UNIVERSIDADE FEDERAL FLUMINENSE

MANOEL LIMEIRA DE LIMA JÚNIOR

PREVISÃO DE INTEGRADORES E TEMPO DE VIDA DE PULL REQUESTS

NITERÓI

2017

UNIVERSIDADE FEDERAL FLUMINENSE

MANOEL LIMEIRA DE LIMA JÚNIOR

PREVISÃO DE INTEGRADORES E TEMPO DE VIDA DE PULL REQUESTS

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Engenharia de Sistemas e Informação.

Orientador:
Alexandre Plastino de Carvalho

Coorientador:
Leonardo Gresta Paulino Murta

NITERÓI

2017

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

L732 Lima Júnior, Manoel Limeira de
Previsão de integradores e tempo de vida de *pull requests* / Manoel Limeira de Lima Júnior. – Niterói, RJ : [s.n.], 2017.
113 f.

Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2017.
Orientadores: Alexandre Plastino de Carvalho, Leonardo Gresta Paulino Murta.

1. Engenharia de software. 2. Inteligência artificial. 3. Mineração de dados (Computação). 4. Engenharia de sistemas. I. Título.

CDD 005.1

MANOEL LIMEIRA DE LIMA JÚNIOR

PREVISÃO DE INTEGRADORES E TEMPO DE VIDA DE PULL REQUESTS

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Engenharia de Sistemas e Informação.

Aprovada em Julho de 2017.

BANCA EXAMINADORA

Prof. D.Sc. Alexandre Plastino de Carvalho - Orientador, UFF

Prof. D.Sc. Leonardo Gresta Paulino Murta - Coorientador, UFF

Prof. D.Sc. Anselmo Antunes Montenegro, UFF

Prof. D.Sc. José Viterbo Filho, UFF

Prof. D.Sc. Eduardo Soares Ogasawara, CEFET/RJ

Prof. D.Sc. Renata Mendes de Araujo, UNIRIO

Niterói

2017

in memoriam: Manoel Limeira de Lima e Maria Edna Barreto de Lima.

Agradecimentos

Em primeiro lugar agradeço a Deus pelo mais importante: a vida.

Aos professores Alexandre Plastino e Leonardo Murta pelas muitas contribuições, pelos valiosos conselhos e pelo tempo dedicado durante os últimos cinco anos.

Aos amigos que compartilharam dessa intensa jornada Daricélio Soares e Laura Sarkis pelo companheirismo e pela ótima convivência que tivemos.

Ao professor Sérgio Brazil que "só na manha" auxiliou o programa de doutorado em Computação.

Aos meus eternos irmãos: Antonio Rogério Barreto de Lima, Rones Antonio Barreto de Lima e Rúbia Antonia Barreto de Lima.

Aos meus tios e tias, primos e primas, sobrinhos e sobrinhas que tanto amo.

À minha amada esposa Andressa Almeida de Souza Limeira por continuar me incentivando nas horas difíceis.

A todos que me ajudaram de alguma forma na realização deste trabalho, meu muito obrigado.

E para meus amigos e familiares um pedido de desculpas pela ausência em momentos importantes.

À UFAC, UFF e Capes pelo apoio financeiro em parte do doutorado.

Resumo

No desenvolvimento distribuído de software, muitos projetos têm adotado o fluxo de trabalho "gerente de integração" para organizar o processo de colaboração. Nesse fluxo, o desenvolvedor (solicitante) envia sua colaboração por meio de uma solicitação, denominada *pull request*, que contém título, descrição, autor e o código necessário para corrigir um *bug* ou adicionar uma funcionalidade. A solicitação deve ser revisada por um membro da equipe principal do projeto (integrador), que decide rejeitar ou aceitar o código. Em muitos projetos *open-source*, o processo de integração de *pull requests* possui grande demanda de solicitações e pouca oferta de tempo dos integradores, o que aumenta a quantidade de *pull requests* abertos e o tempo médio das integrações. Por outro lado, os integradores estão interessados em reduzir o tempo de integração dos *pull requests* e garantir a qualidade do código. Nesse processo de tomada de decisões, pode ser extremamente útil prever informações como, por exemplo, o integrador mais apropriado para integrar e o tempo de vida de um *pull request*. Alguns trabalhos já exploraram esses cenários de previsão. No entanto, esses trabalhos se diferenciam em vários aspectos referentes aos materiais e métodos utilizados: conjuntos de atributos preditivos, técnicas de classificação e regressão, processos experimentais e quantidade de projetos. Nesse contexto, os principais objetivos desta tese são: comparar os diferentes conjuntos de atributos preditivos utilizados em trabalhos anteriores com o conjunto aqui proposto e avaliar se técnicas de seleção de atributos podem identificar subconjuntos de atributos mais adequados para melhorar o desempenho das tarefas de prever integradores e tempo de vida de *pull requests*. Nos experimentos, os conjuntos de atributos foram avaliados com diferentes algoritmos de classificação, regressão e estratégias de seleção de atributos. Comparando com abordagens anteriores, nossa proposta para recomendar integradores apresentou a melhor acurácia em 29 dos 32 projetos considerando a recomendação Top-1 e atingiu as melhores médias de ganho normalizado para as recomendações Top-1 (19,93%), Top-3 (41,91%) e Top-5 (52,60%). Na previsão do tempo de vida, nossa proposta também apresentou as melhores médias de ganho normalizado quando comparada com outras abordagens, obtendo a melhor acurácia em 18 dos 20 projetos utilizados e um ganho normalizado médio de 14,68%.

Palavras-chave: *Pull Request*, Integradores, Tempo de Vida, Classificação, Regressão e Seleção de Atributos.

Abstract

In distributed software development, many projects have adopted the "integration manager" workflow to organize the collaboration process. In this workflow, the developer (requester) sends his or her contribution through a request, called pull request, which contains title, description, author, and the code needed to fix bugs or add features. The pull request must be reviewed by a member of the core team (integrator), who decides to reject or accept it. In many open-source projects, the pull requests integration process has a high demand for requests and a shortage of integrators' time, which increases the number of opened pull requests and the average time for integrations. On the other hand, integrators are interested in reducing the time to integrate the pull requests and ensure the code quality. In this decision-making process, it may be extremely useful to predict information, such as the most appropriate integrator to integrate and the lifetime of a pull request. Some researches have already explored these forecasting scenarios. However, these researches differ in many aspects related to the materials and methods used: sets of predictive attributes, classification and regression techniques, experimental processes, and quantity of projects. In this context, the main objectives of this thesis are: to compare the different sets of predictive attributes used in previous works with the set proposed here and to evaluate whether attributes selection techniques can identify more adequate subsets of attributes in order to improve the performance of the tasks of predicting integrators and lifetime of pull requests. In the experiments, the sets of attributes were evaluated with different classification, regression, and attribute selection strategies. Compared to previous approaches, our proposal to recommend integrators showed the best accuracy in 29 out of the 32 projects considering the Top-1 recommendation and reached the best normalized improvement averages for the recommendations Top-1 (19.93%), Top-3 (41.91%), and Top-5 (52.60%). In the prediction of lifetime, our proposal also presented the best normalized improvement averages when compared to other approaches, obtaining the best accuracy in 18 out of the 20 projects used and normalized improvement average of 14,68%.

Keywords: Pull Request, Integrators, Lifetime, Classification, Regression, and Attribute Selection.

