of Authorship - DOA*) que o desenvolvedor tem com o arquivo, e a quantidade de interações (seleções e edições) que o desenvolvedor teve com o arquivo, chamada de nível de interesse (*Degree of Interest - DOI*) (MURPHY; KERSTEN; FINDLATER, 2006; KERSTEN, 2007). Os autores utilizaram dados do desenvolvimento de dois *sites* para verificar a eficiência do modelo em identificar *experts* em elementos de código fonte dos projetos. O cálculo do modelo DOK requer o uso de *plugins* instalados no ambiente de desenvolvimento, o que torna a sua utilização e comparação inviável para o estudo apresentado nessa dissertação. Com relação às diferenças para os modelos propostos nessa dissertação, o DOK não lida com a recência diretamente, e não considera o tamanho do arquivo no cálculo de conhecimento.

Há outras técnicas na literatura que tentam modelar o impacto do tempo no conhecimento que desenvolvedores têm com artefatos de código fonte. Silva et al. (2015) apresentam um modelo que determina a *expertise* de um desenvolvedor em dois níveis de granularidade: com um artefato inteiro (atômico), e com subpartes (classes internas e métodos) de um artefato. A *expertise* é computada baseada no número de mudanças (*commits*) feitas por um desenvolvedor. A análise de *expertise* pode ser feita usando janelas de tempo que dividem a história de um artefato em subconjuntos de *commits*. Isso permite uma análise de *expertise* em certos períodos da história de um arquivo.

Outras abordagens que levam em consideração a recência de modificações na determinação de *expertise* podem ser encontradas em estudos que focam na recomendação de desenvolvedores para a resolução de requisição de mudanças. Kagdi et al. (2012) propõem uma abordagem que localiza arquivos de código fonte relevantes a uma determinada requisição de mudança, e identifica *experts* nesses arquivos. Para identificar *experts*, os autores usam a abordagem *xFinder* (KAGDI; HAMMAD; MALETIC, 2008; KAGDI; POSHYVANYK, 2009) que priorizam desenvolvedores que fizeram o maior número de *commits* em determinado arquivo, e o quão recente em relação ao último *commit* feito no projeto que incluiu esse arquivo. Já Sülün, Tüzün e Doğrusöz (2020), estendendo um trabalho anterior (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019), adiciona informações sobre recência de modificação para o cálculo do conhecimento, objetivando recomendar revisores de código.

Os estudos citados acima utilizaram alguma medida de recência na identificação de *experts* em arquivos de código fonte. Porém, esses estudos não apresentaram uma análise que demonstre que as variáveis utilizadas são as mais apropriadas para a identificação. Por exemplo, todos os estudos utilizaram *número de commits* como medida de alteração que indica conhecimento. No entanto, essa não foi a variável que demonstrou maior correlação com o conhecimento em arquivos, como será discutido no próximo Capítulo.

Com relação a utilização de aprendizagem de máquina em contextos semelhantes, Montandon, Silva e Valente (2019) investigaram a performance de classificadores supervisionados e não supervisionados na identificação de *experts* em três bibliotecas *open-source*. Mesmo realizando um processo similar na coleta de dados e análise, os modelos apresentados nessa dissertação têm um propósito diferente. Essa dissertação apresenta classificadores para a identificação de *experts* em arquivos de código fonte enquanto Montandon, Silva e Valente (2019) utilizam-os para identificar *experts* no uso de bibliotecas e *frameworks*, utilizando *features* diferentes das utilizadas nessa dissertação.

Não foram identificados estudos que utilizam algoritmos de aprendizagem de máquina para a classificação de mantenedores de arquivo com base nas informações encontradas em SCV, como as utilizadas neste trabalho. Muitos exemplos de aplicação de aprendizagem de máquina no contexto de *expertise* de desenvolvedores podem ser encontrados em pesquisas da área de atribuição de *bugs* (*bug assignment*) (SAJEDI-BADASHIAN; STROULIA, 2020). Esses estudos utilizam técnicas de recuperação de informação (*information retrieval*) e/ou processamento de linguagem natural, e depois aplicam técnicas de aprendizagem de máquina.

## 2.2 Comparação de Técnicas

Avelino et al. (2018) realizaram uma comparação da performance das técnicas *Commits*, *Blame* e *Degree-of-Authorship* (DOA) na identificação de mantenedores de arquivos de código fonte. Um processo de aplicação de *survey* similar ao presente nesta dissertação foi feito para criar um *dataset* com dados de 8 projetos *open-source* e 2 projetos privados. Como resultado, a técnica DOA mostrou-se a mais recomendada para a identificação de mantenedores. Outro achado-chave do trabalho foi que as variáveis recência, tamanho do arquivo, e atividades extra repositório têm impacto significativo no conhecimento que desenvolvedores declararam possuir em arquivos de código fonte.

Krüger et al. (2018) analisaram empiricamente o impacto do esquecimento na familiaridade de desenvolvedores com código fonte, utilizando dados de 10 (dez) projetos *open-source*. Eles estudaram se a curva de esquecimento descrita por Ebbinghaus (EBBINGHAUS, 1885) pode ser aplicada no contexto de desenvolvimento de software, e quais outros fatores influenciam na familiaridade de um desenvolvedor com código fonte. Eles analisaram fatores como número de *commits*, mudanças realizadas por outros desenvolvedores, porcentagem de código escrito por um desenvolvedor presente na versão atual do arquivo, e o comportamento de acompanhamento de mudanças realizadas por outros desenvolvedores.

Hannebauer et al. (2016) compararam a performance de oito algoritmos para a recomendação de revisores de código. Desses oito algoritmos, seis eram baseados em

*expertise* por modificação, e dois eram baseados em *expertise* de revisão. Os seis algoritmos baseados em *expertise* por modificação foram: *Line 10 Rule* (MCDONALD; ACKERMAN, 2000), Número de alterações (*Number of Changes*) *Expertise Recommender* (MCDONALD; ACKERMAN, 2000), Propriedade de Código (*Code Ownership*) (GIRBA et al., 2005), *Expertise Cloud* (ALONSO; DEVANBU; GERTZ, 2008), e DOA. Os algoritmos baseados em *expertise* de revisão foram: *File Path Similarity* (FPS) (THONGTANUNAM et al., 2014), e um modelo proposto no trabalho chamado de *Weighted Review Count* (WRC). Eles utilizaram dados de quatro projetos FLOSS (*Free, Libre and Open Source Software*): Firefox, AOSP, OpenStack, e Qt. Os dois algoritmos baseados em *expertise* de revisão tiveram melhores performances que os algoritmos baseados em *expertise* de modificação, com o algoritmo WRC alcançando os melhores resultados.

Anvik e Murphy (2007) compararam duas abordagens para determinar desenvolvedores apropriados com *expertise* de implementação para a resolução de *bug reports*. Uma abordagem utiliza dados de repositórios de código para definir quais desenvolvedores são *experts* nos arquivos associados com o *bug report* usando a heurística *Line 10*. A outra abordagem utiliza dados presentes em *bug networks* como: lista *carbon-copy*, comentários, e informações de quem resolveu *bugs* anteriores. Foram utilizados dados de desenvolvimento da plataforma Eclipse<sup>5</sup>. Eles concluíram que a melhor abordagem depende do que os usuários buscam em termos de *precision* e *recall*.

Pode-se diferenciar o trabalho apresentado nesta dissertação aos trabalhos citados nos parágrafos anteriores em três aspectos: propósito, técnicas comparadas, e abrangência. Com relação ao propósito, dois trabalhos têm objetivos iguais a comparação de performance feita neste estudo: Avelino et al. (2018) e Anvik e Murphy (2007). Hannebauer et al. (2016) comparam técnicas para a recomendação de revisores de código, e Krüger et al. (2018) comparam a relação de alguns fatores com a familiaridade de desenvolvedores com código fonte.

Com relação a quais técnicas comparadas, Avelino et al. (2018), Krüger et al. (2018) e Hannebauer et al. (2016), utilizaram de técnicas presentes neste trabalho. Porém, o estudo apresentado nesta dissertação propõe novas técnicas, e adiciona o modelo proposto por Lira (2016) na comparação de performance. Finalizando, com relação a abrangência dos estudos, nenhum utilizou dados de quantidade similar de projetos analisados neste trabalho.

---

<sup>5</sup> <https://www.eclipse.org/eclipse/>



Objetivo do estudo	Estudos
<b>Novas Técnicas</b>	<p>(MCDONALD; ACKERMAN, 2000),          (HOSSEN; KAGDI; POSHYVANYK, 2014),          (MOCKUS; HERBSLEB, 2002), (GIRBA et al., 2005),          (RAHMAN; DEVANBU, 2011),          (HATTORI; LANZA, 2010), (MINTO; MURPHY, 2007),          (FALCÃO et al., 2020), (COSTA et al., 2016b),          (COSTA et al., 2019b), (HATTORI; LANZA, 2009),          (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019),          (FRITZ et al., 2014), (FRITZ et al., 2010),          (MURPHY; KERSTEN; FINDLATER, 2006),          (SILVA et al., 2015), (KAGDI et al., 2012),          (KAGDI; HAMMAD; MALETIC, 2008),          (KAGDI; POSHYVANYK, 2009), (KERSTEN, 2007),          (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020),          (MONTANDON; SILVA; VALENTE, 2019)</p>
<b>Comparação de Técnicas</b>	<p>(AVELINO et al., 2018), (KRÜGER et al., 2018),          (HANNEBAUER et al., 2016),          (ANVIK; MURPHY, 2007)</p>

Tabela 3 – Estudos relacionados citados categorizados por objetivo.



Parte III

Proposta



## 3 Proposta

Este Capítulo descreve todos os passos executados que levaram a propostas das técnicas e a posterior análise de seus desempenhos. A Seção 3.1 descreve como foram escolhidos os projetos *open-sources* e privados, como foram extraídos e analisados os históricos de desenvolvimento, e como foi feita a aplicação do *survey* com os desenvolvedores desses projetos.

A Seção 3.2 mostra quais variáveis foram extraídas do histórico de desenvolvimento e quais as suas correlações com o conhecimento dos desenvolvedores. A Seção 3.3 descreve as técnicas comparadas neste trabalho. Finalizando, a Seção 3.4 explica como foi realizada a classificação de mantenedores utilizando as técnicas lineares, e como foram analisadas suas performances.

### 3.1 Coleta de Dados

Esta Seção descreve como foram selecionados os 111 projetos *open-source* e os 2 projetos privados utilizados nesse estudo. O procedimento descrito é baseado nos adotados em trabalhos anteriores (AVELINO et al., 2016; AVELINO et al., 2018).

#### 3.1.1 Projetos Open-Source

Todos os dados de projetos *open-source* utilizados nesse estudo foram extraídos da plataforma GitHub<sup>1</sup>. Para realizar o download dos projetos, foi executado um procedimento similar aos encontrados em estudos com o mesmo objetivo (AVELINO et al., 2016; YAMASHITA et al., 2015; RAY et al., 2014). Primeiramente, foram escolhidas 6 linguagens de programação populares no GitHub: Java, Python, Ruby, JavaScript, PHP, e C ++<sup>2</sup>.

Em seguida foram escolhidos os 50 projetos mais populares de cada uma das 6 linguagens, baseando-se no número de estrelas que um repositório tem como medida de popularidade. Essa medida é amplamente utilizada por pesquisadores na seleção de projetos do GitHub (AVELINO et al., 2016; RAY et al., 2014; PADHYE; MANI; SINHA, 2014; HILTON et al., 2016; MAZINANIAN et al., 2017; JIANG et al., 2017; NIELEBOCK; HEUMÜLLER; ORTMEIER, 2019; RIGGER et al., 2018; CASTRO; SCHOTS, 2018), e vista por desenvolvedores como um bom indicador de popularidade (BORGES; VALENTE, 2018).

---

<sup>1</sup> <https://github.com/>

<sup>2</sup> As 6 linguagens de programação mais populares em 2019 <https://octoverse.github.com/#top-languages>

Após o download, foi executada uma etapa de filtragem dos projetos baseando-se em três métricas: Número de *Commits* ( $n_c$ ), Número de Arquivos ( $n_f$ ), e Número de Desenvolvedores ( $n_d$ ). Considerando os 50 projetos mais populares de uma linguagem ( $S_l$ ) do conjunto de linguagens escolhidas ( $L$ ), foram removidos projetos no primeiro quartil ( $Q_1$ ) das distribuições de cada uma das três métricas utilizadas na filtragem, resultando na interseção dos conjuntos restantes ( $T^0$ ). Em outras palavras, foram removidos projetos com poucos *commits*, arquivos e desenvolvedores. Na Tabela 4 estão apresentados os primeiros quartis de cada uma das três métricas para cada linguagem de programação.