Lista de Figuras

1.1 Visão geral do fluxo de trabalho "gerente de integração". 2

2.1 *Workflow* centralizado [1]. 10

2.2 *Workflow* "gerente de integração" [1]. 11

2.3 Listagem de *pull requests* abertos do projeto rails no Github. 14

2.4 Matriz de Confusão [2]. 18

2.5 Exemplo de Árvore de Decisão [2]. 22

2.6 Margens e hiperplanos separadores [2]. 23

2.7 Mudança de dimensão com *kernel* no SVM. 24

2.8 Exemplo de Árvore de Modelo [3]. 27

2.9 SVM para regressão [3]. 28

5.1 Quantidade de *pull requests* por mês no projeto rails [4]. 67

5.2 Tempo de vida dos *pull requests* nos projetos selecionados. 72

5.3 Tempo de vida de *pull requests* por projeto. 73

Lista de Tabelas

4.1 Atributos do Conjunto *A*. 43

4.2 Atributos do Conjunto *B*. 45

4.3 Atributos do Conjunto *C*. 46

4.4 Características dos projetos. 49

4.5 Resultados obtidos pelo IBk com diferentes valores de k para o Conjunto *C*. 54

4.6 Resultados obtidos pelo SMO com *kernels Polynomial*, *Puk* e RBF para os Conjuntos *B* e *C*. 55

4.7 Resultados obtidos pelo *Random Forest* com diferentes números de árvores para o Conjunto *C*. 55

4.8 Resultados obtidos pelos diferentes classificadores no Conjunto *C*. 56

4.9 Resultados obtidos após a discretização de atributos no Conjunto *C*. 56

4.10 Resultados obtidos pelo algoritmo RF com diferentes números de atributos para a estratégia IGAR. 57

4.11 Resultados obtidos após seleção de atributos no Conjunto *C*. 58

4.12 Resultados obtidos pelas três abordagens para as recomendações Top-1, Top-3 e Top-5. 59

4.13 Ganho normalizado das Abordagens A, B e C por projeto 60

4.14 Importância dos atributos. 63

5.1 Atributos do Conjunto *D*. 69

5.2 Atributos adicionados ao Conjunto *D* para formar o Conjunto *E*. 71

5.3 Características dos projetos. 74

5.4 Resultados obtidos com regressão linear utilizando o Conjunto *D*. 78

5.5 Resultados obtidos com algoritmos de regressão utilizando o Conjunto *D*. . 79

5.6 Resultados obtidos com a variação do valor de k no IBk. 81

5.7 Resultados obtidos com a variação do *kernel* no SMO. 81

5.8 Resultados obtidos com a variação do número de árvores no *Random Forest*. 81

5.9 Resultados obtidos com algoritmos de classificação utilizando o Conjunto E 82

5.10 Resultados obtidos para a estratégia Relief-F. 83

5.11 Resultados obtidos para a estratégia IGAR. 84

5.12 Resultados obtidos com seleção de atributos utilizando o Conjunto E. . . . 84

5.13 Resultados obtidos pelas Abordagens A, B e C. 85

5.14 Ganho normalizado obtido pelas abordagens, por projeto. 86

5.15 Importância dos atributos. 87

Conteúdo

1 Introdução **1**

1.1 Contexto 1

1.2 Motivação 3

1.3 Objetivos 4

1.4 Organização 5

2 Revisão de Conceitos **6**

2.1 Introdução 6

2.2 Gerência de Configuração de Software 7

2.3 *Pull Requests* 11

2.4 Análise de Repositórios 14

2.5 Mineração de Dados 15

2.6 Algoritmos de Classificação 20

2.7 Algoritmos de Regressão 25

2.8 Considerações Finais 29

3 Trabalhos Relacionados **30**

3.1 Introdução 30

3.2 Trabalhos sobre Integração de *Pull Requests* 31

3.3 Trabalhos sobre Previsão de Desenvolvedores 33

3.4 Trabalhos sobre Tempo de Vida de *Pull Requests* 37

3.5 Considerações Finais 39

4 Previsão de Integradores de Pull Requests 41

4.1 Introdução 41

4.2 Materiais e Métodos 42

4.2.1 Conjuntos de Atributos 43

4.2.2 Dados e Projetos 47

4.2.3 Algoritmos de Classificação, Discretização e Seleção de Atributos . 50

4.2.4 Processo Experimental 52

4.3 Resultados e Discussões 53

4.3.1 Ajuste de Parâmetros dos Algoritmos de Classificação 54

4.3.2 Comparação dos Algoritmos de Classificação no Conjunto C 55

4.3.3 Melhorando o Desempenho com Discretização 56

4.3.4 Identificando Melhores Subconjuntos de Atributos 57

4.3.5 Avaliação das Diferentes Abordagens 58

4.3.6 Importância dos Atributos 62

4.4 Limitações e Ameaças à Validade dos Resultados 63

4.5 Considerações Finais 64

5 Previsão do Tempo de Vida de Pull Requests 66

5.1 Introdução 66

5.2 Materiais e Métodos 68

5.2.1 Conjuntos de Atributos 68

5.2.2 Dados e Projetos 71

5.2.3 Algoritmos de Classificação, Regressão e Seleção de Atributos . . . 75

5.2.4 Processo Experimental 76

5.3 Resultados dos Experimentos 76

5.3.1 Previsão do Tempo de Vida com Regressão Linear 77

5.3.2 Previsão do Tempo de Vida com Algoritmos de Regressão 78

5.3.3 Ajuste de Parâmetros dos Algoritmos de Classificação 80

5.3.4 Previsão do Tempo de Vida com Classificação 82

5.3.5 Melhorando o Desempenho com Seleção de Atributos 83

5.3.6 Comparando o Desempenho das Diferentes Abordagens 85

5.3.7 Atributos Relevantes na Previsão do Tempo de Vida 86

5.4 Limitações e Ameaças à Validade dos Resultados 88

5.5 Considerações Finais 88

6 Conclusão **90**

6.1 Resultados e Contribuições 90

6.2 Trabalhos Futuros 92

Referências **93**

Capítulo 1

Introdução

1.1 Contexto

Nos Sistemas de Controle de Versões Distribuídos (SCVD), os dois principais fluxos de trabalho (*workflows*) utilizados pelas equipes de projeto de software para organizar o processo de colaboração entre os desenvolvedores são: "repositório compartilhado" e "gerente de integração" [1].

No fluxo de trabalho "repositório compartilhado", os desenvolvedores com permissão de escrita no repositório central copiam o repositório completo para suas estações de trabalho (operação *clone*), fazem as modificações necessárias localmente (operação *commit*) e enviam essas modificações diretamente para o repositório central (operação *push*).

No fluxo de trabalho "gerente de integração", o repositório principal do projeto é compartilhado com permissão de leitura para qualquer desenvolvedor. Com isso, um colaborador interno ou externo à equipe do projeto pode criar uma réplica do repositório no próprio servidor (operação *fork*), onde passa a ter permissão de escrita. Em seguida, o colaborador copia essa réplica para sua estação de trabalho (operação *clone*), podendo fazer as modificações desejadas localmente (operação *commit*) e enviar essas modificações para sua réplica do repositório no servidor (operação *push*). Essas etapas ocorrem de forma análoga ao fluxo de trabalho "repositório compartilhado", mas tendo como referência a réplica do repositório principal do projeto.

No entanto, para que o conjunto de modificações seja efetivado no repositório principal, o colaborador (solicitante) precisa submeter uma solicitação chamada *pull request*, já que não tem permissão de escrita no mesmo. Além disso, um membro da equipe interna do projeto (integrador) adiciona a réplica pública do repositório principal que pertence ao

solicitante como um repositório remoto, faz *pull* desse repositório e depois *merge* em seu repositório local. Em seguida, o integrador deve revisar as modificações e, eventualmente, solicitar esclarecimentos por meio de comentários. Por fim, o integrador, que tem permissão de escrita no repositório principal, deve efetuar o processo de integração, ou seja, definir se o *pull request* será rejeitado (*closed*) ou aceito (*merged*) e, consequentemente, enviar as modificações ao repositório principal (operação *push*) [1].

Para apresentar uma visão geral do processo que envolve o desenvolvimento baseado em *pull requests*, a Figura 1.1 ilustra o funcionamento básico do fluxo de trabalho "gerente de integração" modelado com a notação BPMN (*Business Process Model and Notation*).

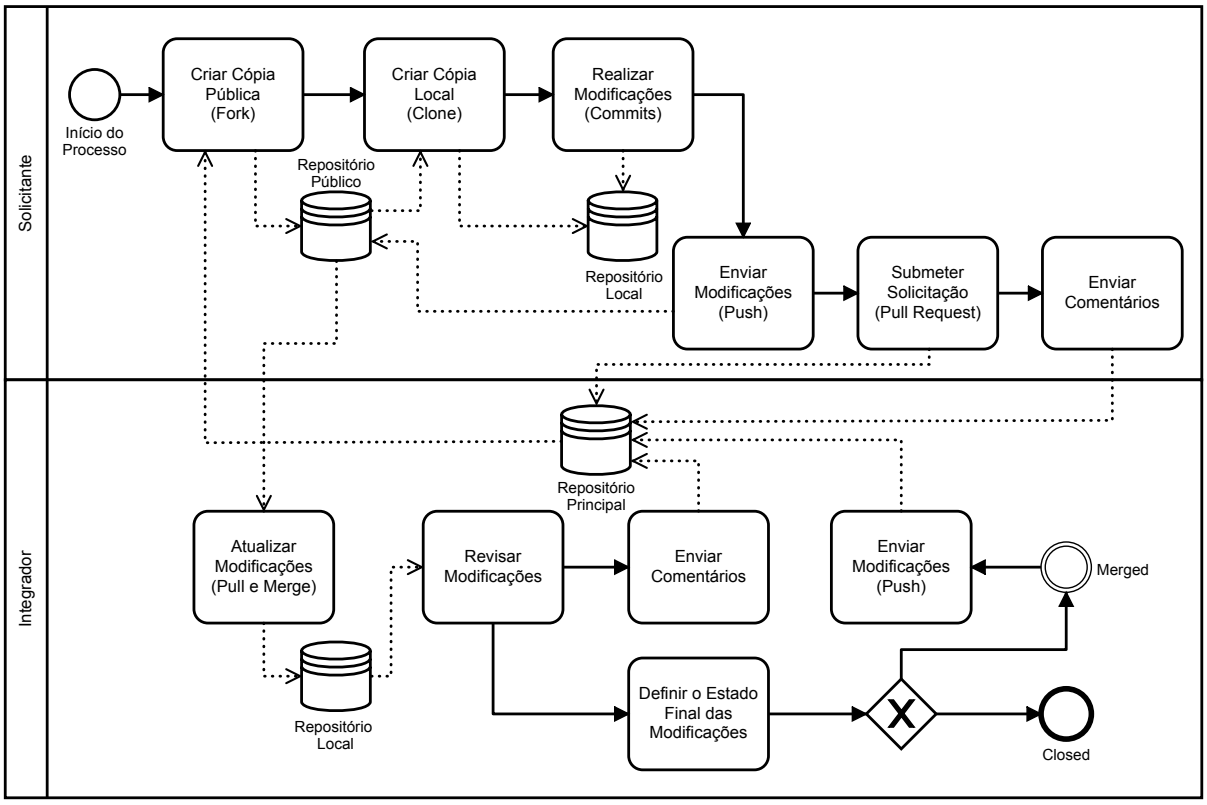

Figura 1.1: Visão geral do fluxo de trabalho "gerente de integração".

A possibilidade de desenvolvedores externos ao projeto enviarem suas contribuições proporcionou o crescimento e a popularização da utilização do modelo de desenvolvimento baseado em *pull requests*, principalmente em projetos *open-source* [5].

Inicialmente, o modelo começou a ser utilizado para facilitar e incentivar o envio de contribuições de desenvolvedores externos ao projeto. Contudo, em vários projetos (*e.g.*, rails, rosdistro e docker), os *pull requests* também passaram a ser utilizados pelos membros da equipe interna do projeto, viabilizando assim um processo de inspeção de código por outros membros da equipe antes da sua efetiva aceitação no repositório principal [4].

1.2 Motivação

O processo de integração de um *pull request* pode ser influenciado por diversas variáveis. Por exemplo, tamanho, localização, complexidade e prioridade do código submetido são fatores que podem influenciar no esforço e na experiência necessária do integrador alocado para realizar a integração. Nos casos em que o *pull request* modifica arquivos essenciais para o funcionamento do projeto, é muito importante que esse processo seja realizado, em tempo hábil, por um desenvolvedor com experiência e conhecimento sobre a área modificada. Além disso, um *pull request* pode gerar conflito com o código já existente no repositório principal, dificultando a decisão sobre o estado final (aceitar ou rejeitar), pois é preciso solucionar o conflito, geralmente modificando o código novo ou aquele que já estava no repositório principal.

Em muitos projetos *open-source* hospedados no GitHub[1], a quantidade de *pull requests* abertos é bastante alta [4]. Por outro lado, os desenvolvedores das equipes precisam dividir o seu tempo disponível entre a integração de *pull requests* e suas tarefas de desenvolvimento e manutenção. Esses fatores aumentam o tempo médio do processo de integração de *pull requests* e ampliam a lista dos *pull requests* que precisam ser integrados.

Para tornar o processo de integração dos *pull requests* mais eficiente, os integradores responsáveis pelo processo devem tomar decisões rápidas e corretas. Para isso, os integradores buscam reduzir o tempo de tratamento dos *pull requests* e garantir a qualidade do código que será aceito e incorporado ao projeto, além de priorizar aqueles que necessitam ser integrados [6]. Nesse processo de tomada de decisões, pode ser extremamente útil prever informações como, por exemplo, o integrador mais apropriado para fazer a integração e o tempo de vida de um *pull request*.

Na literatura, existem alguns trabalhos, bastante recentes, que exploraram cenários de previsão no contexto do modelo de desenvolvimento baseado em *pull requests* como, por exemplo, o problema de previsão de integradores [7] e tempo de vida de *pull requests* [4, 8].

Jiang *et al.* [7] utilizaram quatro algoritmos de classificação e cinco projetos *open-source* para prever integradores de *pull requests*. No entanto, a abordagem deixou de considerar atributos preditivos importantes como, por exemplo: (i) informações sobre o tamanho e a complexidade do *pull request*; (ii) características sobre a experiência dos solicitantes; e (iii) informações sobre as relações sociais entre solicitantes e integradores.

Gousios *et al.* [4] avaliaram algoritmos de classificação em dados de projetos *open-*

[1] `https://github.com/`

source para prever o tempo de vida de *pull requests* em três classes, ou seja, intervalos de tempo ($C_1 \leq 1$ hora $< C_2 \leq 1$ dia $< C_3$). Yu *et al.* [8] utilizaram a técnica de regressão linear múltipla para modelar a latência de *pull requests* extraídos de projetos *open-source*. Esses trabalhos utilizaram diferentes conjuntos de atributos preditivos, técnicas de previsão, processos experimentais e quantidade de projetos.

1.3 Objetivos

A partir do contexto e da motivação apresentados, o objetivo desta tese é melhorar o desempenho de previsões de informações em projetos *open-source* que adotam o modelo de desenvolvimento baseado em *pull requests*, referentes ao desenvolvedor apropriado para integrar *pull requests* e ao tempo de vida de *pull requests*.

Técnicas preditivas de mineração de dados (regressão e classificação) são tradicionalmente indicadas para fazer previsões e costumam apresentar bons desempenhos em diferentes domínios de aplicação [9]. Portanto, propõe-se nesta tese comparar os diferentes conjuntos de atributos preditivos utilizados em trabalhos anteriores com os conjuntos aqui propostos, visando melhorar a capacidade preditiva nos cenários acima mencionados. Nos experimentos realizados para conduzir essa investigação, os conjuntos de atributos foram avaliados com diferentes algoritmos de classificação e regressão. Além disso, nossa proposta inclui a utilização de técnicas de discretização e estratégias de seleção de atributos com o intuito de identificar subconjuntos de atributos mais adequados e aprimorar o desempenho dos algoritmos [10].