Formalmente,

$$T^0 = \bigcup_{l \in L} T_{n_c}^0 \cap T_{n_f}^0 \cap T_{n_d}^0 \quad (3.1)$$

onde,  $T_{n_c}^0 = S_l - Q_1(n_c(S_l)); T_{n_f}^0 = S_l - Q_1(n_f(S_l)); T_{n_d}^0 = S_l - Q_1(n_d(S_l))$

Linguagem	Número de Commits	Número de Arquivo	Número de Desenvolvedores
<b>Python</b>	510,75	87,50	45,25
<b>Java</b>	829,25	318	39,25
<b>PHP</b>	823,5	89	97,5
<b>Ruby</b>	1650,25	152,75	198,25
<b>C++</b>	2010,25	706	113,50
<b>JavaScript</b>	1455,75	113,25	129,25

Tabela 4 – Valores dos primeiros quartis das distribuições das métrica de filtragem para cada linguagem.

Em seguida, foram removidos projetos cujo o histórico de desenvolvimento sugere que a maior parte do projeto foi desenvolvida fora do GitHub. Efetivamente, foram removidos projetos onde mais da metade dos arquivos criados foram adicionados em poucos *commits*. Para isso, foi computado o conjunto  $C_x$  formado pelo número de arquivos adicionados em cada *commit* do projeto  $X \in T^0$ . Logo após, foi identificado o subconjunto  $U_x$  dos *outliers* do conjunto  $C_x$ . Considerou-se *outliers* elementos que encontravam-se acima do terceiro quartil mais 3 (vezes) o intervalo interquartil ( $IIQ$ ). Com isso, buscou-se *commits* que se destacam muito com relação a adição de arquivos, baseando-se na distribuição dessa variável no projeto em questão. Se a soma dos arquivos adicionados nesses *outliers* fosse maior que a metade de todos os arquivos adicionados, o projeto era removido. O conjunto  $T^1$  resultante desse processo é formado pelos projetos *open-source* que participaram desse estudo. Formalmente,

$$C_x = \{a_1, a_2, a_3, \dots, a_n\} \quad (3.2)$$

$$U_x = \{a \in T_x \mid a > Q_3 + (3 * IIQ)\} \quad (3.3)$$

$$N = \{X \in T^0 \mid \sum_{a \in U_x} a > 0.5 * \sum_{a \in C_x} a\} \quad (3.4)$$

$$T^1 = T^0 - N \quad (3.5)$$

onde,

$$IIQ = Q_3 - Q_1$$

O conjunto resultante  $T^1$  é formado pelos 111 projetos mais relevante das 6 linguagens de programação escolhidas nesta dissertação, que possuem um número considerável de desenvolvedores, arquivos, e *commits*. A Tabela 5 sumariza as características dos 111 repositórios, de acordo com cada linguagem de programação.

Linguagem	Repos.	Devs.	Commits	Arquivos
Python	25	18.936	192.587	39.154
Java	17	5.733	138.473	62.429
PHP	16	9.802	144.092	29.902
Ruby	25	38.036	605.546	88.869
C++	14	11.467	350.345	72.991
JavaScript	14	10.319	109.541	21.477

Tabela 5 – Características dos projetos *open-source* analisados.

### 3.1.2 Projetos Privados

Foram utilizados dados do desenvolvimento de dois projetos da empresa de tecnologia Ericsson<sup>3</sup>. Ambos os projetos foram desenvolvidos na linguagem de programação Java. A Tabela 6 apresenta informações sobre as mesmas três métricas usadas para filtrar projetos *open-source*, como descrito na Seção 3.1.1. Como pode-se observar, os dois projetos privados são considerados relevantes de acordo com os mesmos critérios aplicados para filtrar os projetos *open-source*.

	Número de Commits	Número de Arquivos	Número de Desenvolvedores
<b>Projeto 1</b>	74.078	17.329	513
<b>Projeto 2</b>	26.678	15.930	262

Tabela 6 – Número de *commits*, arquivos e desenvolvedores dos dois projetos privados.

### 3.1.3 Extraindo Histórico de Desenvolvimento

Após o download e filtragem dos projetos, foi iniciado o processo de extração de dados do histórico de desenvolvimento de cada um desses projetos. Essas informações

<sup>3</sup> <https://www.ericsson.com/en>

foram extraídas a partir do histórico de *commits* do *master/default-branch* de cada projeto, como descrito a seguir.

Primeiramente, foi executado o comando `git log --no-merge --find-renames` para extrair dados dos *logs* de *commits* de cada projeto. Esse comando retorna todos os *commits* que não tem mais de um *parent*. Ou seja, esse comando não retorna *commits* originários de operações de *merge*. Além disso, esse comando identifica possíveis renomeações de arquivos<sup>4</sup>. De cada *commit* retornado por esse comando, foram extraídas três informações:

1. Os caminhos dos arquivos envolvidos.
2. O nome e email do autor do *commit*.
3. O tipo de mudança realizado em cada arquivo do *commit*: adição, modificação, ou renomeação.

A partir dessas informações, foram descartados arquivos que não contêm código fonte (e.g. imagens, documentação), bibliotecas de terceiros e arquivos que não faziam parte de uma das 6 linguagens de programação escolhidas neste estudo. Para isso, foi utilizada a ferramenta Linguist tool<sup>5</sup>.

Para lidar com *alias* de desenvolvedores, foi seguido o mesmo procedimento adotado em trabalhos relacionados (AVELINO et al., 2018; AVELINO et al., 2016; AVELINO et al., 2017). Cada *commit* no Git é caracterizado por um par (*nome-dev*, *email*). *Aliases* surgem do fato de um desenvolvedor estar associado a mais de um par. Dessa forma um mesmo desenvolvedor pode contribuir com o projeto com nomes e/ou emails diferentes. Porém, esse histórico de contribuições precisa ser unificado para os propósitos deste trabalho.

A unificação de um histórico de contribuições foi feita em duas etapas. Primeiramente, foram unificados usuários que compartilham o mesmo email, com nomes de usuário distintos. Adicionalmente, foram agrupados usuários com emails diferentes, mas com nomes de usuário semelhantes. Essa semelhança foi encontrada aplicando a distância de *Levenshtein* (NAVARRO, 2001) utilizando um *threshold* máximo de modificação de 30%.

### 3.1.4 Gerando Amostras para Aplicação do Survey

As informações extraídas dos históricos de desenvolvimento dos projetos foram utilizadas para criar amostras de pares (*desenvolvedor*, *arquivo*) de cada projeto. Essas amostras foram usadas para criar o *survey* aplicado nesse estudo. As amostras foram geradas executando os seguintes passos para cada um dos projetos:

<sup>4</sup> <https://git-scm.com/docs/git-log/1.5.6>

<sup>5</sup> <https://github.com/github/linguist/>



1. Um arquivo é selecionado aleatoriamente, e é obtida a lista de desenvolvedores que o tocaram (criar ou modificar).
2. O arquivo era descartado caso ao menos um dos desenvolvedores tivesse alcançado o *limite máximo de arquivos*. Caso contrário, o arquivo era adicionado na lista daquele desenvolvedor.
3. Os passos 1-2 foram repetidos até não haver mais arquivos a serem escolhidos.

A razão para o possível descarte do arquivo no passo 2 se dá pelo desejo de obter informações de todos os desenvolvedores que o tocaram. Porém, foi estabelecido um *limite máximo de arquivo* de 5, buscando não desencorajar os desenvolvedores de responderem o *survey*, o que pode acontecer de acordo com *guidelines* presentes na literatura (KASUNIC, 2005; LINAKER et al., 2015). Dessa forma, arquivos nos quais um dos contribuidores ultrapassava esse limite eram descartados, pois não havia possibilidade de obter respostas de todos os seus contribuidores.

Ao final dessa etapa, uma lista de pares (*desenvolvedor, arquivo*) foi criada para cada projeto. Na Tabela 7, pode-se observar dados sobre a quantidade de pares gerados e desenvolvedores envolvidos dos projetos públicos e privados.

	Quant. Pares	Desenvolvedores
<b>Públicos</b>	20.564	7.803
<b>Privados</b>	394	92

Tabela 7 – Quantidade de pares *desenvolvedor-arquivo* e desenvolvedores envolvidos na aplicação do *survey*, por tipo de projeto.

### 3.1.5 Aplicação do Survey

Foi enviado um email para cada desenvolvedor nas amostras geradas, pedindo para eles avaliarem seu conhecimento em cada um dos arquivos da sua lista. Cada desenvolvedor foi convidado a especificar seu conhecimento em cada arquivo utilizando uma escala de 1 (um) a 5 (cinco), onde:

“(1) significa que você não tem conhecimento sobre o código fonte do arquivo; (3) significa que você precisa realizar uma investigação antes de reproduzir o código; e (5) significa que você é um *expert* nesse código”

Foi explicitado que as respostas seriam utilizadas apenas para propósitos acadêmicos. Informações sobre a quantidade de emails enviados e de respostas recebidas podem ser vista na Tabela 8.

Nota-se uma diferença substancial entre a taxa de resposta obtida com projetos públicos e privados. Baixos percentuais de respostas também foram obtidos em outros

estudos que envolvem a aplicação de *surveys* online com desenvolvedores de projetos públicos (AVELINO et al., 2018; MONTANDON; SILVA; VALENTE, 2019). Essa diferença pode ser explicada por dois fatores que contribuem com a adesão a *surveys* (SMITH et al., 2013; GHAZI et al., 2018). O primeiro é a falta de 'recompensas', uma vez que pessoas tendem a responder questionários que prometem algum tipo de ganho (fator mencionado como *Compensation Value*). O segundo vem do fato de que o questionário foi aplicado aos desenvolvedores dos projetos privados por um funcionário da empresa em questão, sendo que pessoas tendem a responder questionários aplicados por conhecidos (fator mencionado como *Linking*).

	Enviados	Respostas	Porcentagem
<b>Open-source</b>	7.803	501	7%
<b>Privados</b>	92	38	41%

Tabela 8 – Informações sobre a quantidade de emails enviados e respostas recebidas.

Das respostas obtidas dois *datasets* foram criados: um com 1024 pares (*desenvolvedor, arquivo*) vindos das respostas dos desenvolvedores dos projetos *open-source*, e um segundo com 163 pares extraídos das respostas dos desenvolvedores dos projetos privados. No restante deste trabalho, esses dois *datasets* vão ser chamados de *dataset* público e *dataset* privado.

### 3.1.6 Processando as Respostas

Das respostas do *survey*, cada par (*desenvolvedor, arquivo*) foi classificado em um de dois conjuntos disjuntos: mantenedores declarados ( $O_m$ ), e não-mantenedores declarados ( $O_{\bar{m}}$ ). Um mantenedor declarado é um desenvolvedor que afirmou ter conhecimento acima de 3 (três) em um arquivo, caso contrário, ele é um não-mantenedor declarado.

Pode-se observar nas Figuras 4 e 5 as distribuições das respostas do *survey* para ambos *datasets* público e privado. E pode-se observar na Tabela 9 a porcentagem de mantenedores e não-mantenedores considerando os dois *datasets*.

	Mantenedores	Não-Mantenedores
<b>Público</b>	54%	46%
<b>Privado</b>	47%	53%

Tabela 9 – Porcentagem de mantenedores e não-mantenedores.

## 3.2 Extraíndo e Analisando Variáveis de Desenvolvimento

Trabalhos anteriores demonstraram a necessidade de melhorar abordagens tradicionais de identificação de mantenedores de código adicionando variáveis como *tamanho de*

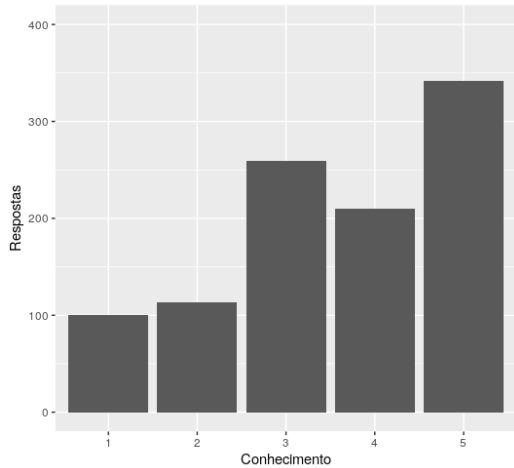


Figura 4 – Distribuição das respostas no *dataset* público.

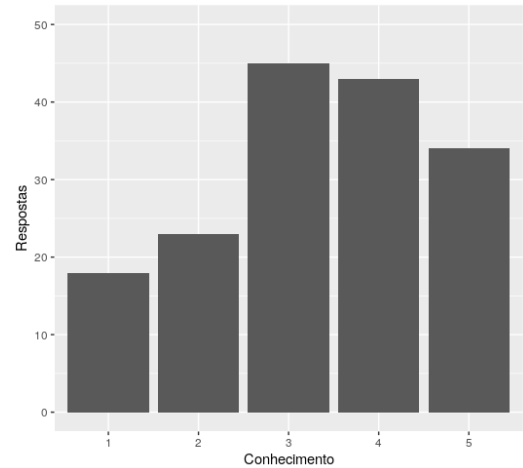


Figura 5 – Distribuição das respostas no *dataset* privado.