Com o objetivo de avaliar os algoritmos de forma adequada nas tarefas preditivas envolvendo dados de *pull requests*, nesta tese, propõe-se ainda um método de avaliação que divide os conjuntos de treinamento e teste preservando a ordem cronológica de submissão dos *pull requests*, denominado *training-test sliding validation*.

Para atingir essas metas investigativas, foram identificados os seguintes objetivos específicos:

1. Definir um método de avaliação para algoritmos preditivos adequado ao contexto do desenvolvimento baseado em *pull requests*;

2. Planejar e executar experimentos com diferentes algoritmos preditivos e diferentes conjuntos de atributos;

3. Avaliar o desempenho preditivo nas tarefas de previsão de integradores e tempo de vida de *pull requests*;

4. Aprimorar o desempenho das previsões com a utilização de estratégias de seleção de atributos.

1.4 Organização

Esta tese está organizada em outros cinco capítulos. O Capítulo 2 contém uma revisão de conceitos necessários para o entendimento dos procedimentos e dos dados utilizados nos experimentos de preditivos. Para isso, são apresentados conceitos que envolvem os sistemas de controle de versão e modificações da Gerência de Configuração de Software, a utilização do modelo de desenvolvimento baseado em *pull requests* e as técnicas preditivas da mineração de dados (classificação e regressão). O Capítulo 3 apresenta os trabalhos relacionados: tanto sobre a previsão da integração de *pull requests* quanto sobre os cenários de previsão de integradores e tempo de vida de *pull requests*. No Capítulo 4, são reportados os resultados dos experimentos com os algoritmos de classificação no cenário de previsão de desenvolvedores para integração de *pull requests*. No Capítulo 5, são apresentados os resultados dos experimentos com algoritmos de regressão e de classificação conduzidos com o intuito de prever o tempo de vida de um *pull request*. O Capítulo 6 contém as considerações finais e aponta alguns trabalhos futuros.

Capítulo 2

Revisão de Conceitos

2.1 Introdução

Gerência de Configuração de Software (GCS) é uma disciplina da Engenharia de Software (ES) que fornece mecanismos para o controle da evolução de sistemas de software [11]. Durante o ciclo de desenvolvimento e a manutenção do software, muitos artefatos são produzidos e alterados. Neste contexto, a GCS utiliza sistemas de apoio a fim de controlar as versões e as solicitações de modificação. Dentre esses sistemas, se destaca no contexto deste trabalho o Sistema de Controle de Versão Distribuído (SCVD).

Os SCVDs controlam e armazenam toda a história de um software. Além disso, fornecem uma arquitetura de serviços que visa facilitar o processo de colaboração no desenvolvimento distribuído de software. Um exemplo desses serviços é o fluxo de trabalho de "gerente de integração", onde as contribuições são enviadas por solicitações de mudanças chamadas de *pull requests*, que já contêm o código necessário para corrigir um *bug* ou criar uma funcionalidade. Em sites como o GitHub, muitos projetos *open-source* têm adotado esse fluxo de trabalho como principal forma de colaboração. Nesses projetos, os repositórios armazenam um grande volume de dados (*e.g.*, código, solicitações, comentários em redes sociais) que tem sido explorado com processos de Mineração de Dados para descobrir ou fortalecer padrões e conhecimento úteis para a ES.

Este capítulo apresenta alguns conceitos relacionados à GCS e à área de mineração de dados, com destaque especial para as tarefas preditivas. A Seção 2.2 aborda os fundamentos teóricos da GCS. A Seção 2.3 explica o processo que envolve o modelo de colaboração baseado em *pull requests* no desenvolvimento distribuído. A Seção 2.4 apresenta ferramentas que disponibilizam dados de repositórios *open-source*. Nas Seções 2.5, 2.6 e 2.7, são apresentados os principais conceitos da área de mineração de dados relacionados às

tarefas de classificação e regressão. Na Seção 2.8, são feitas as considerações finais.

2.2 Gerência de Configuração de Software

Na área de Engenharia de Software, controlar a evolução de software é um dos papéis da GCS [11]. A GCS é caracterizada por um conjunto de atividades que permite manter sob controle os Itens de Configuração (ICs) de um software ao longo do tempo como, por exemplo, arquivos de configuração, código fonte, diagramas e documentação [12]. As atividades da GCS têm como objetivo [13]: (i) identificar os ICs; (ii) controlar os ICs; (iii) armazenar as informações necessárias para gerenciar os ICs; (iv) avaliar e revisar os ICs; e (v) gerenciar a liberação e entrega do software.

Murta [14] afirma que a GCS pode ser dividida em três sistemas, que são: controle de modificações, controle de versões e gerenciamento de construção. Esta tese pretende utilizar as informações dos ICs armazenados pelos dois primeiros sistemas.

Os Sistemas de Controle de Modificações (*Issue Tracking System* – ITS) são encarregados de executar a função de controle da configuração de um software de forma sistemática. Nesses sistemas, as informações geradas pelas solicitações de modificações são armazenadas e relatadas aos participantes interessados e autorizados [14].

No ITS, quando defeitos são encontrados ou novas funcionalidades necessitam ser criadas, uma solicitação de modificação (*issue*) deve ser relatada, descrevendo a situação. A equipe de desenvolvimento, em seguida, verifica cada *issue* e estabelece os critérios de tratamento, ou seja, analisa e decide quais *issues* devem ser implementadas, a ordem em que isso vai ser realizado e qual desenvolvedor estará encarregado de cada tarefa [14, 15].

Em geral, as *issues* podem passar por três fases em um ITS [15]: (i) Solicitação, quando são preenchidos os dados que a caracterizam, como, por exemplo: o título, a descrição, o tipo de modificação, a versão e o componente do software que originou a solicitação de modificação; (ii) Modificação, quando a *issue* pode ser aceita para implementação ou não, dependendo da sua relevância e da qualidade das informações enviadas. Caso seja aceita, a *issue* deve ser tratada pelo desenvolvedor mais adequado para lidar com o defeito ou melhoria; e (iii) Verificação, que garante a correta implementação da *issue*. São exemplos de ITS: Bugzilla[1], Mantis[2], Redmine[3] e Trac[4].

[1] `http://www.bugzilla.org/`
[2] `http://www.mantisbt.org/`
[3] `http://www.redmine.org/`
[4] `http://trac.edgewall.org/`

Os Sistemas de Controle de Versões (SCVs) permitem que os ICs sejam identificados, conforme a função de identificação da configuração, e que evoluam de forma distribuída e concorrente, porém disciplinada. Tal característica é necessária para que as solicitações de modificação efetuadas possam ser tratadas em paralelo, sem corromper a GCS [14].

O SCV cria uma nova revisão sempre que um IC sofre alguma alteração. Assim, uma revisão é uma versão criada com o objetivo de substituir a sua antecessora [16]. Portanto, um IC evolui como um conjunto de revisões. Durante o processo de evolução, para cada nova revisão, o SCV armazena um *commit* que registra quem foi o desenvolvedor responsável pela criação da versão, quando a versão foi armazenada e uma breve descrição do conjunto de alterações.

Um SCV deve permitir que os desenvolvedores possam gerenciar diferentes linhas de desenvolvimento para promover a evolução paralela do software. Isto garante que, a partir de qualquer revisão, uma nova linha de desenvolvimento possa ser criada. Cada nova linha de desenvolvimento é chamada de ramo (*branch*) [12].

De forma geral, um SCV fornece acesso aos ICs por meio de políticas de controle de concorrência definidas conforme a necessidade da equipe de desenvolvimento. Murta [14] cita as duas principais: a pessimista, que evita o desenvolvimento paralelo por meio de bloqueios (*locks*), e a otimista, que permite o paralelismo e oferece mecanismos para a integração (*merge*) posterior do que foi feito concomitantemente [17].

Os SCVs que utilizam a política pessimista atendem melhor a situações específicas, onde os ICs não são textuais e a ferramenta que conhece a representação binária dos ICs não dá apoio automatizado para integração de código. Por outro lado, muitas situações no desenvolvimento de software necessitam de um ambiente de evolução paralela, onde a política otimista é mais indicada que a pessimista [18].

Quanto à arquitetura, um SCV pode ser classificado em: centralizado (SCVC) ou distribuído (SCVD). Os SCVCs (*e.g.*, Subversion[5], Perforce[6] e CVS[7]) controlam e armazenam os ICs em um único repositório. Nos SVCDs (*e.g.*, Git[8], Mercurial[9] e Bazaar[10]), qualquer desenvolvedor com permissões adequadas pode fazer uma cópia completa do repositório em sua máquina local, enviar suas contribuições para outros repositórios e, ao mesmo tempo, receber contribuições de outros repositórios [1].

[5] `http://subversion.tigris.org/`
[6] `http://www.perforce.com/`
[7] `http://www.nongnu.org/cvs/`
[8] `http://git-scm.com/`
[9] `https://www.mercurial-scm.org/`
[10] `http://wiki.bazaar.canonical.com/`

A principal característica de um SCVC é a existência de um único servidor central que contém todos os arquivos versionados. A partir desse servidor, os desenvolvedores podem resgatar (*check-out*) uma versão específica dos arquivos para sua máquina local. No seu espaço de trabalho, o desenvolvedor pode realizar as alterações e, para concluir, executar o comando *commit*, que reflete as alterações locais no repositório remoto. Caso alterações ocorram em paralelo, o desenvolvedor deve atualizar (*update*) o seu espaço de trabalho em relação ao repositório remoto. Logo após essa sincronização, eventuais conflitos devem ser resolvidos antes de realizar o *commit* [1]. Uma desvantagem do SCVC é não permitir que *commits* sejam realizados quando o desenvolvedor está sem acesso ao repositório remoto, por exemplo, por estar sem acesso à internet [19].

Já no SCVD, os desenvolvedores podem obter em sua máquina local uma cópia pública e completa de um repositório remoto a partir de um servidor com o comando *clone*. Com isso, cada máquina local torna-se um servidor em potencial e o repósitório pode ser copiado por outros desenvolvedores. Para compartilhar suas alterações, o desenvolvedor pode enviar as modificações realizadas localmente (*commits*) para o repositório remoto via comando *push*. Quando o repositório local não está atualizado em relação ao repositório remoto, é preciso garantir que todas as modificações feitas por outros desenvolvedores sejam propagadas em seu repositório local, o que é feito com o comando *pull* [1]. Em um SCVD, o desenvolvedor é forçado a trabalhar com a política otimista mesmo que a política pessimista seja mais indicada para o projeto [19].

Em projetos que utilizam SCVDs, as equipes de desenvolvimento costumam estabelecer fluxos de trabalho, denominados *workflows*, para organizar o processo de envio e recebimento das contribuições. Isso ocorre porque cada desenvolvedor pode contribuir com outros repositórios e, ao mesmo tempo, manter uma cópia pública do seu repositório, onde outros desenvolvedores também podem enviar contribuições. A depender dos objetivos do projeto, existem vantagens e desvantagens no uso de cada *workflow*, sendo possível adotar um ou combinar suas características e customizar um *workflow* específico para o projeto [1].

No *workflow* centralizado, também chamado de "repositório compartilhado" (*shared repository*), o modelo de colaboração conta com um repositório central (*hub*) que deve ser copiado (operação *clone*) em repositórios locais. Os desenvolvedores devem atualizar o seu espaço de trabalho local com o repositório central para pegar as alterações feitas em paralelo (operação *pull*). Em seguida, podem fazer as modificações no código, registrando os *commits* para depois enviá-los para o repositório central (operação *push*), conforme

ilustra a Figura 2.1. Se dois desenvolvedores trabalham em paralelo, o segundo a contribuir com o projeto deve fazer *merge* do seu trabalho com o do primeiro antes de enviar suas alterações para o repositório central [1].

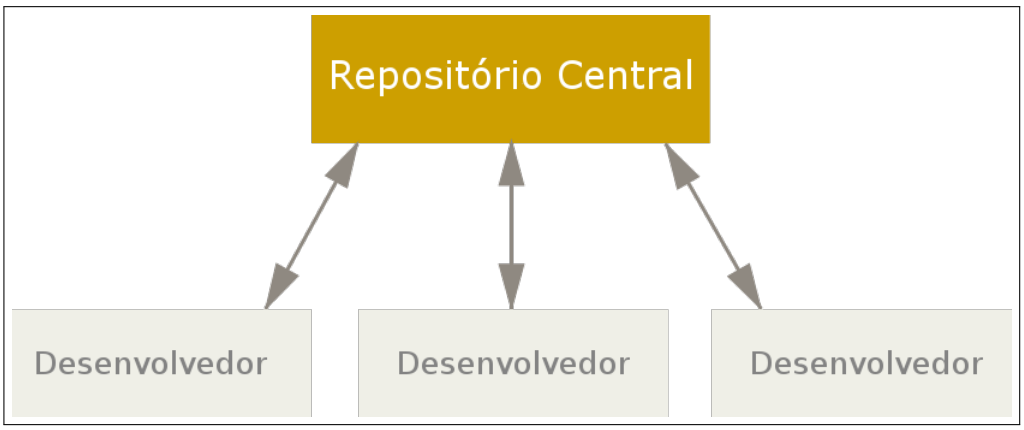

Figura 2.1: *Workflow* centralizado [1].

O "repositório compartilhado" contém ramos (*branches*) onde os colaboradores podem ler e escrever. Desse modo, os desenvolvedores podem modificar localmente o código, eventualmente introduzindo novos ramos, e fazer *push* das alterações de volta para o repositório central. Para lidar com várias versões e vários desenvolvedores, projetos maiores costumam adotar uma estratégia de ramificação, ou seja, uma forma organizada para inspecionar as contribuições antes de fazer a operação de *merge* com o ramo principal de desenvolvimento [20].

No *workflow* "gerente de integração" (*integration manager*), é possível ter múltiplos repositórios remotos. Assim, cada desenvolvedor possui acesso de escrita em seu repositório público e acesso de leitura nos demais repositórios. Geralmente, este cenário estabelece um repositório principal de referência para o projeto, onde todos têm permissão de leitura com exceção dos gerentes de integração, que também têm permissão de escrita. Para contribuir com esse projeto, o desenvolvedor cria uma réplica do repositório no próprio servidor (operação *fork*), onde passa a ter permissão de escrita. Em seguida, copia essa réplica para sua estação de trabalho (operação *clone*), podendo fazer as modificações desejadas localmente (operação *commit*) e enviar essas modificações para sua réplica do repositório no servidor (operação *push*) [1, 4].

A Figura 2.2 ilustra o processo do *workflow* gerente de integração, que funciona conforme os seis passos a seguir.

1. O desenvolvedor cria uma réplica pública (*fork*) do repositório principal.

2. O desenvolvedor faz uma cópia (*clone*) em sua máquina local, onde pode realizar suas alterações (*commit*).

3. O desenvolvedor envia as alterações para a sua réplica pública do repositório principal (*push*).

4. O desenvolvedor envia uma solicitação pedindo para um gerente de integração aceitar as alterações no repositório principal do projeto (*pull request*).

5. O gerente de integração adiciona a réplica pública do repositório principal que pertence ao desenvolvedor como um repositório remoto, faz *pull* desse repositório e depois *merge* das alterações localmente.

6. O gerente de integração faz *push* das alterações para o repositório principal.

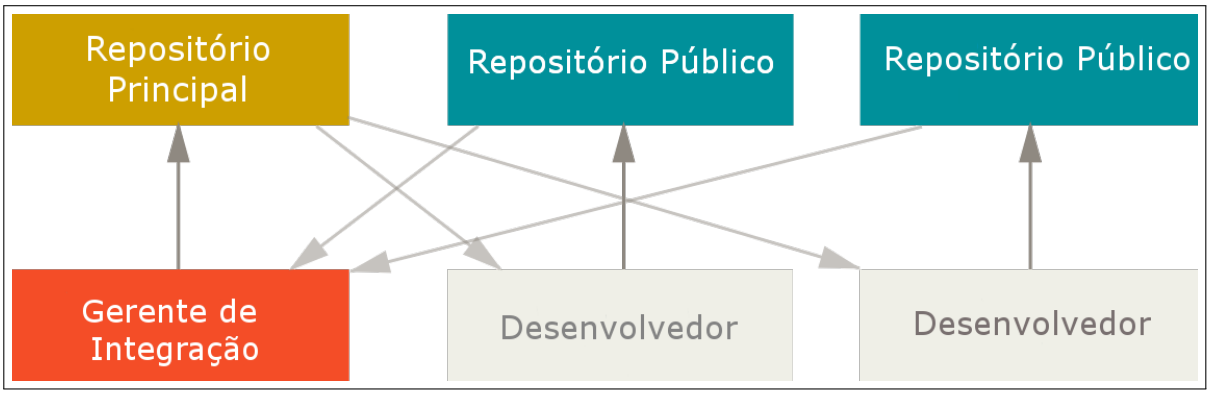

Figura 2.2: *Workflow* "gerente de integração" [1].

Este *workflow* é muito comum em projetos que compartilham seu código em plataformas de hospedagem como BitBucket[11] e GitHub, onde as funcionalidades de *fork*, *merge* e *push* nos repositórios são facilitadas pela interface dos sites. Uma das principais vantagens desta abordagem é que o desenvolvedor pode continuar trabalhando no repositório local, e o gerente de integração do repositório principal pode fazer o *merge* das alterações a qualquer momento. Assim, os desenvolvedores não precisam esperar a equipe do projeto aceitar o código para dar continuidade ao desenvolvimento do software [1].

2.3 *Pull Requests*

Em projetos que utilizam o *workflow* "gerente de integração", especialmente nos projetos *open source*, o processo de colaboração exige interações frequentes entre o desenvolvedor

[11] `https://bitbucket.org/`

que enviou a contribuição e a equipe principal do projeto, pois apesar de o projeto ser público, os desenvolvedores externos não têm permissão de escrita para atualizar diretamente o repositório principal [1].

Para contribuir com um projeto que adota o *workflow* "gerente de integração", o desenvolvedor envia uma solicitação denominada *pull request*. Em síntese, essa solicitação contém os arquivos de código fonte em um ramo do sistema de controle de versões para serem aceitos no repositório principal. Tal código fornece uma solução para a solicitação registrada pelo desenvolvedor [4].

O modelo de colaboração baseado em *pull requests* fornece um fluxo de trabalho que tem como objetivo facilitar o processo de colaboração de desenvolvedores externos. Entretanto, o *workflow* também pode ser utilizado por desenvolvedores internos, usualmente visando fomentar revisão de código. Segundo Gousios *et al.* [4], a metodologia de colaboração baseada em *pull requests* está presente em 14% dos repositórios do GitHub. Em projetos populares e que possuem uma comunidade ativa, uma grande quantidade de *pull requests* é recebida por mês [4].

Um *pull request* é constituído basicamente por uma solicitação associada a um ramo que contém vários *commits*. Cada *commit* é composto por um conjunto de alterações em código [4]. Em sua forma mais simples, *pull requests* são mecanismos para notificar a equipe principal que um desenvolvedor forneceu uma contribuição para o projeto. Isso permite que os envolvidos fiquem cientes da necessidade de revisar o código, para então decidir se o *pull request* será aceito ou rejeitado. Entretanto, os sites de hospedagem (*e.g.*, GitHub e BitBucket) oferecem junto a esse mecanismo um fórum dedicado a discutir as contribuições. Se houver qualquer problema, os membros da equipe principal do projeto podem enviar perguntas, comentários e até mesmo ajustar o código [21, 22].

O ciclo de vida de um *pull request* segue o *workflow* de "gerente de integração", descrito anteriormente (no fim da Seção 2.2). Após o envio do *pull request*, um membro da equipe do projeto (em geral um gerente de integração) precisa revisar e se responsabilizar por aceitar ou rejeitar o código. O *pull request* pode assumir três estados: aberto (*open*), indicando que o *pull request* ainda não foi revisado; aceito (*merged*), significando que o código foi aceito no repositório; e fechado (*closed*), quando a solicitação foi rejeitada. Durante a revisão, se o código é considerado insatisfatório, então um membro da equipe pode solicitar mudanças ou esclarecimentos por meio de comentários. Os colaboradores respondem aos comentários e opcionalmente atualizam o ramo com novos *commits* para que o *pull request* seja novamente avaliado [4].

Ao fim do processo, caso o *pull request* seja considerado satisfatório, o código pode ser aceito e integrado ao repositório principal por um dos membros da equipe. De acordo com Gousios *et al.* [4], os *pull requests* podem ser aceitos de três maneiras, vistas a seguir.

1. Através das ferramentas disponibilizadas pela interface do GitHub e do BitBucket. Caso não haja conflitos (automaticamente verificados), o código pode ser aceito no repositório, registrando o *merge* e mantendo todo o histórico do *pull request* (autor, código e ramo de origem).

2. Usando o comando *merge*. Quando um *pull request* não pode ser aceito pela interface web dos sites de hospedagem ou quando as políticas relacionadas ao projeto não permitem, a aceitação pode ser feita utilizando as seguintes técnicas.

   - *Branch merging*: o ramo que contém o código do *pull request* é aceito fora da linha principal do repositório. As alterações e as informações de autoria são mantidas, mas o GitHub, por exemplo, não consegue detectar o evento de *merge*.

   - *Cherry-picking*: consiste na escolha dos *commits* específicos do ramo remoto que serão aceitos no ramo principal. A história das alterações originais não pode ser mantida, mas o autor dos *commits* é preservado.

3. Aplicando *patch*. Quando apenas a diferença textual entre o código de um *pull request* e a linha principal (operação *diff*) é aplicada diretamente ao repositório. Nesse caso, as informações sobre a história e o autor do *pull request* são perdidas.

A Figura 2.3 ilustra a interface do GitHub com uma das formas de listagem dos *pull requests* no projeto rails. A lista apresenta o título, o número de identificação e o tempo de vida dos *pull requests* mais recentes no projeto (a). O GitHub oferece opções de filtros para facilitar a visualização das solicitações (*issues* e *pull requests*) por parte dos desenvolvedores (b), tais como: mostrar a lista de acordo com o estado (aberto ou integrado) da solicitação; exibir apenas *issues* ou *pull requests* enviados por um desenvolvedor; mostrar apenas as solicitações alocadas para um integrador; e mostrar *issues* ou *pull requests* em que o nome de um desenvolvedor foi mencionado nos comentários. A utilização dos filtros altera a lista e a quantidade de solicitações exibidas (c). A listagem também pode ser filtrada pelo autor da solicitação, pelos *Labels*, representados por rótulos que ajudam a identificar *issues* e *pull requests*, pelos *Milestones*, rótulos que costumam identificar versões do projeto, pelo *Assignee*, integrador alocado para o *pull request* e pelos revisores,

desenvolvedores que fizeram algum comentário. Além disso, a listagem pode ser ordenada pelas solicitações mais antigas, pelas mais recentes, pelo maior ou menor número de comentários e pelas solicitações que foram recentemente atualizadas (d).