*arquivo* e *recência* (AVELINO et al., 2017; KRÜGER et al., 2018). Portanto, esse trabalho analisou a correlação entre diversas variáveis extraídas do histórico de desenvolvimento.

### 3.2.1 Variáveis

Foram selecionadas 12 variáveis que podem ser extraídas de um histórico de desenvolvimento. Essas variáveis e seus significados estão apresentados na Tabela 10. Neste trabalho, essas variáveis foram extraídas para a última versão de cada arquivo.

As variáveis *Adds*, *Dels*, *Mods* e *Conds* são variáveis propostas e utilizadas para inferir a familiaridade de desenvolvedores com arquivos de código fonte nos trabalhos de Lira (2016), Ibiapina et al. (2017), Alves et al. (2018). Essas variáveis foram extraídas usando o comando `git diff`<sup>6</sup>.

O comando `git diff` retorna apenas as linhas adicionadas e removidas entre duas versões de um arquivo. Neste trabalho, uma modificação é classificada como sendo um conjunto de linhas removidas seguidas por um conjunto de linhas adicionadas do mesmo tamanho (LIRA, 2016; IBIAPINA et al., 2017). Foi utilizada a distância de *Levenshtein* (NAVARRO, 2001) para identificar quais pares (*remoção*, *adição*) caracterizavam uma modificação entre duas versões dos arquivos. O algoritmo tem como entradas a linha removida e adicionada, e retorna um valor que representa a quantidade de caracteres que devem ser modificados para transformar a linha removida na linha adicionada. Neste trabalho, foi considerado que houve uma modificação se o valor retornado fosse menor que determinada porcentagem (*threshold*) da linha removida. Foi utilizado um *threshold* de 40%, como sugerido por Canfora, Cerulo e Penta (2007).

Todas as outras variáveis extraídas já foram analisadas por outros estudos. Na

<sup>6</sup> <https://git-scm.com/docs/git-diff>

Variável	Significado
Adds	O número de linhas adicionadas por um desenvolvedor $d$ até a versão $v$ de um arquivo $f$ .
Dels	O número de linhas removidas por um desenvolvedor $d$ até a versão $v$ de um arquivo $f$ .
Mods	O número de linhas modificadas por um desenvolvedor $d$ até a versão $v$ de um arquivo $f$ .
Conds	O número de condicionais adicionadas por um desenvolvedor $d$ até a versão $v$ de um arquivo $f$ .
Amount	A soma do número de linhas removidas e adicionadas por um desenvolvedor $d$ até a versão $v$ de um arquivo $f$ .
FA	Um valor binário que indica se um desenvolvedor $d$ adicionou o arquivo $f$ ao projeto.
Blame	O número de linhas adicionadas por um desenvolvedor $d$ que estão na versão $v$ de um arquivo $f$ .
NumCommits	O número de <i>commits</i> feitos por um desenvolvedor $d$ no arquivo $f$ até a versão $v$ .
NumDays	O número de dias desde o último commit feito por um desenvolvedor $d$ no arquivo $f$ até a versão $v$ .
NumModDevs	O número de <i>commits</i> feitos por outros desenvolvedores no arquivo $f$ desde o último <i>commit</i> de um desenvolvedor $d$ no arquivo $f$ .
Size	Número de linhas de código (LOC) do arquivo $f$ .
AvgDaysCommits	A média do número de dias entre os <i>commits</i> de um desenvolvedor $d$ em um arquivo $f$ até a versão $v$ .

Tabela 10 – Variáveis extraídas dos históricos de desenvolvimento. O significado das variáveis são apresentados considerando um desenvolvedor  $d$  e um arquivo  $f$  na sua versão  $v$ .

Tabela 11 estão trabalhos que analisaram essas variáveis no contexto de conhecimento de desenvolvedores com código fonte. A variável *NumCommits* é a mais explorada em estudos da área.

### 3.2.2 Analisando as Variáveis Extraídas

Primeiramente, foram analisadas as correlações entre as 12 variáveis extraídas do histórico de desenvolvimento com as respostas sobre o conhecimento dos desenvolvedores com arquivos, obtidas pelo *survey*. Daqui em diante, essas respostas obtidas pelo *survey* serão representados por uma variável chamada *Conhecimento*. A condução e análise das correlações foram guiadas pelas seguintes questões de pesquisa:

- **QP1** - Qual das variáveis extraídas do histórico de desenvolvimento possui a maior correlação positiva com *Conhecimento*?
- **QP2** - Qual das variáveis extraídas do histórico de desenvolvimento possui a maior correlação negativa com *Conhecimento*?

Variável	Trabalhos
Adds, Dels	(LIRA, 2016; IBIAPINA et al., 2017; ALVES et al., 2018) (MOCKUS; HERBSLEB, 2002; GIRBA et al., 2005) (HATTORI; LANZA, 2009)
Mods, Conds, Amount	(LIRA, 2016; IBIAPINA et al., 2017; ALVES et al., 2018)
Blame	(GIRBA et al., 2005; RAHMAN; DEVANBU, 2011) (KRÜGER et al., 2018; AVELINO et al., 2018) (HANNEBAUER et al., 2016)
FA	(FRITZ et al., 2014; FRITZ; MURPHY; HILL, 2007)
NumCommits	(IBIAPINA et al., 2017; ALVES et al., 2018) (KRÜGER et al., 2018; AVELINO et al., 2018) (MONTANDON; SILVA; VALENTE, 2019) (OLIVEIRA; VIGGIATO; FIGUEIREDO, 2019) (MOCKUS; HERBSLEB, 2002; SILVA et al., 2015) (MCDONALD; ACKERMAN, 2000) (ALONSO; DEVANBU; GERTZ, 2008; FALCÃO et al., 2020) (CONSTANTINOU; KAPITSAKI, 2016) (KAGDI et al., 2012; KAGDI; POSHYVANYK, 2009) (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019; FRITZ et al., 2014) (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020; LIRA, 2016) (MINTO; MURPHY, 2007; HANNEBAUER et al., 2016)
NumDays	(LIRA, 2016; IBIAPINA et al., 2017; ALVES et al., 2018) (KRÜGER et al., 2018; AVELINO et al., 2018) (SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020)
NumModDevs	(LIRA, 2016; IBIAPINA et al., 2017; ALVES et al., 2018) (FRITZ et al., 2014)
Size	(AVELINO et al., 2018) (OLIVEIRA; VIGGIATO; FIGUEIREDO, 2019)
AvgDaysCommits	(MONTANDON; SILVA; VALENTE, 2019)

Tabela 11 – Variáveis extraídas e trabalhos relacionados.

- **QP3** - Quais os níveis de correlações existentes entre as variáveis extraídas?

Através da correlação com o grau de conhecimento, buscou-se identificar quais variáveis têm maior potencial para fazer parte de um novo modelo de conhecimento a ser proposto. Na Tabela 12 estão listadas as direções e intensidades das correlações das variáveis extraídas com *Conhecimento*, utilizando o *rho* de *Spearman* (SCHÖBER; BOER; SCHWARTZ, 2018).

Apesar de *NumCommits* ser a variável mais utilizada para inferir conhecimento de desenvolvedores em estudos presentes na literatura, ela não apresentou a maior correlação positiva com a variável *Conhecimento*. A variável que apresentou a correlação positiva mais alta com *Conhecimento* foi *FA* (QP1), seguida de perto por *Adds*. Isso sugere que uma medida de mudança de maior granularidade como *Adds* é mais importante que *NumCommits* para compor modelos de conhecimento. A variável que apresentou a

Variável	Direção	Corr. com Conhecimento	p-value
FA	positiva	0,33	0
Adds	positiva	0,31	0
Blame	positiva	0,29	0
Amount	positiva	0,28	0
NumDays	negativa	0,24	0
Size	negativa	0,21	6,313e-13
NumModDevs	negativa	0,21	1,588e-13
AvgDaysCommits	positiva	0,21	4,894e-13
NumCommits	positiva	0,20	1,279e-11
Cond	positiva	0,19	8,221e-11
Dels	positiva	0,02	0,369
Mods	positiva	0,01	0,515

Tabela 12 – Correlação das variáveis extraídas com Conhecimento.

correlação mais baixa com *Conhecimento* foi *Mods*.

Por outro lado, a variável que mostrou a maior correlação negativa com *Conhecimento* foi *NumDays* (**QP2**), seguida por *NumModDevs* e *Size*. Isso reforça a importância da recência no conhecimento que um desenvolvedor tem com um arquivo de código fonte.

Para responder à Questão de Pesquisa 3, foi analisada a correlação entre as variáveis extraídas usando o  $\rho$  de *Spearman* (SCHOBER; BOER; SCHWARTE, 2018). Essas correlações podem ser observadas na Tabela 15 do Apêndice A, e estão representadas na Figura 6. O uso de variáveis com certo nível de correlação pode levar a criação de modelos inacurados (YU; LIU, 2003).

A variável *NumCommits* obteve correlações positivas moderadas com as variáveis: *Adds*, *Dels*, *Mods*, e *Amount* ( $\rho \geq 0,5$ ). Todas essas 5 variáveis podem ser utilizadas com medidas do número de mudanças feitas por um desenvolvedor ao longo da história de um arquivo.

As variáveis *NumModDevs* e *NumDays* também apresentaram uma correlação positiva moderada ( $\rho \geq 0,5$ ) entre si. Isso sugere que *NumModDevs* pode ser vista como uma forma de recência de modificação, sendo que quanto mais tempo passa desde a última modificação, maiores as chances de o arquivo ser modificado por outros desenvolvedores.

### 3.3 Técnicas Comparadas

Essa Seção descreve as técnicas utilizadas neste estudo para identificar mantenedores de código. As técnicas descritas nas Seções 3.3.1, 3.3.2, e 3.3.3 foram utilizadas em um estudo anterior (AVELINO et al., 2018).

As Seções 3.3.4 e 3.3.5, descrevem dois modelos lineares de conhecimento, um

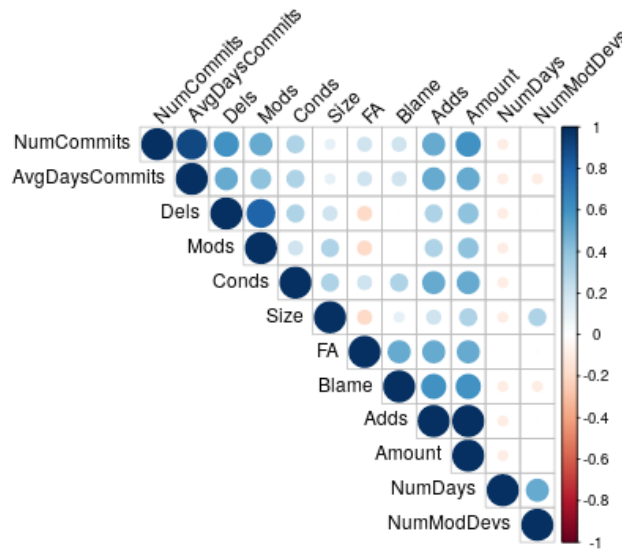


Figura 6 – Correlações entre as variáveis extraídas do histórico de desenvolvimento.

apresentado por [Lira \(2016\)](#), e outro proposto nesta dissertação. A construção do modelo descrita na Seção 3.3.5 é guiada pela análise de correlações feita nas seções anteriores. Finalmente, em 3.3.6 é proposto a utilização de classificadores de aprendizagem de máquina para a identificação de mantenedores.

### 3.3.1 Commits

Essa técnica usa o número de *commits* como medida de conhecimento que um desenvolvedor possui sobre um arquivo de código fonte. Quanto mais *commits* um desenvolvedor fez em um arquivo, maior o seu conhecimento sobre o conteúdo daquele arquivo.

Essa técnica é amplamente utilizada no contexto de expertise de desenvolvedores, como pode-se observar na Tabela 11. Essa medida é utilizada tanto de forma atômica ([AVELINO et al., 2018](#); [SILVA et al., 2015](#); [HANNEBAUER et al., 2016](#)), como na composição de outras técnicas ([FRITZ et al., 2014](#); [KAGDI et al., 2012](#)). Neste trabalho, utilizou-se apenas o número de *commits* (forma atômica) como técnica de identificação de mantenedores. Neste estudo, essa técnica é referenciada como *NumCommits*.

### 3.3.2 Blame

Essa técnica infere o conhecimento que um desenvolvedor tem em um arquivo com o número de linhas adicionados por ele presentes na última versão do arquivo. Ferramentas de

Blame (*Blame-like tools*), como o comando `git-blame`<sup>7</sup>, são usadas para identificar autores de cada linha de um arquivo. Como em outros trabalhos (GIRBA et al., 2005; RAHMAN; DEVANBU, 2011; HANNEBAUER et al., 2016; AVELINO et al., 2018; KRÜGER et al., 2018), utilizou-se a porcentagem de linhas associadas a um desenvolvedor como medida de conhecimento.