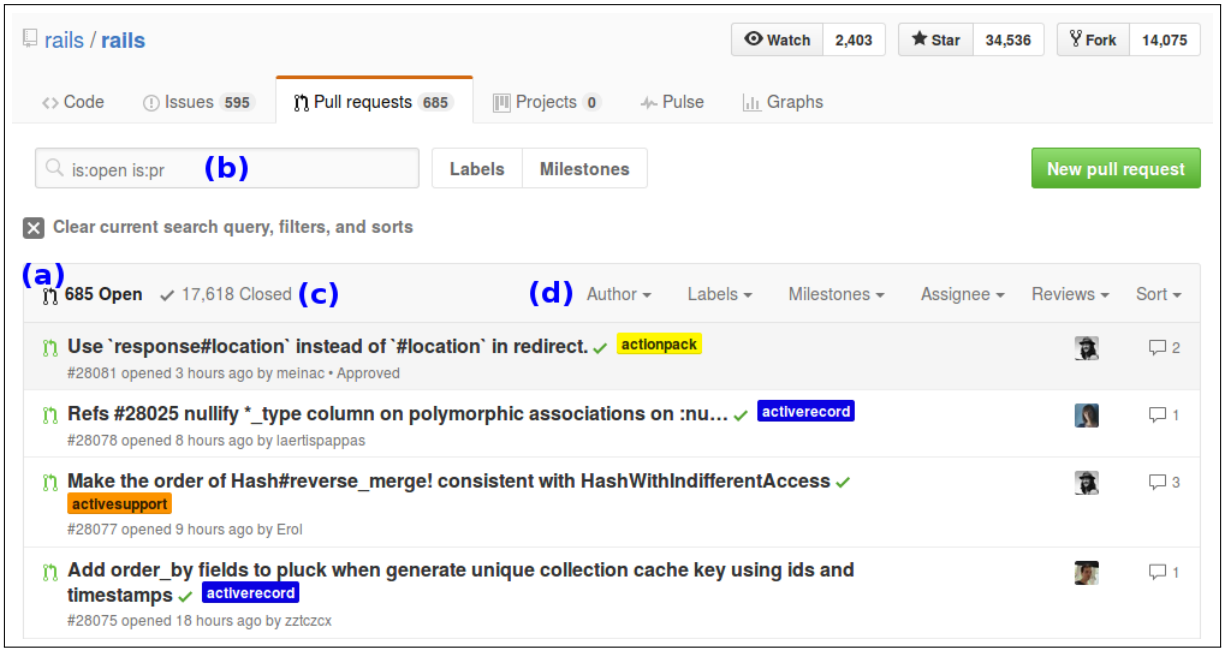

Figura 2.3: Listagem de *pull requests* abertos do projeto rails no Github.

Na parte superior direita da Figura 2.3, alguns números que indicam a popularidade de um projeto no GitHub podem ser visualizados como, por exemplo: a quantidade de desenvolvedores que desejam receber notificações quando novos *pull requests*, *issues* e *commits* são enviados para o projeto (*Watch*), quantos desenvolvedores cadastraram em seu perfil o projeto como um dos favoritos (*Star*) e a quantidade de desenvolvedores que fizeram uma cópia pública do repositório (*Fork*). Além de uma grande quantidade de *pull requests* recebidos e altos índices de popularidade, o projeto rails também possui muitos desenvolvedores em sua equipe principal.

2.4 Análise de Repositórios

Existem algumas ferramentas com o intuito de coletar informações sobre a evolução de projetos disponíveis no GitHub. Essas ferramentas viabilizam que pesquisadores de Engenharia de Software tenham acesso estruturado aos dados históricos dos projetos. O GHTorrent[12] é uma ferramenta que monitora eventos em repositórios públicos do GitHub. Para cada evento, a ferramenta recupera seu conteúdo e suas dependências, por exemplo,

[12] `http://ghtorrent.org`

commits, *issues* e *pull requests*. Em seguida, essas informações são enviadas para dois bancos de dados. No MongoDB são armazenados os dados brutos em formato BSON (*Binary JavaScript Object Notation*) e no MySQL são armazenadas informações no modelo relacional, utilizando-se SQL (*Structured Query Language*) [23].

O código fonte da ferramenta está disponível em um repositório público[13] no GitHub. Pesquisadores podem copiar o repositório e utilizar a GHTorrent para construir um banco de dados com informações de projetos *open source* como uma fonte de informações sobre o processo de desenvolvimento de software [24].

Embora o projeto GHTorrent ofereça versões para *download* das bases de dados com vários projetos *open souce* que estão hospedados no GitHub, a plataforma GHTorrent também oferece serviços de acesso direto aos sistemas de gerenciamento de banco de dados MongoDB e MySQL via cliente SSH (*Secure Shell*) e através do serviço Google BigQuery [25].

A ferramenta Github Archive[14] coleta e armazena o fluxo de eventos públicos do GitHub. Em seu banco de dados são registrados vários eventos como, por exemplo, os *commits*, *pull requests*, *issues*, *forks* e informações sobre a rede social de desenvolvedores do GitHub. O banco de dados completo está disponível publicamente para análise por meio do serviço Google BigQuery. Os dados são atualizados automaticamente a cada hora e podem ser acessados por consultas com sintaxe semelhante à SQL [26].

2.5 Mineração de Dados

Mineração de Dados (*Data Mining*) consiste na descoberta de informações e conhecimento úteis, na forma de regras e padrões, a partir de grandes bases de dados [27]. Trata-se de uma das etapas do processo chamado Descoberta de Conhecimento em Bancos de Dados (*Knowledge Discovery in Databases* − KDD), que é definido como um processo não trivial de identificação de padrões potencialmente úteis e compreensíveis [28]. Esse processo de descoberta de conhecimento é uma sequência iterativa das seguintes etapas [2]:

1. Limpeza dos dados, que consiste em remover ruídos e dados inconsistentes.

2. Integração de dados, quando múltiplas fontes de dados podem ser combinadas.

[13] `https://github.com/gousiosg/github-mirror`
[14] `https://www.githubarchive.org/`

3. Seleção de dados, na qual os dados relevantes para a tarefa de análise são selecionados a partir da base de dados integrada.

4. Transformação de dados, na qual os dados são transformados e consolidados em formas adequadas para a mineração.

5. Mineração de dados, quando são aplicados métodos para extrair padrões dos dados.

6. Avaliação, na qual identificam-se os padrões que representam o conhecimento verdadeiramente interessante, avaliados a partir de medidas de interesse.

7. Apresentação do conhecimento, quando técnicas de visualização e representação do conhecimento são utilizadas para apresentar o conhecimento extraído.

As etapas de 1 a 4 são fases de pré-processamento dos dados, nas quais os dados são preparados para a etapa de mineração. As fontes de dados podem incluir bancos de dados, *data warehouses*, a web e outras, tais como os repositórios da GCS [2]. As tarefas de mineração de dados são geralmente divididas em duas categorias principais [27]:

1. Tarefas Preditivas têm a finalidade de prever o valor de um atributo alvo (variável dependente) baseando-se nos valores de outros atributos (variáveis independentes). Existem dois tipos básicos de previsão: classificação, que é utilizada quando o atributo alvo é discreto, e regressão, usada para prever o valor de um atributo numérico. As duas modelagens trabalham para minimizar o erro entre os valores previsto e real da variável dependente.

2. Tarefas Descritivas têm o propósito de derivar padrões (*e.g.*, regras, correlações, tendências, grupos) que expliquem relacionamentos nos dados. São de natureza exploratória e frequentemente requerem técnicas de pós-processamento para validar e explicar os resultados. As principais tarefas descritivas são a extração de regras de associação e o agrupamento (*clustering*).

A tarefa de classificação é um processo sistemático para construção de modelos a partir de um conjunto de dados de entrada. Consiste em duas etapas: uma de aprendizagem e outra de classificação. Na primeira, um modelo de classificação é construído descrevendo um conjunto de dados de treinamento que contém o valor da classe (variável dependente), discreto e previamente conhecido. Nesta etapa, um algoritmo de classificação analisa as instâncias do conjunto de treinamento e seus rótulos de classe. Na segunda etapa, o modelo

é utilizado para prever o rótulo das classes em outro conjunto de instâncias, chamado conjunto de teste, no qual os valores da variável dependente não são apresentados [2, 27].

Quando a variável dependente é definida por valores numéricos, a tarefa de regressão utiliza as mesmas etapas da classificação para fazer a previsão. A tarefa de regressão consiste em encontrar uma função entre o conjunto de variáveis independentes e a variável dependente com o intuito de predizer um valor numérico para a variável dependente [29]. Na regressão linear, por exemplo, a ideia central é expressar o valor numérico para a váriável dependente como uma combinação linear das variáveis independentes [3]. Quando o modelo não segue um comportamento linear, a relação entre as variáveis pode ser representada por uma função polinomial [2, 27].

Por exemplo, prever o desenvolvedor apropriado para fazer a integração de um *pull request* é considerada uma tarefa de classificação, porque a variável dependente assume apenas valores discretos, ou seja, os nomes dos desenvolvedores da equipe de cada projeto. Por outro lado, a previsão do tempo de vida de um *pull request* pode ser explorada como uma tarefa de regressão, pois o tempo é um atributo que assume valores numéricos.

Para avaliar o desempenho de um modelo de classificação, é necessário definir uma estratégia para construir os conjuntos de treinamento e teste. Os principais métodos de avaliação de classificadores são [2, 30]:

- *Hold Out*: os dados são divididos aleatoriamente em dois grupos independentes. Normalmente, dois terços dos dados são alocados para o conjunto de treinamento e o restante para o conjunto de teste. O conjunto de treinamento é usado para derivar o modelo. A acurácia do modelo é estimada a partir do conjunto de teste, como sendo o número de elementos do conjunto de teste cujas classes foram corretamente preditas pelo classificador.

- *Random Subsampling*: é uma variação do *Hold Out*, na qual o método é executado k vezes. A acurácia geral é obtida pela média das acurácias de cada execução.

- *Cross-Validation*: a base é particionada (aleatoriamente) em k partições (aproximadamente de mesmo tamanho), a partir das quais o treinamento e teste são executados k vezes. Em cada execução, uma das partições é utilizada para teste e as demais para treinamento $(k-1)$, ou seja, todas as partições são utilizadas em algum momento para teste e $k-1$ vezes para treinamento. Neste caso, a acurácia é definida como o número de elementos da base cujas classes foram corretamente preditas pelo classificador.

Considerando um problema de classificação binária, em que a classe pode assumir apenas dois valores (Positivo e Negativo), a contagem dos elementos que são correta e incorretamente previstos pelo classificador é tabulada em uma matriz, chamada matriz de confusão, conforme pode ser visto na Figura 2.4, onde [2]:

- Verdadeiros Positivos (*True Positives* − *TP*): quantidade de elementos positivos corretamente classificados como positivos.
- Verdadeiros Negativos (*True Negatives* − *TN*): quantidade de elementos negativos corretamente classificados como negativos.
- Falsos Positivos (*False Positives* − *FP*): quantidade de elementos negativos incorretamente classificados como positivos.
- Falsos Negativos (*False Negatives* − *FN*): quantidade de elementos positivos incorretamente classificados como negativos.

		Classe Prevista	
		Positivo	Negativo
Classe Real	Positivo	*TP*	*FN*
	Negativo	*FP*	*TN*

Figura 2.4: Matriz de Confusão [2].

A partir da matriz de confusão é possível representar o total de elementos positivos (P) pela Equação 2.1 e negativos (N) pela Equação 2.2.

$$P = TP + FN \tag{2.1}$$

$$N = FP + TN \tag{2.2}$$

A partir dos conceitos anteriores, algumas métricas podem ser definidas para medir o desempenho do classificador, tais como [2]:

- Acurácia − representa a porcentagem de elementos do conjunto de teste que foram corretamente classificados:

$$\frac{TP + TN}{P + N} \tag{2.3}$$

- Cobertura (*Recall*) – representa a proporção de elementos positivos que foram classificados como positivos:

$$\frac{TP}{P} \tag{2.4}$$

- Precisão (*Precision*) – representa a proporção de elementos classificados como positivos que realmente são positivos:

$$\frac{TP}{TP\ +\ FP} \tag{2.5}$$

- *F-Score* ou *F-Measure* – representa a média harmônica entre *Precision* e *Recall*:

$$\frac{2\ \times\ Precision\ \times\ Recall}{Precision\ +\ Recall} \tag{2.6}$$

Considerando um problema de regressão, no que diz respeito à divisão da base em conjuntos de treinamento e teste, os métodos *Holdout*, *Random Subsampling* e *Cross-Validadtion* também podem ser utilizados. Para verificar o desempenho preditivo dos algoritmos de regressão, os valores reais e preditos de cada instância de teste são utilizados para calcular as medidas de qualidade do algoritmo. Neste trabalho, foram utilizadas as medidas RMSE (*Root Mean Squared Error*), NRMSE (*Normalized Root Mean Squared Error*) e o coeficiente de correlação de *Spearman*.

RMSE é uma medida para indicar o quanto os valores das variáveis observadas se distanciam do valor predito pelos modelos, em média. O valor de RMSE é representado na mesma unidade que o atributo alvo e definido pela Equação 2.7 [31].

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}\ , \tag{2.7}$$

onde n representa a quantidade de variáveis observadas, y_i representa o valor real da variável dependente e $\hat{y}_i$ indica o valor previsto pelo algoritmo de regressão.

NRMSE é uma medida baseada no cálculo de RMSE, em que a diferença entre o valor real e o previsto é dividida pelo valor real. Essa medida expressa uma porcentagem, o que a torna independente da unidade de medida da variável estudada. Isso pode ajudar a uma melhor interpretação dos resultados. A medida é definida na Equação 2.8.

$$NRMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2}\ , \tag{2.8}$$

onde n, y_i e $\hat{y}_i$ são as mesmas informações utilizadas no cálculo da medida RMSE.

O coeficiente de correlação de *Spearman* mede o grau de associação entre duas variáveis, que possuem valores em uma escala ordinal e podem ser dispostas em postos em duas séries ordenadas [32]. Esse coeficiente de correlação não exige que as variáveis tenham uma distribuição normal, pois utiliza apenas os postos dos valores observados. O valor do coeficiente pode variar entre $+1$, correlação positiva perfeita, e -1, correlação negativa perfeita entre os valores das duas variáveis, conforme definido na Equação 2.9.

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{(n^3 - n)} \ , \tag{2.9}$$

onde n representa a quantidade de variáveis observadas, d_i representa a diferença entre o posto do valor real (p_i) e o posto do valor previsto pelo algoritmo ($\hat{p}_i$). Para obter o valor dos postos, uma lista é gerada com os postos de cada variável, atribuindo o posto 1 para o menor valor e o posto n para o maior valor de cada variável. Se o coeficiente atingir o valor $+1$, isso significa que os valores previstos sempre crescem quando os valores reais crescem. Para o valor -1, o coeficiente indica que os valores previstos sempre diminuem quando os valores reais aumentam, ou vice-versa.

2.6 Algoritmos de Classificação

Esta seção apresenta uma descrição básica das técnicas que foram utilizadas nos experimentos de classificação: *Naive Bayes* [33], Árvore de Decisão [34], *Random Forest* [35], *Support Vector Machine* (SVM) [36] e *k-Nearest Neighbor* (k-NN) [37]. Esses métodos representam diferentes paradigmas e são estratégias tradicionalmente indicadas por apresentar bom desempenho em diferentes domínios de aplicação [9].

As definições a seguir serão utilizadas nas descrições dos algoritmos de classificação. Seja D uma base de dados composta pelo conjunto de atributos $(A_1, A_2, ..., A_n)$. Cada instância t de D é representada por um vetor de dimensão n: $t = (v_1, v_2, ..., v_n)$, que contém os valores dos n atributos. A base de dados D é dividida em dois conjuntos: base de treinamento (T_r) e base de teste (T_e). Dada uma instância $X = (x_1, x_2, ..., x_n)$ do conjunto T_e a ser classificada com uma das classes do conjunto $\{C_1, C_2, ..., C_m\}$, o algoritmo de classificação utiliza o seu paradigma de aprendizado para observar os valores dos atributos nas instâncias de T_r e indicar uma das classes C_i para a instância X.

Os classificadores bayesianos são técnicas estatísticas baseadas no Teorema de Bayes. O classificador ***Naive Bayes*** calcula a probabilidade de uma instância de entrada pertencer a cada uma das classes. Seja X uma instância com n atributos e C_i um valor

do atributo classe. Na tarefa de classificação, o *Naive Bayes* calcula a probabilidade *a posteriori* $P(C_i|X)$ − probabilidade de X ser da classe C_i considerando os valores dos atributos de X − para cada classe C_i. O classificador decide que X é da classe C_i se e somente se $P(C_i|X)$ for maior do que $P(C_j|X)$ para qualquer outra classe C_j, ou seja, X é da classe C_i se e somente se $P(C_i|X) > P(C_j|X)$, para todo $1 \leq j \leq m$, $j \neq i$.

A probabilidade *a posteriori* $P(C_i|X)$ é calculada a partir do Teorema de Bayes, conforme ilustrada na Equação 2.10.

$$P(C_i|X) = \frac{P(X|C_i) \times P(C_i)}{P(X)} \tag{2.10}$$

Como $P(X)$ é constante para todas as classes, basta maximizar o numerador da Equação 2.10, tornando suficiente calcular $P(X|C_i)$ e $P(C_i)$.

A probabilidade *a priori* da classe C_i é representada por $P(C_i)$ e estimada pela Equação 2.11, onde $|C_i, T_r|$ é o número de instâncias da classe Ci em T_r e $|T_r|$ é o número de instâncias de T_r.

$$P(C_i) = \frac{|C_i, T_r|}{|T_r|} \tag{2.11}$$

A implementação do *Naive Bayes* considera que os atributos são condicionalmente independentes dado o valor da classe [2]. Logo, $P(X|C_i)$ pode ser calculado conforme a Equação 2.12.

$$P(X|C_i) = P(x_1|C_i) \times P(x_2|C_i) \times ... \times P(x_n|C_i) \tag{2.12}$$

A indução de uma árvore de decisão como um modelo de classificação é uma técnica amplamente utilizada. Uma **Árvore de Decisão** é um modelo que representa o conhecimento contido no conjunto de treinamento. A representação do conhecimento em uma árvore é intuitivo e, geralmente, de fácil entendimento. As árvores são compostas por nós internos, rotulados por atributos, arestas, que representam os possíveis valores dos atributos, e folhas, que identificam as classes. Cada nó interno combinado com uma aresta representa uma condição a ser satisfeita. Cada caminho da raiz até uma folha, define uma conjunção de condições que corresponde a uma regra de classificação [2].

Ao construir uma árvore de decisão, cada nó representa um atributo que precisa ser escolhido de acordo com a sua capacidade de separar as classes. Uma medida comumente utilizada é chamada de Ganho de Informação (*Information Gain* − IG). Trata-se de uma heurística baseada no conceito de entropia, que minimiza o número de testes necessários para classificar uma instância. O ganho de informação determina quão bem um atributo

separa as instâncias do conjunto de treinamento de acordo com o valor da classe. Assim, o atributo com maior ganho de informação é escolhido como atributo para o nó corrente. O ganho de informação representa a redução na entropia causada por particionar as instâncias de acordo com os valores do respectivo atributo [2].

No entanto, cada algoritmo de indução de árvore de decisão pode seguir uma metodologia para escolher o atributo mais informativo [27]. Um algoritmo muito conhecido e que utiliza a medida IG para indução de árvores de decisão é o ID3 [38], que evoluiu para o também conhecido C4.5 [34].

Uma árvore de decisão para prever se um cliente representa um comprador de computador é ilustrada na Figura 2.5. Cada nó interno, destacado em negrito, combinado com uma aresta define uma característica do cliente a ser classificado. As folhas determinam a classificação do cliente ('Sim' indica que representa um comprador e 'Não' representa o contrário) [2].

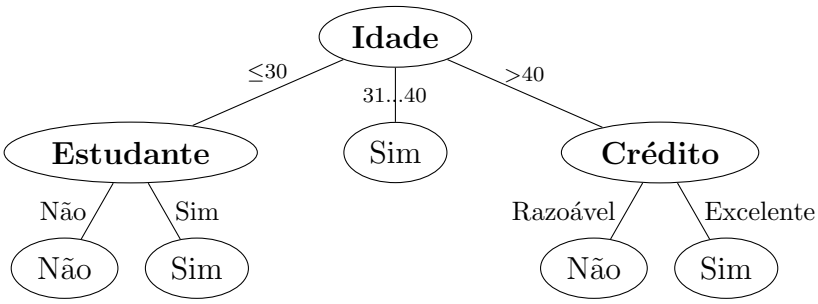

Figura 2.5: Exemplo de Árvore de Decisão [2].

Random Forest é um método de classificação do tipo *ensemble*, proposto por Breiman [35]. Um método *ensemble* combina um conjunto de k modelos de classificação, chamados de classificadores base $(M_1, M_2, ..., M_k)$. No *Random Forest*, cada classificador base é uma árvore de decisão construída a partir de um subconjunto distinto de treinamento. Para classificar uma instância, o método realiza uma votação majoritária entre as classes previstas por cada uma das árvores [2].

Seja T_r uma conjunto de treinamento. Cada classificador base M_i, $1 \geq i \geq k$, é treinado a partir de um subconjunto de T_r. Cada subconjunto é gerado por uma amostra aleatória e com reposição do conjunto de treinamento, chamada de *bootstrap*. Essa amostra é formada por 2/3 das instâncias. Dessa forma, algumas instâncias podem ser selecionadas mais de uma vez em cada subconjunto. Na escolha de cada nó das árvores de decisão, um subconjunto de atributos é selecionado aleatoriamente e o melhor atributo para o nó deve ser escolhido apenas entre os atributos desse subconjunto. Cada um dos classificadores vota uma vez para decidir qual classe será atribuída à instância a ser classificada [2].

O método de classificação denominado Máquina de Suporte Vetorial (***Support Vector Machine*** – SVM) tem por ideia básica encontrar uma fronteira de decisão, conhecida como hiperplano, com a melhor margem de separação de instâncias de duas classes distintas. Se as instâncias do conjunto de treinamento possuem duas classes linearmente separáveis, então existe um hiperplano que consegue separar corretamente todas as instâncias de treinamento. O hiperplano com máxima margem de separação, chamado de hiperplano ótimo, é o que proporciona a maior separação possível entre as classes [2, 39].

Caso as instâncias do conjunto de treinamento sejam linearmente separáveis, existem infinitos hiperplanos que podem separar as duas classes (C_1 e C_2). No entanto, como pode ser visto na Figura 2.6, o hiperplano em (b) oferece uma margem de separação maior entre as classes do que o hiperplano em (a). Isso indica que o hiperplano em (b) terá maior precisão na tarefa de classificar futuras instâncias [2].