### 3.3.3 Degree of Authorship (DOA)

Fritz et al. (2014) propõe que o conhecimento que um desenvolvedor possui com um arquivo de código fonte depende de fatores como: autoria do arquivo, número de contribuições, número de mudanças feitas por outros desenvolvedores. Eles combinaram essas variáveis em um modelo linear chamado *Degree of Authorship* (DOA). Em um processo similar ao realizado nesta dissertação, os autores coletaram dados sobre o conhecimento de desenvolvedores em elementos de código e juntamente com dados sobre o histórico de desenvolvimento, aplicaram regressão linear múltipla (TRANMER; ELLIOT, 2008) para encontrar pesos associados a cada variável do modelo definido. O valor DOA que um desenvolvedor  $d$  tem com a versão  $v$  de um arquivo  $f$  é calculado da seguinte forma:

$$\text{DOA}(\mathbf{d}, \mathbf{f}(\mathbf{v})) = 3,293 + 1,098 * FA + 0,164 * DL - 0,321 * \ln(1 + AC) \quad (3.6)$$

onde,

- $FA$ : 1 se o desenvolvedor  $d$  é o criador do arquivo  $f$ , 0 caso contrário.
- $DL$ : é o número de mudanças feitas por um desenvolvedor  $d$  até a versão  $v$  de um arquivo  $f$ .
- $AC$ : é o número de mudanças feitas por outros desenvolvedores no arquivo  $f$ .

### 3.3.4 CoDiVision

Um modelo linear para a inferência do conhecimento de um desenvolvedor com um arquivo de código fonte foi proposto por Lira (2016), e publicada em trabalhos posteriores (IBIAPINA et al., 2017; ALVES et al., 2018). Nestes trabalhos foi utilizado o termo familiaridade para se medir o conhecimento de um desenvolvedor a partir de alguns indicadores. Como os autores não deram nenhum nome específico ao modelo, neste trabalho ele será chamado de **CoDiVision**. CoDiVision é o nome do sistema web<sup>8</sup> que permite a visualização das familiaridades de desenvolvedores com arquivos e módulos de um projeto, e utiliza este modelo como núcleo de várias funcionalidades.

<sup>7</sup> <https://git-scm.com/docs/git-blame>

<sup>8</sup> <http://easii.ufpi.br/codivision/>



O modelo CoDiVision pode ser resumido nas fórmulas apresentadas a seguir. O modelo é apresentado levando em consideração um desenvolvedor  $d$  e um arquivo de código fonte  $f$  na sua versão  $v$ .

Esse modelo leva em consideração o montante de alterações realizadas por um desenvolvedor na história do arquivo. Esse montante é dado por:

$$\begin{aligned} \mathbf{W}(\mathbf{d}, \mathbf{f}(\mathbf{v})) = & (\mathbf{Adds}^{d,f(v)} * W_{Adds}) + (\mathbf{Mods}^{d,f(v)} * W_{Mods}) \\ & + (\mathbf{Dels}^{d,f(v)} * W_{Dels}) + (\mathbf{Conds}^{d,f(v)} * W_{Conds}) \end{aligned} \quad (3.7)$$

onde,

- $W(d, f(v))$ : total de mudanças feitas por um desenvolvedor  $d$  até a versão  $v$  de um arquivo  $f$ .
- $W_{Adds}$ : peso associado (coeficiente) ao número de adições.
- $W_{Mods}$ : peso associado (coeficiente) ao número de modificações.
- $W_{Dels}$ : peso associado (coeficiente) ao número de remoções.
- $W_{Conds}$ : peso associado (coeficiente) ao número de condicionais adicionados.

Esse modelo leva em consideração a degradação do conhecimento de um desenvolvedor pelo tempo decorrido desde a última modificação. A degradação é dada por:

$$\mathbf{T}(\mathbf{d}, \mathbf{f}(\mathbf{v})) = 1 - (\mathbf{NumDays} * P_t) \quad (3.8)$$

onde,

- $T(d, f(v))$ : o valor que representa a porcentagem que será removida do conhecimento que um desenvolvedor  $d$  tem com um arquivo  $f$ .
- $P_t$ : o peso associado (coeficiente) ao número de dias desde a última modificação de um desenvolvedor  $d$  em um arquivo  $f$ .

O modelo também leva em consideração a degradação de conhecimento por mudanças realizadas por outros desenvolvedores desde a última modificação. Essa degradação é dada por:

$$\mathbf{N}(\mathbf{d}, \mathbf{f}(\mathbf{v})) = 1 - (\mathbf{NumModDevs} * P_n) \quad (3.9)$$

onde,

- $N(d, f(v))$ : valor que representa a porcentagem que será removida do conhecimento de um desenvolvedor  $d$  tem com um arquivo  $f$ .
- $P_n$ : é o peso associado (coeficiente) ao número de modificações realizadas por outros desenvolvedores desde a última alteração de um desenvolvedor  $d$  em um arquivo  $f$ .

Finalmente, a familiaridade do desenvolvedor  $d$  na versão  $v$  de um arquivo  $f$  é dado por:

$$\text{Familiaridade}(d, f(v)) = W(d, f(v)) * T(d, f(v)) * N(d, f(v)) \quad (3.10)$$

Em resumo, essa fórmula representa o conhecimento de um desenvolvedor de acordo com a quantidade de alterações que ele realizou em um arquivo ( $W(d, f(v))$ ), tendo uma porcentagem removida de acordo com o tempo decorrido ( $T(d, f(v))$ ) e quantidade de alterações feitas por outros ( $N(d, f(v))$ ).

Porém, os autores não propuseram coeficientes para este modelo nos estudos realizados (LIRA, 2016; IBIAPINA et al., 2017; ALVES et al., 2018). Neste trabalho, durante a avaliação da performance deste modelo em identificar mantenedores de arquivos, é utilizada regressão linear múltipla para achar os melhores coeficientes que ajustam os dados disponíveis.

### 3.3.5 Nível de Expertise

Nesta Seção é proposto um modelo linear utilizando as variáveis extraídas e analisadas nas Seções 3.2.1 e 3.2.2. A seguir é descrito o processo de escolha das variáveis que compuseram o modelo.

Avelino et al. (2018) demonstrou como as variáveis *tamanho de arquivo* e *recência* influenciam no conhecimento de um desenvolvedor com um arquivo de código fonte. Dessa forma, modelos que levem em conta essas informações podem ser mais acurados na identificação de mantenedores de código. Isso é reforçado com os resultados apresentados na Tabela 12, sendo essas as variáveis com maiores correlações negativas com *Conhecimento*. Portanto, as variáveis *Size* e *NumDays* foram incluídas no modelo proposto.

Continuando, entre as variáveis *Adds*, *Amount*, e *Blame*, foi escolhida a variável *Adds*, por ser a que apresentou a maior correlação positiva com *Conhecimento* entre as três. Não foi escolhida mais de uma das três porque essas variáveis são conceitualmente relacionadas, com  $Blame \subseteq Adds \subseteq Amount$ , e possuem correlações moderadas a fortes

entre si (Tabela 15). Também não foi adicionada a variável *NumCommits*. Essa variável é apenas uma forma macro de contabilizar as alterações feitas por um desenvolvedor em um arquivo de código fonte, e já foi escolhida a variável *Adds* que possui uma maior correlação positiva com *Conhecimento*. Além disso, a variável *Adds* possui uma correlação moderada com *NumCommits* ( $\rho \geq 0,5$ , Figure 6). Consequentemente, a variável *AvgDaysCommits* também foi descartada do modelo. Essa distinção entre Heurística Baseada em Commits (*Commit-Based Heuristic*) como a variável *NumCommits*, e Heurística Baseada em LOC (*LOC-Based Heuristic*) como a variável *Adds* é suportada por outros estudos (YAMASHITA et al., 2015; OLIVEIRA et al., 2020).

Também foi incluída a variável *FA* no novo modelo. Essa variável apresentou a maior correlação positiva com *Conhecimento* entre todas as variáveis extraídas. A variável *NumModDevs* não foi incluída devida a correlação moderada ( $\rho \geq 0,5$ ) que apresentou com a variável *NumDays*, que possui a maior correlação negativa com *Conhecimento*. Em conclusão, as variáveis escolhidas para compor o modelo de conhecimento neste trabalho foram: *Adds*, *FA*, *Size*, e *NumDays*.

Na definição do modelo linear, foi utilizada transformação *log* nas variáveis *Adds*, *NumDays* e *Size*, para lidar com a diferença nas escalas. O **Conhecimento** de um desenvolvedor *d* na versão *v* de um arquivo *f* é dado pelo modelo:

$$\begin{aligned} \text{Conhecimento}(\mathbf{d}, \mathbf{f}(\mathbf{v})) = & C + b_0 * \ln(1 + \mathbf{Adds}^{d,f(v)}) + b_1 * (\mathbf{FA}^f) \\ & - b_2 * \ln(1 + \mathbf{NumDays}^{d,f(v)}) - b_3 * \ln(\mathbf{Size}^{f(v)}) \end{aligned} \quad (3.11)$$

Onde  $C, b_0, b_1, b_2, b_3$  é o *intercept* e coeficientes que melhor ajustam determinada base de dados por meio de uma técnica como regressão linear múltipla, sendo *intercept* o valor retornado pelo modelo quando os valores das variáveis que o compõe são zero.

No resto deste trabalho, este modelo será chamado de *Nível de Expertise*. Por questões de praticidade, na apresentação dos resultados ele será referenciado pela sigla **NE**.

### 3.3.6 Classificadores de Aprendizagem de Máquina

Além das técnicas de regressão linear, neste trabalho é investigado o uso de algoritmo de aprendizagem de máquina para classificar desenvolvedores em mantenedores e não-mantenedores. Essa proposta se resume a uma classificação binária: não-mantenedores (Conhecimento 1-3) e mantenedores (Conhecimento 4-5). Como os *datasets* são todos balanceados na proporção das duas classes (Seção 3.1.6), não houve necessidade de usar técnicas para lidar com isso.

Foram utilizadas as mesmas *features* (variáveis) selecionadas no processo descrito na Seção 3.3.5. Dessa forma, podemos analisar a performance de modelos lineares e não-lineares que utilizam dessas variáveis para identificar mantenedores de arquivo. Foram avaliados cinco conhecidos classificadores de aprendizagem de máquina: Random Forest (LIAW; WIENER et al., 2002), Support-Vector Machines (SVM) (WESTON; WATKINS, 1998), K-Nearest Neighbor (KNN) (PETERSON, 2009), Gradient Boosting Machine (FRIEDMAN, 2001), Logistic Regression (JR; LEMESHOW; STURDIVANT, 2013).

## 3.4 Metodologia de Avaliação

### 3.4.1 Utilizando Técnicas Lineares para Identificar Mantenedores

Neste trabalho foram avaliadas as performances das técnicas lineares *NumCommits*, *Blame*, *DOA*, *Nível de Expertise* e *CoDiVision*, apresentados na Seção 3.3, na identificação de mantenedores de arquivo. Para alcançar esse objetivo, foi utilizada uma avaliação similar a feita em um trabalho relacionado (AVELINO et al., 2018).

Os valores de cada técnica foram normalizados para cada par (desenvolvedor, arquivo) dos dois *datasets* públicos e privados. Nesta normalização, define-se a *expertise* de um desenvolvedor  $d$  com um arquivo  $f$  ( $expertise(d, f)$ ) como 1 para o desenvolvedor com maior valor para uma dada técnica, caso contrário ele recebe um valor proporcional. Por exemplo, para um determinado arquivo  $f$  os desenvolvedores  $d1$ ,  $d2$ , e  $d3$  têm valores *Blame* de 10, 15 e 20 respectivamente. Seus valores normalizados de *Blame* para o arquivo  $f$  são  $expertise(d1, f) = 10/20 = 0,5$ ,  $expertise(d2, f) = 15/20 = 0,75$ , e  $expertise(d3, f) = 1$ . O mesmo processo de normalização é feito para todas as outras técnicas lineares.

Feita a normalização, um desenvolvedor  $d$  é classificado como um mantenedor de um arquivo  $f$  se a sua  $expertise(d, f)$  é maior que um *threshold*  $k$ , caso contrário ele é considerado um não-mantenedor. Seguindo o exemplo dado no parágrafo anterior, considerando um *threshold*  $k = 0,7$ , os desenvolvedores  $d2$  e  $d3$  seriam considerados mantenedores do arquivo  $f$  de acordo com a técnica *Blame*. Dessa forma, pode-se classificar cada desenvolvedor de cada par (*desenvolvedor*, *arquivo*) dos dois *datasets* utilizados neste estudo, em mantenedor ou não mantenedor, de acordo com determinada técnica linear, considerando determinado *threshold* de classificação.