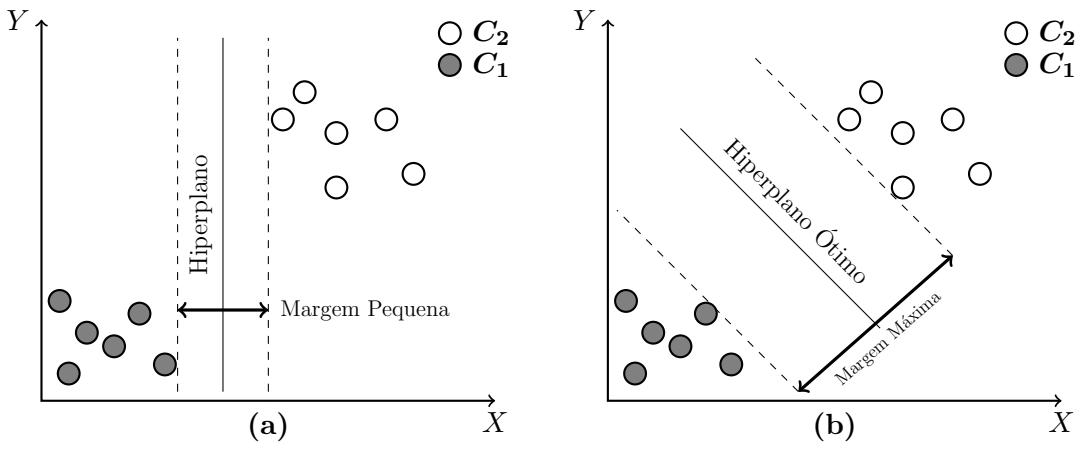

Figura 2.6: Margens e hiperplanos separadores [2].

O algoritmo SVM também é utilizado no caso em que as instâncias do conjunto de treinamento não são linearmente separáveis. Nesse caso, é feito um mapeamento não linear para transformar os dados de treinamento para outra dimensão, onde as instâncias passam a ser linearmente separáveis por um hiperplano [2].

Para fazer o mapeamento não linear no SVM é necessário utilizar uma função de *kernel* que será aplicada para transformar os dados. Alguns *kernels* conhecidos são:

- O *kernel* Polynomial [2],
- O *Pearson VII function-based universal kernel* (Puk) [40],
- O *kernel Radial Basis Funcioton* (RBF) [2].

Por exemplo, considere o conjunto de elementos $\{-2, -1, 0, 1, 2\}$. Na Figura 2.7, o lado (a) representa esses pontos não linearmente separáveis considerando as classes C_1 e C_2. O lado (b) representa o mapeamento para a nova dimensão ($\mathbb{R}^1 \rightarrow \mathbb{R}^2$) que torna os elementos linearmente separáveis por meio de um *kernel* $\varphi(X) = (X_i, X_i^2)$. Nessa nova dimensão, é possível encontrar o hiperplano ótimo que separa as instâncias das classes C_1 e C_2.

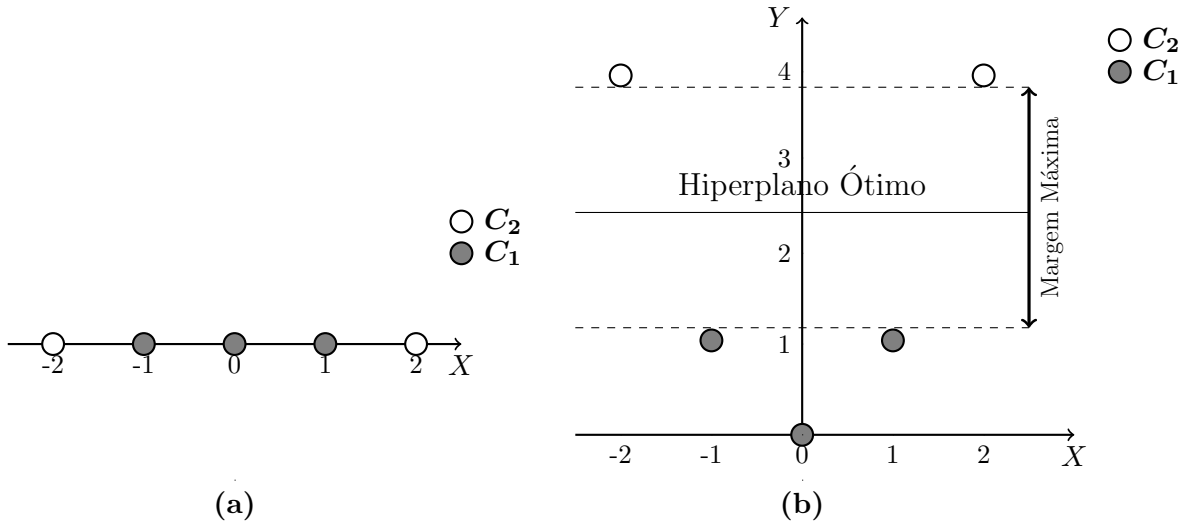

Figura 2.7: Mudança de dimensão com *kernel* no SVM.

O classificador ***k-Nearest Neighbor*** (k-NN) se baseia na ideia de aprendizado por analogia, ou seja, a classe de uma instância de entrada será determinada pelo conhecimento das classes de instâncias da conjunto de treinamento que são similares à instância de entrada. Cada instância possui n atributos e, portanto, pode ser caracterizada por um ponto em um espaço n-dimensional. A técnica procura pelas k instâncias de treinamento mais próximas, ou seja, mais semelhantes à instância a ser classificada nesse espaço n-dimensional. Essas instâncias são os k vizinhos mais próximos. Em seguida, o k-NN escolhe a classe predominante entre esses k vizinhos como a classe a ser atribuída à instância de entrada [2]. O valor de k é um parâmetro de entrada do algoritmo.

A proximidade (ou semelhança) entre as instâncias é definida por uma métrica de distância como, por exemplo, a distância Euclidiana. Quanto menor a distância, mais semelhantes são os elementos. Sejam $X_1 = (x_{11}, x_{12}, ..., x_{1n})$ e $X_2 = (x_{21}, x_{22}, ..., x_{2n})$ duas instâncias representadas no espaço n-dimensional. A distância entre X_1 e X_2 pode ser calculada pela Equação 2.13.

$$dist(X_1, X_2) = \sqrt{\sum_{1}^{n} (x_{1i} - x_{2i})^2} \tag{2.13}$$

O cálculo da distância exige que o valor dos atributos seja numérico. Por padrão, o algoritmo k-NN realiza uma normalização nos valores dos atributos no intervalo $[0, 1]$. O objetivo é evitar que atributos com valores muito diferentes, por exemplo, renda e idade, contribuam de formas distintas no cálculo da distância entre as instâncias [2]. Caso os atributos sejam categóricos (*e.g.*, cor), se os valores dos atributos são iguais (*e.g.*, as instâncias X_1 e X_2 são da cor azul), então a diferença é igual a 0 (zero). Se os valores são diferentes (*e.g.*, a instância X_1 é azul e a instância X_2 é vermelha), então a diferença é igual a 1 (um) [2].

2.7 Algoritmos de Regressão

Esta seção apresenta uma descrição básica das técnicas de regressão avaliadas nos experimentos: Regressão Linear [3], M5' [41], que implementa a descrição do algoritmo M5 de Árvores de Regressão inicialmente proposto por Quinlan [42], e o algoritmo SVM adaptado para previsão de valores numéricos [43].

O *Random Forest* também foi utilizado como algoritmo de regressão, pois o mesmo pode ser adaptado para construir árvores de regressão e realizar previsões quando a variável dependente contém valores numéricos. Nesse caso, o valor previsto é a média dos valores previstos por cada árvore.

A **Regressão Linear** consiste em encontrar uma equação linear que relaciona matematicamente os valores das variáveis independentes, também chamadas de variáveis explicativas ou atributos preditivos, com a variável dependente. Por exemplo, uma variável dependente y pode ser modelada como uma função linear de outras variáveis independentes, conforme a Equação 2.14 [2, 3].

$$y = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n, \tag{2.14}$$

onde y é o valor previsto para a variável dependente, x_1, x_2,..., x_n são os valores das variáveis independentes, e w_0, w_1, w_2,..., w_n são os pesos associados às n variáveis. Os pesos são calculados a partir das instâncias de treinamento pelo método dos mínimos quadrados, procurando minimizar o erro entre os valores reais e os valores previstos pela regressão.

Se a previsão envolve apenas uma variável dependente e outra independente é chamada de regressão linear simples. Quando a variável dependente se relaciona com mais de uma variável independente, denomina-se regressão linear múltipla. Assim, uma variável

dependente (y) pode ser modelada como uma função linear de duas ou mais variáveis independentes [2, 3].

As árvores utilizadas para previsão quando o atributo alvo assume valores numéricos são semelhantes às árvores de decisão. No entanto, as folhas podem armazenar: (i) um valor numérico que representa o valor médio do atributo alvo para instâncias que atingem aquela folha, nesse caso a árvore é chamada **Árvore de Regressão**; (ii) um modelo de regressão linear que prediz o valor do atributo alvo para as instâncias que atingem aquela folha, nesse caso a árvore é denominada **Árvore de Modelo**. Quando uma árvore de regressão ou de modelo é usado para prever o valor de uma instância de teste, a árvore é percorrida normalmente até uma folha, usando os valores dos atributos da instância para fazer decisões em cada nó [3].

A construção das árvores de regressão e de modelo utiliza o mesmo mecanismo de indução de árvore de decisão. No entanto, diferente dos algoritmos de árvore de decisão que utilizam o ganho de informação para escolher o atributo, nas árvores de regressão, para escolher o atributo que melhor divide as instâncias de treinamento, é utilizado um critério com base na Redução do Desvio Padrão (*Standard Deviation Reduction* − SDR). Para cada ciclo recursivo da indução, é escolhido o atributo que apresenta o maior valor de SDR de acordo com a Equação 2.15 [3, 42].

$$SDR \;=\; sd(T) - \sum_{i=1}^{n} \frac{|T_i|}{|T|} \times sd(T_i), \tag{2.15}$$

onde T é a porção do conjunto de treinamento que atinge o nó que está sendo construído, n é a quantidade de valores distintos do atributo escolhido, que define o número de partições que resultam da divisão de T, cada partição T_i é formada pelas instâncias de T que contêm um mesmo valor do atributo escolhido, $sd(T_i)$ é o desvio padrão para os valores do atributo alvo das instâncias de T_i e $sd(T)$ é o desvio padrão para os valores do atributo alvo das instâncias de T.

O processo de indução termina quando os valores do atributo alvo das instâncias que atingem o nó variam muito pouco, ou seja, quando seu desvio padrão é somente uma fração pequena (menos de 5%) do desvio padrão do conjunto original de instâncias. O processo também termina quando restam poucas instâncias (quatro ou menos) [3].

No caso das árvores de modelo, um modelo linear é calculado para cada nó interno da árvore, utilizando alguma técnica de regressão. Para construir cada modelo linear, em vez de todos os atributos, apenas os atributos que são testados nas subárvores pertencentes

ao nó corrente são utilizados. Após cada modelo linear ser obtido, ele é simplificado para minimizar o erro. Posteriormente, alguns testes são realizados para podar a árvore em um processo que retrocede a partir das folhas. Cada nó não terminal é examinado, iniciando dos nós próximo aos nós folha. O algoritmo pode selecionar como modelo final para cada nó testado, tanto o modelo linear simplificado calculado anteriormente, como o modelo das subárvores do nó sendo testado, dependendo de qual retorna um erro mais baixo. Se o modelo linear é selecionado para o nó, as subárvores desse nó são eliminadas [3, 42].