### 3.4.2 Avaliando a Performance

Buscou-se comparar a performance de todas as técnicas nas suas melhores configurações. Para as técnicas lineares, isso requer achar um *threshold* de classificação  $k$  para cada técnica linear que maximize a identificação correta de mantenedores de arquivo.

Nas avaliações das técnicas lineares, a performance de cada técnica foi verificada utilizando 11 *thresholds* diferentes. O *threshold*  $k$  foi variado de 0 a 1, usando passos de 0,1. A cada passo, foi computado o *F-Measure* de cada técnica usando as seguintes equações:

$$Precision(k) = \begin{cases} \frac{|(d,f) \in O_m \mid expertise(d, f) > k|}{|expertise(d, f) > k|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid expertise(d, f) \geq k|}{|expertise(d, f) \geq k|}, & \text{otherwise} \end{cases} \quad (3.12)$$

$$Recall(k) = \begin{cases} \frac{|(d,f) \in O_m \mid expertise(d, f) > k|}{|O_m|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid expertise(d, f) \geq k|}{|O_m|}, & \text{otherwise} \end{cases} \quad (3.13)$$

$$F - Measure(k) = 2 * \frac{Precision(k) * Recall(k)}{Precision(k) + Recall(k)} \quad (3.14)$$

onde,

$O_m$  = conjunto de mantenedores declarados (*Conhecimento* > 3) inferidos das respostas do *survey* (Seção 3.1.6).

Ao final deste processo é obtido a melhor performance de cada técnica juntamente com o *threshold*  $k$  associado. Para os dois modelos de regressão linear apresentados nas Seções 3.3.4 e 3.3.5, foi executado um *10-fold cross validation* em cada *threshold*  $k$  para verificar sua performance e achar a melhor.

Também foi utilizado o *10-fold cross validation* para avaliar a performance dos classificadores de aprendizagem de máquina (3.3.6). Para esses classificadores, o *F-Measure* foi utilizado como métrica de performance, porém sem o processo de achar melhor *threshold*  $k$ , como as técnicas lineares. Para ajustar os *hyper-parametros* e encontrar as melhores configurações para cada classificador, foi aplicada uma busca em grade (CLAESSEN; MOOR, 2015).



Parte IV

Avaliação





## 4 Resultados e Discussão

Este capítulo descreve a avaliação realizada. A análise foi realizada em 2 (dois) cenários: com os dados públicos e com dados privados. Em cada um desses cenários, foi utilizado o processo descrito na Seção 3.4 para achar a melhor performance das técnicas lineares. Para avaliar a performance dos classificadores de aprendizagem de máquina, foi utilizado o *10-fold cross validation*, como explicado na Seção 3.4.

### 4.1 Melhores Configurações das Técnicas Lineares

Como explicado na Seção 3.4 foi variado um *threshold*  $k$  de 0 a 1, buscando um  $k$  que maximize o *F-Measure* de cada técnica linear na identificação de mantenedores. Pode-se observar nas Figuras 7 e 8 o *F-Measure* das técnicas lineares em cada *threshold* analisado, utilizando dados públicos e privados respectivamente. As performances obtidas utilizando esses *thresholds* foram utilizadas como medidas finais de performances para essas técnicas.

As melhores configurações encontradas para *Blame*, *NumCommits* e *CoDiVision* foram adotando baixos *thresholds*. *Blame* e *CoDiVision* obtiveram suas melhores performances adotando o menor *threshold* possível de  $k = 0$  utilizando dados públicos e privados. De maneira similar, *NumCommits* alcançou sua melhor performance com *thresholds* de  $k = 0,1$  e  $k = 0,3$  utilizando dados públicos e privados respectivamente. Esses baixos *thresholds* indicam que essas técnicas performam melhor requerendo valores menores para considerar desenvolvedores como mantenedores de um arquivo.

Por outro lado, *DOA* tem a sua melhor configuração com  $k = 0,1$  com dados públicos, e  $k = 0,7$  com dados privados. Finalizando, *NE* alcançou os melhores resultados com altos *thresholds* de  $k = 0,7$  com dados públicos, e  $k = 0,8$  com dados privados. Pode-se concluir que a técnica proposta é mais conservativa na identificação de mantenedores, requerendo valores mais altos para classificar um desenvolvedor como mantenedor de um arquivo.

A análise e discussão dos resultados apresentados a seguir tiveram o propósito de responder às seguintes questões de pesquisa:

- **QP4** - Qual técnica linear apresenta a melhor performance na identificação correta de mantenedores de arquivos?
- **QP5** - Qual classificador de aprendizagem de máquina apresenta a melhor performance na identificação correta de mantenedores de arquivos?

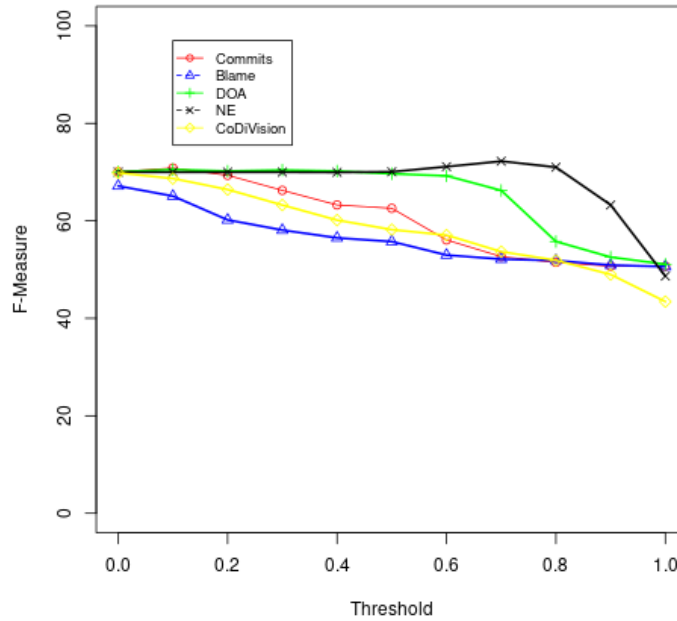


Figura 7 – Performance em cada *threshold* analisado usando dados dos projetos públicos.

- **QP6** - As técnicas propostas neste trabalho trouxeram melhorias na identificação de mantenedores de arquivo?

## 4.2 Resultados

Na Tabela 13 estão apresentadas as performances das técnicas lineares na classificação de mantenedores de arquivos. Essas performances foram obtidas considerando os *thresholds* mencionados na Seção 4.1 em cada um dos 2 (dois) cenários estudados.

Utilizando apenas os dados públicos (Tabela 13 - **Público**), a técnica *NE* apresentou o melhor *F-Measure* de 72% , seguida de perto por *DOA* e *NumCommits*, ambos com 70%. A técnica com menor *F-Measure* foi *Blame* com 67%. Analisando o *Recall*, a técnica que alcançou o mais alto valor foi *CoDiVision* com 99% no cenário utilizando os dados públicos. Já a técnica que apresentou o menor *Recall* nesse mesmo cenário foi *Blame*, com 83%. Finalizando, as técnicas que obtiveram os maiores *Precision* foi *NE* e *NumCommits* com 60%, e *CoDiVision* obtendo o pior *Precision* de 54%.

Utilizando apenas os dados privados (Tabela 13 - **Privado**), *DOA* atingiu o valor mais alto para *F-Measure* com 68%, seguido de perto por *NumCommits* com 67% e *NE* com 64%. Como no outro cenário, a técnica *Blame* obteve o menor *F-Measure* de 55%. O melhor *Recall* foi da técnica *CoDiVision* com 97%, com *Blame* obtendo o menor, 62%. Olhando para o *Precision*, *DOA* alcançou o melhor com 61%, e *CoDiVision* com o menor de 47%.

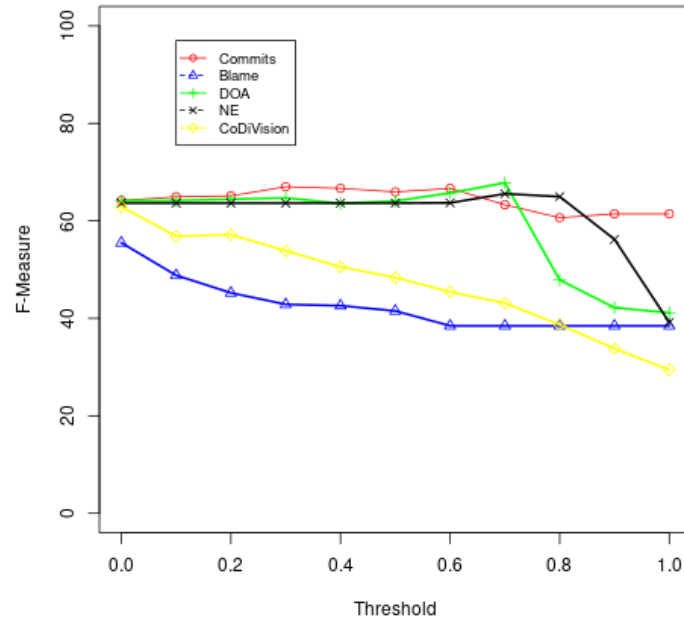


Figura 8 – Performance em cada *threshold* analisado usando dados dos projetos privados.

Na Tabela 14 estão apresentados os resultados da aplicação dos classificadores de aprendizagem de máquina mencionados na Seção 3.3.6. Como foi feito para as técnicas lineares, os resultados mostram a performance desses classificadores nos mesmos 2 (dois) cenários analisados.

Analisando o cenário utilizando os dados públicos (Tabela 14 - **Público**), *Random Forest*, *SVM* e *GBM* apresentaram os melhores *F-Measures* de 73%, com os outros dois classificadores alcançando valores próximos. Ainda nesse cenário, os classificadores que obtiveram o maior *Recall* foram *SVM* e *Logistic Regression* com 75%, e *KNN* obteve o menor com 71%. Analisando o *Precision*, *GBM* e *Random Forest* alcançaram os maiores valores com 73%. O pior *Precision* foi de *Logistic Regression* com 69%.

Na análise utilizando apenas os dados privados (Tabela 14 - **Privado**), o classificador que obteve o mais alto *F-Measure* foi *KNN* com 67%, e *Logistic Regression* apresentou o pior com 58%. Neste cenário, o melhor *Recall* também foi do *KNN* com 69%, e *Logistic Regression* apresentou o pior com 51%. Finalmente, os melhores *Precision* foram alcançados pelos classificadores *Logistic Regression* e *KNN*, ambos com 70%. *Random Forest* apresentou o *Precision* mais baixo de 59%.

## 4.3 Discussão

Analisando os resultados das técnicas lineares, o modelo proposto neste trabalho não trouxe uma melhora significativa na identificação de mantenedores de arquivo. Mesmo

	Público			Privado		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
<b>NE</b>	0.60	0.91	0.72	0.50	0.95	0.64
<b>CoDiVision</b>	0.54	0.99	0.69	0.47	0.97	0.63
<b>DOA</b>	0.55	0.97	0.70	0.61	0.77	0.68
<b>NumCommits</b>	0.60	0.87	0.70	0.55	0.86	0.67
<b>Blame</b>	0.56	0.83	0.67	0.5	0.62	0.55

Tabela 13 – Performance das técnicas lineares nos seus melhores *thresholds k*.

	Público			Privado		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
<b>Random Forest</b>	0.73	0.74	0.73	0.59	0.62	0.59
<b>SVM</b>	0.71	0.75	0.73	0.63	0.60	0.61
<b>KNN</b>	0.71	0.71	0.71	0.70	0.69	0.67
<b>GBM</b>	0.73	0.73	0.73	0.65	0.58	0.60
<b>Logistic Regression</b>	0.69	0.75	0.72	0.70	0.51	0.58

Tabela 14 – Performance dos classificadores de aprendizagem de máquina.

alcançando o maior *F-Measure* no cenário com maior volume de dados, o modelo proposto obteve um *Precision* máximo de 60% (Tabela 13 - Público).

A técnica apresentada por Lira (2016), Ibiapina et al. (2017) também não obteve um desempenho satisfatório, com um *F-Measure* máximo de 69% associado a um *Precision* de 54%. Isso se deve ao fato do modelo utilizar as variáveis poucos correlacionadas com *Conhecimento: Mods, Dels e Conds*, e com altas correlações entre si, como pode ser observado na Tabela 12 e Figura 6 da Seção 3.2.2. Isso acaba diminuindo o poder preditivo do modelo (YU; LIU, 2003).

Esse resultado de baixo *Precision* e alto *Recall* foi obtido por todas as técnicas lineares nos dois cenários. Resultados similares também foram obtidos com técnicas lineares em um estudo de propósito similar. Anvik e Murphy (2007) utilizaram informações sobre quem modificou recentemente um arquivo (Seção 2.1) para identificar *experts* em arquivos relacionados a relatórios de *bugs* (*bug reports*). Eles conseguiram um *Precision* máximo de 59% associado com um *Recall* de 71%. Esses resultados mostram que, de maneira geral, essas técnicas lineares tendem a uma classificação positiva de desenvolvedores como mantenedores, mesmo contendo uma quantidade significativa de falsos positivos nestas classificações.

Para responder a QP4, é preciso levar em consideração os dois cenários analisados. No cenário com a maior quantidade de dados, levando em consideração o *F-Measure* como resumo final de performance, a técnica mais eficiente em identificar mantenedores de arquivo é *Nível de Expertise*, significando que ela conseguiu balancear melhor a capacidade de identificação (*Recall*) com identificações precisas (*Precision*). Utilizando os dados privados, a técnica mais eficiente é *DOA*, utilizando a mesma medida de performance.

Porém, mesmo considerando que a natureza dos dados privados e públicos sejam distintas, também deve-se levar em consideração a quantidade de dados utilizada em cada cenário. São 1024 instâncias no *dataset* público, e apenas 163 no *dataset* privado. Desta forma, os resultados podem ser influenciados pela quantidade, principalmente os das técnicas lineares que necessitam de treinamento para ajustar coeficientes de variáveis, como *CoDiVision* e *Nível de Expertise*. Tais técnicas tendem a melhorar de performance à medida que aumenta-se o tamanho do conjunto de treinamento, até determinado ponto de estabilidade, comportamento chamado de Curva de Aprendizado (*Learning Curve*) (MEEK; THIESSON; HECKERMAN, 2002). Dessa forma, pode-se argumentar que os resultados utilizando o *dataset* público são mais representativos das performances das técnicas.

A resposta para qual o classificador de aprendizagem de máquina obteve o melhor desempenho (QP5) também depende do cenário analisado. No cenário que utiliza a maior quantidade de dados, os melhores resultados foram de *Random Forest*, *SVM* e *Gradient Boosting Machines*, atingindo os melhores *F-Measures*. Já no cenário utilizando o *dataset*

privado, o algoritmo *K Nearest Neighbors* obteve o melhor desempenho. Esses resultados mostram que esses três classificadores são os mais apropriados entre os estudados para serem utilizados em implementações de ferramentas, e em pesquisas futuras.

O melhor *F-Measures* de ambos cenários entre todas as técnicas foi de 73% alcançado pelos classificadores *Random Forest* e *GBM*. Mesmo havendo técnicas lineares que obtiveram *F-Measure* próximos a isso, os classificadores de aprendizagem de máquina apresentaram um melhor balanceamento entre *Precision* e *Recall*. Comparados com as técnicas lineares, os classificadores alcançaram os melhores *Precisions* em ambos cenários analisados. A técnica linear que mais se aproximou aos classificadores em termos de *Precision* foi *Blame* com 69%, porém com um *Recall* de apenas 62%.

Com esses resultados, podemos responder a **QP6**. Mesmo não apresentando melhoras significativas, tanto o modelo *Nível de Expertise* quanto os classificadores de aprendizagem de máquina alcançaram os melhores *F-Measures* nos cenários analisados. Pode-se destacar os classificadores de aprendizagem de máquina, que tiveram os melhores *Precisions* entre todas as técnicas comparadas, em ambos cenários analisados.

Como as técnicas lineares e os classificadores de aprendizagem de máquina alcançaram *F-Measures* similares, a definição da melhor técnica depende tolerância do usuário a falsos positivos e falsos negativos. Se a aplicação for em contextos que normalmente requerem achar poucos, ou apenas 1 (um) *expert*, como, por exemplo, realizar um operação de *merge* (COSTA et al., 2016b), instruir um novo membro da equipe (KAGDI; POSHYVANYK, 2009), os classificadores de aprendizagem são mais recomendados, pois são mais precisos ao identificar mantenedores. Porém, se a aplicação requer achar um número maior de mantenedores de um arquivo, em aplicações que normalmente requerem mais desenvolvedores como, por exemplo, implementação de nova funcionalidade (HOSSEN; KAGDI; POSHYVANYK, 2014), ou resolução de *bugs* (ANVIK; HIEW; MURPHY, 2006), tolerando certo nível de falsos positivos, as técnicas lineares são mais apropriadas.

## 4.4 Ameaças à Validade

Existem ameaças à validade dos resultados obtidos neste estudo. Essa Seção descreve essas ameaças baseando-se em 4 categorias: Externa, Interna, Construção, e Conclusão (WOHLIN et al., 2012).

### 4.4.1 Validade de Construção

Alguns dos desenvolvedores que participaram do *survey* podem ter inflado sua auto avaliação de conhecimento, devido a algum receio do uso comercial dessas informações. Visando mitigar esse problema, foi explicitado no *survey* que todas as informações seriam utilizados apenas com propósitos acadêmicos.

Com relação ao risco de má interpretação da escala de conhecimento (1 a 5) utilizada no *survey*, foi especificado que: (1) significa que o desenvolvedor não tinha conhecimento sobre o código fonte do arquivo; (3) que o desenvolvedor precisava realizar algumas investigações antes de reproduzir o código; e (5) que o desenvolvedor era um *expert* naquele código. Essa explicação buscou fornecer uma espécie de guia do significado da escala utilizada. Porém, a definição de *expert* tem um caráter subjetivo, o que permanece como uma ameaça a construção.

Outra ameaça a construção foi a definição de mantenedor de acordo com as respostas do *survey*. Foram considerados como mantenedores de arquivos desenvolvedores com conhecimento acima de 3 (três). Essa classificação não foi feita baseada em um estudo empírico. Porém, acredita-se que essa é uma divisão binária apropriada para a escala de conhecimento utilizada.

#### 4.4.2 Validade Interna

Há outros fatores que podem influenciar o conhecimento que um desenvolvedor tem com um arquivo de código fonte, que não foram levados em consideração neste trabalho. Conhecimento pode ser adquirido em atividades comuns que não necessariamente envolvem *commits* por parte dos desenvolvedores como testes e revisão de código (THONGTANUNAM et al., 2016; RIGBY; BIRD, 2013). No entanto, este trabalho focou na identificação de mantenedores usando apenas informações de autoria presentes em sistemas de controle de versão. Sistemas de controle de versão são ferramentas amplamente utilizadas no desenvolvimento de software, fazendo dos modelos propostos aplicáveis em quase todos os projetos desenvolvidos atualmente.

Ainda assim, há outras variáveis que podem ser extraídas de informações contidas em repositórios de código, que podem ser importantes no processo de identificação de mantenedores de um arquivo. Entre essas variáveis, podem ser mencionadas a interação com arquivos do mesmo módulo, e a complexidade do código de um arquivo. Porém, as variáveis mencionadas são mais complexas, mais computacionalmente custosas, e dependentes da linguagem e padrão de projeto utilizados. Por essa razão, este trabalho focou no uso de variáveis simples que são menos dependentes de particularidades do projeto em análise.

#### 4.4.3 Validade de Conclusão

Com relação a testes estatísticos utilizados, na análise de correlação entre as variáveis extraídas foi utilizado o  $\rho$  de *Spearman*. Esse coeficiente de correlação foi escolhido por não requerer uma distribuição normal ou relação linear entre as variáveis. Não há consenso na interpretação da força da correlação encontrada por esse teste de correlação. Porém, nós utilizamos algumas interpretações que podem ser encontradas

em trabalhos de diferentes áreas de estudo ([SCHOBER; BOER; SCHWARTE, 2018](#); [OVERHOLSER; SOWINSKI, 2008](#)).

#### 4.4.4 Validade Externa

Alguns aspectos do estudo realizado buscaram maximizar a capacidade de generalização dos resultados. Para os sistemas *open-source*, foram escolhidas 6 linguagens de programação populares do Github. Não se limitando a apenas uma linguagem, esse trabalho buscou reduzir o impacto que certos aspectos possam ter na avaliação de familiaridade dos desenvolvedores, e consequentemente nos resultados obtidos.

No total, foram utilizados dados de 113 sistemas. Buscamos maximizar o número de sistemas analisados baseando-se em critérios de relevância definidos em um estudo anterior ([AVELINO et al., 2016](#)).

Foram utilizados dados de sistemas *open-source* e privados, buscando trazer quais diferenças existentes para os modelos propostos. Porém, a quantidade de dados de projetos privados utilizados continua como uma ameaça a validade externa.



## 5 Conclusões e Trabalhos Futuros

Finalizando, este último capítulo apresenta o que foi alcançado na pesquisa, suas limitações, e os planos para trabalhos futuros. A Seção 5.1 sumariza o trabalho realizado, descrevendo-o em linhas gerais. A Seção 5.2 lista os principais achados identificados. A Seção 5.3 discute algumas limitações da pesquisa. E, finalmente, a Seção 5.4 apresenta planos para trabalhos futuros.

### 5.1 Resumo

Este trabalho propõe novas técnicas para a identificação de mantenedores de arquivo e as compara com técnicas presentes na literatura da área. Foram selecionados 113 (cento e treze) projetos de 6 (seis) linguagens de programação para realização de um estudo sobre as técnicas aqui desenvolvidas e técnicas existentes na literatura.

Para cada projeto selecionado, foi aplicado um *survey* a uma amostra dos desenvolvedores, coletando informações sobre o conhecimento deles em certos arquivos do projeto. Utilizando-se da base de dados formado pelas respostas do *survey*, combinada com informações sobre os históricos de desenvolvimento dos projetos, foi realizado um estudo de correlação, buscando variáveis que pudessem ter grande influência com o conhecimento dos desenvolvedores. A partir desse estudo foi proposto um modelo linear que infere o conhecimento de desenvolvedores com arquivos.

Por fim, o desempenho desse modelo na identificação mantenedores foi comparada com o de outras quatro técnicas presentes na literatura e com classificadores de aprendizagem de máquina. Para essa análise de desempenho, foi utilizado um oráculo construído a partir de respostas dadas pelos desenvolvedores dos sistemas selecionados.

### 5.2 Principais Achados

Podemos dividir os principais achados deste trabalho em dois conjuntos: aqueles que dizem respeito aos relacionamentos entre as variáveis extraídas do histórico de desenvolvimento e o conhecimento de desenvolvedores em arquivos, e os relacionados à comparação de performance feita entre as técnicas analisadas.

Sobre a correlação entre as variáveis estudadas, pode-se destacar que:

- A variável que representa a *primeira autoria* (*FA*) demonstrou a maior correlação positiva com o conhecimento de desenvolvedores com arquivos.

- Das variáveis que representam as mudanças feitas por um desenvolvedor ao longo da história de um arquivo, a variável *Número de linhas adicionadas* (*Adds*) apresentou a maior correlação positiva com conhecimento, maior que *Número de Commits* (*NumCommits*), amplamente utilizada em estudos da área.
- A variável que representa a recência da modificação em um arquivo (*NumDays*) apresentou a maior correlação negativa com conhecimento.

Com relação à performance na identificação de mantenedores das técnicas comparadas, pode-se destacar que:

- O modelo linear proposto não trouxe uma melhora significativa de performance, comparado com as outras técnicas existentes. Ele não alcançou o maior *Precision*, nem o melhor *Recall*. Porém, o modelo apresentou o melhor balanceamento entre essas duas medidas, consequentemente alcançando o melhor *F-Measure* entre as técnicas lineares analisadas.
- Os classificadores de aprendizagem de máquina tiveram os melhores *Precisions* nos cenários analisados. Eles também apresentaram os melhores balanceamentos entre *Precision* e *Recall*, alcançando os melhores *F-Measures* entre todas as técnicas analisadas.

### 5.3 Desafios e Limitações

Este trabalho apresenta limitações e desafios à investigações futuras. A seguir são apresentadas algumas limitações identificadas.

Uma das limitações deste estudo diz respeito a aplicabilidade dos modelos propostos. Não foi feita um estudo prático de como esses modelos de identificação de mantenedores de arquivos podem ser utilizados, e qual sua eficácia em determinados contextos do desenvolvimento de software. Entre tais contextos pode-se citar a identificação de mantenedores dentro de uma abordagem para implementação de novas funcionalidades em um software. Uma alternativa seria o uso dentro de uma abordagem de resolução de *bugs*. Também não foi analisado como a inferência de conhecimento pode ser realizada em outros níveis de granularidade, para identificar mantenedores de pacotes, subsistemas e até mesmo funcionalidades.

Outra limitação deste trabalho, ligada diretamente à aplicabilidade dos modelos, são as suas implementações em ferramentas. Esse estudo teve o objetivo de investigação de modelos e análise de performance, e não apresentou uma implementação que pudesse ser utilizada comercial ou academicamente. Os modelos apresentados podem ser implementados

em uma ferramenta como a CoDiVision<sup>1</sup>, um sistema *web* que oferece vários serviços relacionados a mineração de repositório de software e análise de conhecimento em código fonte.

## 5.4 Trabalhos Futuros

Com relação a proposição de modelos ainda há outras informações que podem ser extraídas de sistemas de controle de versão para a identificação de expertise. Em investigações futuras pretende-se utilizar de informações como: quantidade de alterações em arquivos relacionados, similaridade de caminhos dos arquivos, complexidade do código fonte, dentre outras métricas. Além disso, pretende-se combinar essas informações com outras presentes em fontes como: repositório de *bugs* e sítios de perguntas e respostas de programação, com o objetivo de criar modelos ainda mais precisos.

Além da implementação dos modelos propostos, em trabalhos futuros pretende-se verificar o impacto da aplicação desses modelos em um contexto real de desenvolvimento de software. Para isso, pode-se comparar algumas métricas de eficiência e qualidade em cenários utilizando inferências feitas pelos modelos. Além disso, pode-se comparar a eficácia de diferentes técnicas em contextos mais elaborados que os apresentados neste trabalho. Pode-se compará-las em contextos de designação de atividades como os vários citados anteriormente.

---

<sup>1</sup> <https://easii.ufpi.br/codivision>



# Referências

AHSAN, S. N.; FERZUND, J.; WOTAWA, F. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In: IEEE. *2009 Fourth International Conference on Software Engineering Advances*. [S.l.], 2009. p. 216–221. Citado na página 16.

AL-JAMIMI, H. A.; AHMED, M. Machine learning-based software quality prediction models: state of the art. In: IEEE. *2013 International Conference on Information Science and Applications (ICISA)*. [S.l.], 2013. p. 1–4. Citado na página 14.

ALI, J. et al. Random forests and decision trees. *International Journal of Computer Science Issues (IJCSI)*, International Journal of Computer Science Issues (IJCSI), v. 9, n. 5, p. 272, 2012. Citado na página 15.

ALKHALID, A.; ALSHAYEB, M.; MAHMOUD, S. Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. *Advances in Engineering Software*, Elsevier, v. 41, n. 10-11, p. 1160–1178, 2010. Citado na página 16.

ALKHALID, A.; LUNG, C.-H.; AJILA, S. Software architecture decomposition using adaptive k-nearest neighbor algorithm. In: IEEE. *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. [S.l.], 2013. p. 1–4. Citado na página 16.

ALONSO, O.; DEVANBU, P. T.; GERTZ, M. Expertise identification and visualization from cvs. In: *Proceedings of the 2008 international working conference on Mining software repositories*. [S.l.: s.n.], 2008. p. 125–128. Citado 2 vezes nas páginas 26 e 39.

ALVES, F. V. de M. et al. Analysis of code familiarity in module and functionality perspectives. In: *Proceedings of the 17th Brazilian Symposium on Software Quality*. [S.l.: s.n.], 2018. p. 41–50. Citado 4 vezes nas páginas 37, 39, 42 e 44.

ANVIK, J.; HIEW, L.; MURPHY, G. C. Who should fix this bug? In: *Proceedings of the 28th international conference on Software engineering*. [S.l.: s.n.], 2006. p. 361–370. Citado 2 vezes nas páginas 3 e 56.

ANVIK, J.; MURPHY, G. C. Determining implementation expertise from bug reports. In: IEEE. *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. [S.l.], 2007. p. 2–2. Citado 5 vezes nas páginas 13, 14, 26, 27 e 55.

AVELINO, G. et al. A novel approach for estimating truck factors. In: IEEE. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. [S.l.], 2016. p. 1–10. Citado 4 vezes nas páginas 4, 31, 34 e 58.

AVELINO, G. et al. Assessing code authorship: The case of the linux kernel. In: SPRINGER, CHAM. *IFIP International Conference on Open Source Systems*. [S.l.], 2017. p. 151–163. Citado 2 vezes nas páginas 34 e 37.

AVELINO, G. et al. Who can maintain this code?: Assessing the effectiveness of repository-mining techniques for identifying software maintainers. *IEEE Software*, IEEE,

v. 36, n. 6, p. 34–42, 2018. Citado 13 vezes nas páginas 4, 25, 26, 27, 31, 34, 36, 39, 40, 41, 42, 44 e 46.

BADAMPUDI, D.; BRITTO, R.; UNTERKALMSTEINER, M. Modern code reviews-preliminary results of a systematic mapping study. In: *Proceedings of the Evaluation and Assessment on Software Engineering*. [S.l.: s.n.], 2019. p. 340–345. Citado na página 13.

BERRAR, D. Cross-validation. *Encyclopedia of bioinformatics and computational biology*, Academic, v. 1, p. 542–545, 2019. Citado na página 18.

BIRD, C. et al. Don't touch my code! examining the effects of ownership on software quality. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 4–14. Citado na página 4.

BISHOP, C. M. *Pattern recognition and machine learning*. [S.l.]: springer, 2006. Citado na página 16.

BORGES, H.; VALENTE, M. T. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, Elsevier, v. 146, p. 112–129, 2018. Citado na página 31.

BREIMAN, L. Random forests. *Machine learning*, Springer, v. 45, n. 1, p. 5–32, 2001. Citado na página 15.

CANFORA, G.; CERULO, L.; PENTA, M. D. Identifying changed source code lines from version repositories. In: IEEE. *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. [S.l.], 2007. p. 14–14. Citado na página 37.

CASALNUOVO, C. et al. Assert use in github projects. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 755–766. Citado na página 4.

CASTRO, D.; SCHOTS, M. Analysis of test log information through interactive visualizations. In: *Proceedings of the 26th Conference on Program Comprehension*. [S.l.: s.n.], 2018. p. 156–166. Citado na página 31.

CHOMBOON, K. et al. An empirical study of distance metrics for k-nearest neighbor algorithm. In: *Proceedings of the 3rd international conference on industrial application engineering*. [S.l.: s.n.], 2015. p. 280–285. Citado na página 16.

CLAESSEN, M.; MOOR, B. D. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015. Citado na página 47.

CONSTANTINOU, E.; KAPITSAKI, G. 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 na página 39.

COSTA, C. et al. Tipmerge: recommending experts for integrating changes across branches. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2016. p. 523–534. Citado na página 3.

- COSTA, C. et al. Tipmerge: recommending experts for integrating changes across branches. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2016. p. 523–534. Citado 3 vezes nas páginas 23, 27 e 56.
- COSTA, C. et al. Recommending participants for collaborative merge sessions. 2019. Citado na página 3.
- COSTA, C. et al. Recommending participants for collaborative merge sessions. 2019. Citado 2 vezes nas páginas 23 e 27.
- DURELLI, V. H. et al. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, IEEE, v. 68, n. 3, p. 1189–1212, 2019. Citado na página 14.
- EBBINGHAUS, H. *Über das gedächtnis: untersuchungen zur experimentellen psychologie*. [S.l.]: Duncker & Humblot, 1885. Citado na página 25.
- ELISH, K. O.; ELISH, M. O. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, Elsevier, v. 81, n. 5, p. 649–660, 2008. Citado na página 16.
- FALCÃO, F. et al. On relating technical, social factors, and the introduction of bugs. In: IEEE. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2020. p. 378–388. Citado 3 vezes nas páginas 23, 27 e 39.
- FERREIRA, M. et al. Algorithms for estimating truck factors: a comparative study. *Software Quality Journal*, Springer, v. 27, n. 4, p. 1583–1617, 2019. Citado na página 4.
- FERREIRA, M.; VALENTE, M. T.; FERREIRA, K. A comparison of three algorithms for computing truck factors. In: IEEE. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.], 2017. p. 207–217. Citado na página 4.
- FOLLECO, A. et al. Software quality modeling: The impact of class noise on the random forest classifier. In: IEEE. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. [S.l.], 2008. p. 3853–3859. Citado na página 15.
- FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, JSTOR, p. 1189–1232, 2001. Citado na página 46.
- FRITZ, T.; MURPHY, G. C.; HILL, E. Does a programmer’s activity indicate knowledge of code? In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.: s.n.], 2007. p. 341–350. Citado na página 39.
- FRITZ, T. et al. Degree-of-knowledge: Modeling a developer’s knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 23, n. 2, p. 1–42, 2014. Citado 7 vezes nas páginas 3, 4, 23, 27, 39, 41 e 42.
- FRITZ, T. et al. A degree-of-knowledge model to capture source code familiarity. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. [S.l.: s.n.], 2010. p. 385–394. Citado 4 vezes nas páginas 3, 4, 24 e 27.

- GHAZI, A. N. et al. Survey research in software engineering: Problems and mitigation strategies. *IEEE Access*, IEEE, v. 7, p. 24703–24718, 2018. Citado na página 36.
- GIRBA, T. et al. How developers drive software evolution. In: IEEE. *Eighth international workshop on principles of software evolution (IWPSE'05)*. [S.l.], 2005. p. 113–122. Citado 6 vezes nas páginas 4, 23, 26, 27, 39 e 42.
- GÜEMES-PEÑA, D. et al. Emerging topics in mining software repositories. *Progress in Artificial Intelligence*, Springer, v. 7, n. 3, p. 237–247, 2018. Citado na página 14.
- GUNN, S. R. et al. Support vector machines for classification and regression. *ISIS technical report*, v. 14, n. 1, p. 5–16, 1998. Citado na página 15.
- HANNEBAUER, C. et al. Automatically recommending code reviewers based on their expertise: An empirical comparison. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2016. p. 99–110. Citado 8 vezes nas páginas 3, 13, 25, 26, 27, 39, 41 e 42.
- HASANLUO, M.; GHAREHCHOPOGH, F. S. Software cost estimation by a new hybrid model of particle swarm optimization and k-nearest neighbor algorithms. *Journal of Electrical and Computer Engineering Innovations (JECEI)*, Shahid Rajaei Teacher Training University, v. 4, n. 1, p. 49–55, 2016. Citado na página 16.
- HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. *The elements of statistical learning: data mining, inference, and prediction*. [S.l.]: Springer Science & Business Media, 2009. Citado 2 vezes nas páginas 18 e 19.
- HATTORI, L.; LANZA, M. Mining the history of synchronous changes to refine code ownership. In: IEEE. *2009 6th IEEE international working conference on mining software repositories*. [S.l.], 2009. p. 141–150. Citado 3 vezes nas páginas 23, 27 e 39.
- HATTORI, L.; LANZA, M. Syde: a tool for collaborative software development. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. [S.l.: s.n.], 2010. p. 235–238. Citado 2 vezes nas páginas 23 e 27.
- HAWKINS, D. M. The problem of overfitting. *Journal of chemical information and computer sciences*, ACS Publications, v. 44, n. 1, p. 1–12, 2004. Citado na página 18.
- HERBSLEB, J. D.; GRINTER, R. E. Splitting the organization and integrating the code: Conway's law revisited. In: *Proceedings of the 21st international conference on Software engineering*. [S.l.: s.n.], 1999. p. 85–95. Citado na página 3.
- HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: IEEE. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2016. p. 426–437. Citado na página 31.
- HOSSEN, M. K.; KAGDI, H.; POSHYVANYK, D. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In: *Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.: s.n.], 2014. p. 130–141. Citado 5 vezes nas páginas 3, 4, 23, 27 e 56.
- IBIAPINA, I. M. S. et al. Inferência da familiaridade de código por meio da mineração de repositórios de software. *Simpósio Brasileiro de Qualidade de Software - SBQS*, 2017. Citado 5 vezes nas páginas 37, 39, 42, 44 e 55.



- JIANG, J. et al. Understanding inactive yet available assignees in github. *Information and Software Technology*, Elsevier, v. 91, p. 44–55, 2017. Citado na página 31.
- JR, D. W. H.; LEMESHOW, S.; STURDIVANT, R. X. *Applied logistic regression*. [S.l.]: John Wiley & Sons, 2013. v. 398. Citado na página 46.
- KAGDI, H. et al. Assigning change requests to software developers. *Journal of software: Evolution and Process*, Wiley Online Library, v. 24, n. 1, p. 3–33, 2012. Citado 5 vezes nas páginas 3, 24, 27, 39 e 41.
- KAGDI, H.; HAMMAD, M.; MALETIC, J. I. Who can help me with this source code change? In: IEEE. *2008 IEEE International Conference on Software Maintenance*. [S.l.], 2008. p. 157–166. Citado 3 vezes nas páginas 3, 24 e 27.
- KAGDI, H.; POSHYVANYK, D. Who can help me with this change request? In: IEEE. *2009 IEEE 17th International Conference on Program Comprehension*. [S.l.], 2009. p. 273–277. Citado 5 vezes nas páginas 3, 24, 27, 39 e 56.
- KASUNIC, M. *Designing an effective survey*. [S.l.], 2005. Citado na página 35.
- KAUR, A.; MALHOTRA, R. Application of random forest in predicting fault-prone classes. In: IEEE. *2008 International Conference on Advanced Computer Theory and Engineering*. [S.l.], 2008. p. 37–43. Citado na página 15.
- KEELE, S. et al. *Guidelines for performing systematic literature reviews in software engineering*. [S.l.], 2007. Citado na página 21.
- KERSTEN, M. *Focusing knowledge work with task context*. Tese (Doutorado), 02 2007. Citado 2 vezes nas páginas 24 e 27.
- KIM, J.; LEE, E. Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation. *Symmetry*, Multidisciplinary Digital Publishing Institute, v. 10, n. 4, p. 114, 2018. Citado na página 13.
- KRÜGER, J. et al. Do you remember this source code? In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.], 2018. p. 764–775. Citado 7 vezes nas páginas 13, 25, 26, 27, 37, 39 e 42.
- KULKARNI, P. *Reinforcement and systemic machine learning for decision making*. [S.l.]: John Wiley & Sons, 2012. v. 1. Citado na página 14.
- LIAW, A.; WIENER, M. et al. Classification and regression by randomforest. *R news*, v. 2, n. 3, p. 18–22, 2002. Citado na página 46.
- LINAKER, J. et al. *Guidelines for conducting surveys in software engineering* v. 1.1. *Lund University*, 2015. Citado na página 35.
- LIRA, W. A. L. *Um método para inferência da familiaridade de código em projetos de software*. Dissertação (Mestrado) — Universidade Federal do Piauí, Teresina, 9 2016. Citado 7 vezes nas páginas 26, 37, 39, 41, 42, 44 e 55.
- LOELIGER, J.; MCCULLOUGH, M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado 2 vezes nas páginas 4 e 11.

LUCCA, G. A. D.; PENTA, M. D.; GRADARA, S. An approach to classify software maintenance requests. In: IEEE. *International Conference on Software Maintenance, 2002. Proceedings*. [S.l.], 2002. p. 93–102. Citado na página 16.

LUGER, G. F. *Inteligência Artificial: Estruturas e estratégias para a solução de problemas complexos*. [S.l.]: Bookman, 2004. Citado na página 14.

MANI, S.; PADHYE, R.; SINHA, V. S. Mining api expertise profiles with partial program analysis. In: *Proceedings of the 9th India Software Engineering Conference*. [S.l.: s.n.], 2016. p. 109–118. Citado na página 13.

MAZINANIAN, D. et al. Understanding the use of lambda expressions in java. *Proceedings of the ACM on Programming Languages*, ACM New York, NY, USA, v. 1, n. OOPSLA, p. 1–31, 2017. Citado na página 31.

MCDONALD, D. W.; ACKERMAN, M. S. Expertise recommender: a flexible recommendation system and architecture. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. [S.l.: s.n.], 2000. p. 231–240. Citado 5 vezes nas páginas 4, 22, 26, 27 e 39.

MEEK, C.; THIESSON, B.; HECKERMAN, D. The learning-curve sampling method applied to model-based clustering. *Journal of Machine Learning Research*, v. 2, n. Feb, p. 397–418, 2002. Citado na página 55.

MINTO, S.; MURPHY, G. C. Recommending emergent teams. In: IEEE. *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. [S.l.], 2007. p. 5–5. Citado 4 vezes nas páginas 4, 23, 27 e 39.

MOCKUS, A.; HERBSLEB, J. D. Expertise browser: a quantitative approach to identifying expertise. In: IEEE. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. [S.l.], 2002. p. 503–512. Citado 4 vezes nas páginas 4, 23, 27 e 39.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. *Foundations of machine learning*. [S.l.]: MIT press, 2018. Citado na página 14.

MONTANDON, J. E.; SILVA, L. L.; VALENTE, M. T. Identifying experts in software libraries and frameworks among github users. In: IEEE. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2019. p. 276–287. Citado 8 vezes nas páginas 13, 14, 15, 16, 25, 27, 36 e 39.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the eclipse ide? *IEEE software*, IEEE, v. 23, n. 4, p. 76–83, 2006. Citado 2 vezes nas páginas 24 e 27.

NASSIF, A. B. et al. A treeboost model for software effort estimation based on use case points. In: IEEE. *2012 11th International Conference on Machine Learning and Applications*. [S.l.], 2012. v. 2, p. 314–319. Citado na página 15.

NATEKIN, A.; KNOLL, A. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, Frontiers, v. 7, p. 21, 2013. Citado na página 15.

NAVARRO, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 33, n. 1, p. 31–88, 2001. Citado 2 vezes nas páginas 34 e 37.

NIELEBOCK, S.; HEUMÜLLER, R.; ORTMEIER, F. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empirical Software Engineering*, Springer, v. 24, n. 1, p. 103–138, 2019. Citado na página 31.

OLIVEIRA, E. et al. Code and commit metrics of developer productivity: a study on team leaders perceptions. *EMPIRICAL SOFTWARE ENGINEERING*, Springer, 2020. Citado na página 45.

OLIVEIRA, J.; VIGGIATO, M.; FIGUEIREDO, E. How well do you know this library? mining experts from source code analysis. In: *Proceedings of the XVIII Brazilian Symposium on Software Quality*. [S.l.: s.n.], 2019. p. 49–58. Citado 2 vezes nas páginas 13 e 39.

OTTE, S. Version control systems. *Computer Systems and Telematics*, p. 11–13, 2009. Citado na página 11.

OVERHOLSER, B. R.; SOWINSKI, K. M. Biostatistics primer: part 2. *Nutrition in clinical practice*, Wiley Online Library, v. 23, n. 1, p. 76–84, 2008. Citado na página 58.

PADHYE, R.; MANI, S.; SINHA, V. S. A study of external community contribution to open-source projects on github. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 332–335. Citado na página 31.

PETERSON, L. E. K-nearest neighbor. *Scholarpedia*, v. 4, n. 2, p. 1883, 2009. Citado 2 vezes nas páginas 16 e 46.

PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. W. *Version control with subversion: next generation open source version control*. [S.l.: "O'Reilly Media, Inc."], 2008. Citado na página 12.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *Bioinfo Publications*, 2011. Citado 2 vezes nas páginas 17 e 18.

RAHMAN, F.; DEVANBU, P. Ownership, experience and defects: a fine-grained study of authorship. In: *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.: s.n.], 2011. p. 491–500. Citado 5 vezes nas páginas 4, 23, 27, 39 e 42.

RAJLICH, V. Software evolution and maintenance. In: *Proceedings of the on Future of Software Engineering*. [S.l.: s.n.], 2014. p. 133–144. Citado na página 3.

RAY, B. et al. A large scale study of programming languages and code quality in github. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. p. 155–165. Citado na página 31.

RIGBY, P. C.; BIRD, C. Convergent contemporary software peer review practices. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. [S.l.: s.n.], 2013. p. 202–212. Citado na página 57.

RIGGER, M. et al. An analysis of x86-64 inline assembly in c programs. In: *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. [S.l.: s.n.], 2018. p. 84–99. Citado na página 31.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3rd. ed. USA: Prentice Hall Press, 2009. ISBN 0136042597. Citado na página 14.

SAJEDI-BADASHIAN, A.; STROULIA, E. Guidelines for evaluating bug-assignment research. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2250, 2020. Citado na página 25.

SATAPATHY, S. M.; ACHARYA, B. P.; RATH, S. K. Class point approach for software effort estimation using stochastic gradient boosting technique. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 39, n. 3, p. 1–6, 2014. Citado na página 15.

SATAPATHY, S. M.; ACHARYA, B. P.; RATH, S. K. Early stage software effort estimation using random forest technique based on use case points. *IET Software*, IET, v. 10, n. 1, p. 10–17, 2016. Citado na página 15.

SATAPATHY, S. M.; RATH, S. K. Empirical assessment of machine learning models for agile software development effort estimation using story points. *Innovations in Systems and Software Engineering*, Springer, v. 13, n. 2-3, p. 191–200, 2017. Citado na página 15.

SCHOBER, P.; BOER, C.; SCHWARTE, L. A. Correlation coefficients: appropriate use and interpretation. *Anesthesia & Analgesia*, Wolters Kluwer, v. 126, n. 5, p. 1763–1768, 2018. Citado 3 vezes nas páginas 39, 40 e 58.

SCHULER, D.; ZIMMERMANN, T. Mining usage expertise from version archives. In: *Proceedings of the 2008 international working conference on Mining software repositories*. [S.l.: s.n.], 2008. p. 121–124. Citado na página 13.

SHEPPERD, M.; BOWES, D.; HALL, T. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, IEEE, v. 40, n. 6, p. 603–616, 2014. Citado na página 14.

SILVA, J. R. da et al. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In: IEEE. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2015. p. 409–418. Citado 5 vezes nas páginas 4, 24, 27, 39 e 41.

SMITH, E. et al. Improving developer participation rates in surveys. In: IEEE. *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. [S.l.], 2013. p. 89–92. Citado na página 36.

SPINELLIS, D. Version control systems. *IEEE Software*, IEEE, v. 22, n. 5, p. 108–109, 2005. Citado na página 11.

SÜLÜN, E.; TÜZÜN, E.; DOĞRUSÖZ, U. Reviewer recommendation using software artifact traceability graphs. In: *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. [S.l.: s.n.], 2019. p. 66–75. Citado 6 vezes nas páginas 3, 13, 23, 24, 27 e 39.

- SÜLÜN, E.; TÜZÜN, E.; DOĞRUSÖZ, U. Rstrace+: Reviewer suggestion using software artifact traceability graphs. *Information and Software Technology*, Elsevier, v. 130, p. 106455, 2020. Citado 5 vezes nas páginas 3, 13, 24, 27 e 39.
- THONGTANUNAM, P. et al. Improving code review effectiveness through reviewer recommendations. In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. [S.l.: s.n.], 2014. p. 119–122. Citado na página 26.
- THONGTANUNAM, P. et al. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: *Proceedings of the 38th international conference on software engineering*. [S.l.: s.n.], 2016. p. 1039–1050. Citado na página 57.
- TRANMER, M.; ELLIOT, M. Multiple linear regression. *The Cathie Marsh Centre for Census and Survey Research (CCSR)*, v. 5, p. 30–35, 2008. Citado na página 42.
- WEN, J. et al. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, Elsevier, v. 54, n. 1, p. 41–59, 2012. Citado na página 14.
- WESTON, J.; WATKINS, C. *Multi-class support vector machines*. [S.l.], 1998. Citado na página 46.
- WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 56.
- XING, F.; GUO, P.; LYU, M. R. A novel method for early software quality prediction based on support vector machine. In: IEEE. *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. [S.l.], 2005. p. 10–pp. Citado na página 16.
- YAMASHITA, K. et al. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In: *Proceedings of the 14th International Workshop on Principles of Software Evolution*. [S.l.: s.n.], 2015. p. 46–55. Citado 2 vezes nas páginas 31 e 45.
- YU, L.; LIU, H. Feature selection for high-dimensional data: A fast correlation-based filter solution. In: *Proceedings of the 20th international conference on machine learning (ICML-03)*. [S.l.: s.n.], 2003. p. 856–863. Citado 2 vezes nas páginas 40 e 55.



## Apêndices





## APÊNDICE A – Correlações entre as variáveis extraídas

A Tabela 15 mostra as correlações existentes entre as variáveis extraídas dos históricos de desenvolvimento dos projetos analisados. As escolhas das variáveis que compuseram os modelos foram influenciadas pelas correlações encontradas.

	Adds	Dels	Mods	Conds	Amount	NumCommits	NumDays	NumModDevs	Blame	Size	FA	AvgDaysCommits
Adds	1	0,29	0,27	0,52	0,96	0,55	-0,06	-0,02	0,64	0,22	0,48	0,48
Dels	0,29	1	0,83	0,25	0,45	0,57	-0,09	-0,04	0,05	0,25	-0,23	0,5
Mods	0,27	0,83	1	0,23	0,36	0,51	-0,07	-0,04	0,04	0,26	-0,22	0,44
Conds	0,52	0,25	0,23	1	0,52	0,34	-0,07	0,01	0,32	0,26	0,21	0,3
Amount	0,96	0,45	0,36	0,52	1	0,58	-0,08	-0,03	0,59	0,26	0,48	0,51
NumCommits	0,55	0,57	0,51	0,34	0,58	1	-0,05	0,02	0,25	0,12	0,16	0,88
NumDays	-0,06	-0,09	-0,07	-0,07	-0,08	-0,05	1	0,5	-0,12	-0,05	0,03	-0,1
NumModDevs	-0,02	-0,04	-0,04	0,01	-0,03	0,02	0,5	1	-0,13	0,26	-0,03	-0,05
Blame	0,64	0,05	0,04	0,32	0,59	0,25	-0,12	-0,13	1	0,06	0,47	0,24
Size	0,22	0,25	0,26	0,26	0,26	0,12	-0,05	0,26	0,06	1	-0,17	0,1
FA	0,48	-0,23	-0,22	0,21	0,48	0,16	0,03	-0,03	0,47	-0,17	1	0,16
AvgDaysCommits	0,48	0,5	0,44	0,3	0,51	0,88	-0,1	-0,05	0,24	0,1	0,16	1

Tabela 15 – Correlações entre as 12 (doze) variáveis extraídas dos históricos de desenvolvimento.