Uma árvore de modelo para prever o desempenho de um computador é apresentada na Figura 2.8. A árvore foi construída a partir da base de dados *Relative CPU Performance Data*[15] [44]. O atributo CHMIN, que representa a quantidade mínima de equipamentos de entrada e saída em unidades, foi escolhido como a raiz da árvore, produzindo um modelo linear na folha no ramo esquerdo e a estrutura restante no ramo direito. A árvore contém cinco folhas, destacadas em negrito. Cada uma representa um modelo linear. Por exemplo, para prever o valor do atributo alvo quando o valor do atributo CHMIN é $\leq 7,5$, o modelo linear localizado logo abaixo da folha LM1 deve ser utilizado. O primeiro número entre os parênteses de cada folha indica o número de instâncias que a alcançam. O segundo número é calculado como o erro médio (RMSE) das previsões a partir do modelo linear da folha para as instâncias de treinamento, dividido pelo desvio padrão dos valores do atributo alvo nas instâncias de treinamento e multiplicado por 100 [3].

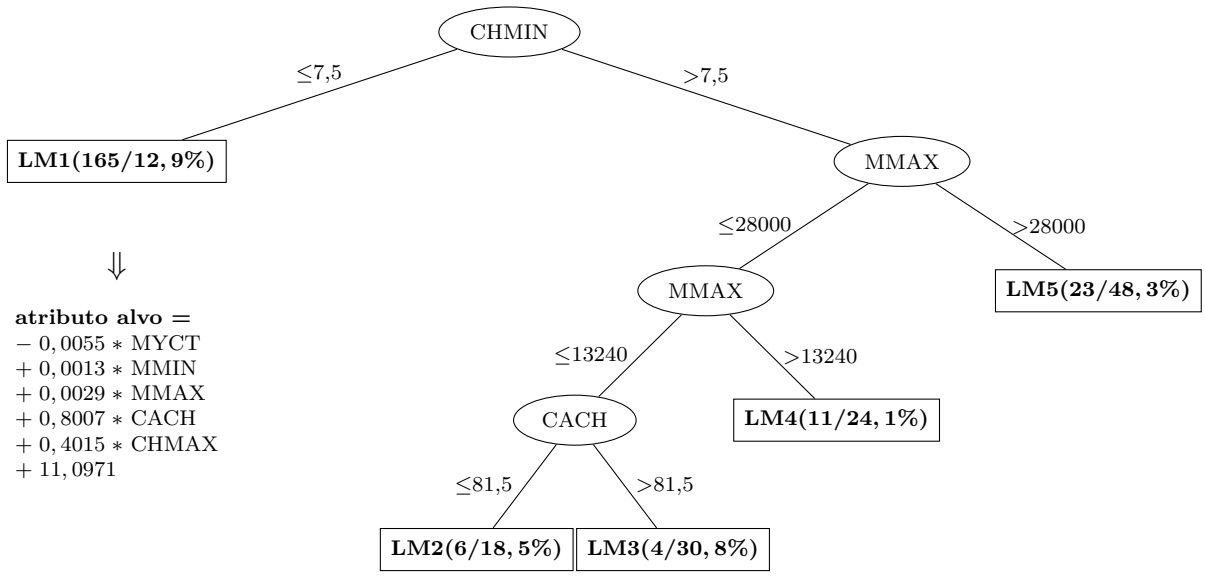

Figura 2.8: Exemplo de Árvore de Modelo [3].

[15]Os demais atributos da base são: CACH, que representa a memória cache em kilobytes (inteiro); CHMAX, que indica a quantidade de máxima de equipamentos de entrada e saída em unidades (inteiro); MMAX, que é a memória principal máxima em kilobytes (inteiro); MMIN, é a memória principal mínima em kilobytes (inteiro); e MYCT, representa o ciclo do computador em nanosegundos (inteiro).

O algoritmo SVM pode ser utilizado em tarefas de classificação e regressão. No primeiro caso, o algoritmo visa encontrar um hiperplano ótimo que maximize a margem de separação das instâncias das duas classes e, dessa forma, classificar corretamente novas instâncias. No segundo, o algoritmo busca uma função para prever os valores numéricos do atributo alvo.

Assim como na regressão linear, a ideia básica do **SVM para Regressão** é encontrar uma função linear que melhor representa os valores de treinamento do atributo alvo minimizando o erro da previsão. Essa função é construída considerando um desvio permitido ε em relação aos valores de treinamento, ou seja, o método tolera erros no intervalo $[-\varepsilon, +\varepsilon]$. Se todos os valores de treinamento produzem erros que estão dentro do intervalo permitido, a função é definida no meio desse intervalo e o erro total é igual a zero [3, 45].

A Figura 2.8 ilustra três soluções para um problema de regressão com oito instâncias. No primeiro caso (a), o valor de ε foi ajustado para 1, de modo que o desvio permitido em torno da função de regressão, indicado pelas linhas pontilhadas, é igual a 2. O segundo caso (b) mostra outra função de aprendizagem quando ε é definido como 2. Nos dois casos, o erro produzido é igual a zero, pois todos os valores estão dentro do intervalo permitido. No entanto, um valor muito alto para ε poderá produzir um modelo sem sentido. No caso extremo, quando ε exceder o intervalo de valores do atributo alvo nas instâncias de treinamento, a linha de regressão será horizontal e o algoritmo prediz apenas o valor médio dos valores extremos. Por outro lado, para valores muito baixos de ε, o desvio permitido pela função de regressão se torna muito penqueno e pode não envolver todas as instâncias de treinamento. No caso (c), a função é a mesma do caso (a), mas o valor de ε é igual a 0,5. Com isso, algumas instâncias ficam fora do intervalo permitido, gerando erro diferente de zero [3].

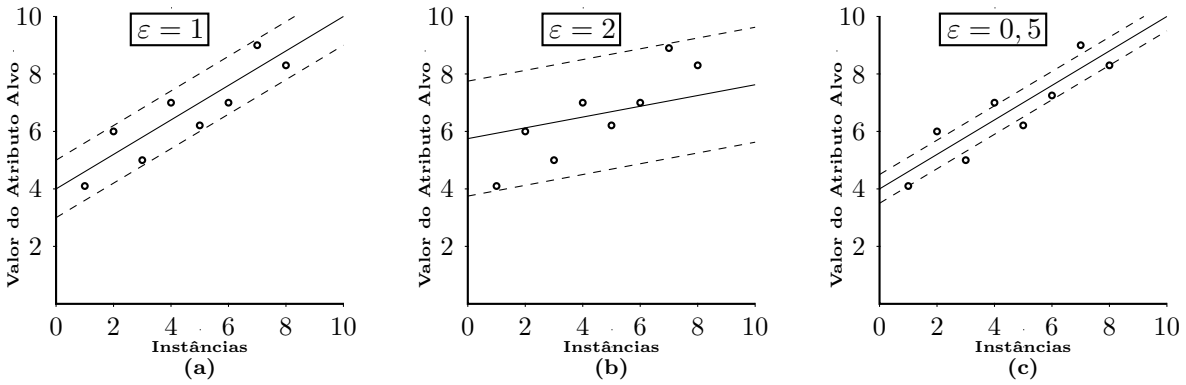

Figura 2.9: SVM para regressão [3].

2.8 Considerações Finais

Este capítulo apresentou os conceitos sobre dois sistemas (controle de versões e de modificações) que compõem a Gerência de Configuração de Software (GCS). No contexto de desenvolvimento distribuído de software, foram apresentadas as características de dois *workflows*: "repositório compartilhado" e "gerente de integração". No *workflow* "gerente de integração", também chamado de modelo *pull request*, a colaboração é baseada no envio de solicitações que fornecem o código necessário para atender à solicitação enviada.

O capítulo também abordou conceitos sobre as tarefas de mineração de dados, com destaque para as tarefas de classificação e regressão que podem ser utilizadas para realizar tarefas preditivas em repositórios de GCS como, por exemplo, prever os desenvolvedores mais apropriados para revisar ou integrar um *pull request* recebido e o tempo de vida de um novo *pull request*. Além disso, foram apresentados os conceitos sobre algoritmos preditivos de diferentes paradigmas para realizar a tarefa de classificação (*Naive Bayes*, Árvores de Decisão, *Random Forest*, SVM e k-NN) e de regressão (Regressão Linear, Árvores de Regressão, SVM para Regressão e *Random Forest*).

Capítulo 3

Trabalhos Relacionados

3.1 Introdução

Os repositórios de gerência de configuração de software armazenam uma grande quantidade de dados sobre a evolução de projetos de software. Ferramentas como GHTorrent [23] e GitHub Archive [26] recuperam informações de repositórios de projetos *open-source* hospedados no GitHub e as disponibilizam para *download*, consulta e análise. Vale destacar que bases de dados disponibilizadas com a ferramenta GHTorrent foram temas do desafio anual no *Mining Software Repositories* em 2014 e 2017 [24, 46].

O crescimento recente na utilização do modelo de colaboração baseado em *pull requests* e a disponibilização desses dados fornecem novas perspectivas e desafios para a descoberta de conhecimento no ambiente de desenvolvimento distribuído de software. Nos últimos quatro anos (2014 até 2017), esses fatores têm contribuído fortemente para o surgimento de vários estudos que exploram tarefas preditivas a partir de dados de *pull requests*. Alguns trabalhos tiveram como alvo da previsão a integração de *pull requests* [4, 47, 48]. A previsão de revisores para realizar comentários em *pull requests* foi explorada em [49, 50, 51, 52]. Outro trabalho avaliou a previsão dos integradores mais apropriados para decidir o estado final de *pull requests* [7] e a previsão do tempo de vida de *pull requests* foi abordada em [4, 8].

Neste capítulo, serão descritos os trabalhos relacionados à tarefa de previsão no contexto de *pull requests*. Uma visão geral sobre os trabalhos que tratam da integração de *pull requets* é apresentada na Seção 3.2. Nas Seções 3.3 e 3.4, são destacados os trabalhos que exploraram os cenários de previsão de desenvolvedores (revisores e integradores) e previsão do tempo de vida de *pull requests*. Por fim, a Seção 3.5 apresenta as considerações finais do capítulo.

3.2 Trabalhos sobre Integração de *Pull Requests*

O modelo de desenvolvimento baseado em *pull requests* difere do modelo de *issues*, pois *pull requests* já contêm o código fonte necessário para solucionar um defeito ou adicionar uma nova funcionalidade. Os *pull requests* fornecem informações como, por exemplo: descrição da solicitação, desenvolvedor solicitante, *commits*, número de linhas adicionadas e removidas, arquivos incluídos, alterados e removidos. Dessa forma, *pull requests* possuem informações tanto do sistema de controle de modificações, quanto do sistema de controle de versões do software. Adicionalmente, o GitHub também fornece dados sobre uma rede social de desenvolvedores que podem ser atribuídos para apoiar a revisão do código [53]. Nesse contexto, algoritmos de classificação podem utilizar esses dados para realizar previsões no processo de triagem de *pull requests* como, por exemplo, predizer se um *pull request* será aceito (*merged*) ou rejeitado (*closed*) pela equipe principal do projeto.

O trabalho de Gousios *et al.* [4] procurou entender o ciclo de vida dos *pull requests* analisando estatísticas descritivas e avaliando os fatores que influenciam nas tarefas de prever a integração de *pull requests*. Os experimentos foram realizados com 291 projetos e 166.884 *pull requests* extraídos pela ferramenta GHTorrent. Foram usados projetos escritos nas seguintes linguagens: Python (99 projetos), Java (91 projetos), Ruby (87 projetos) e Scala (14 projetos). Para avaliar o poder preditivo dos atributos explorados foram utilizados os seguintes algoritmos de classificação: *Naive Bayes* (NB), *Random Forest* (RF), *Support Vector Machine* (SVM) e *Logistic Regression* (LR). Após realizar uma validação cruzada (10-*fold cross validation*), a melhor acurácia foi obtida pelo algoritmo *Random Forest*: 86% para a decisão de integrar o *pull request*.

O estudo de Gousios *et al.* [4] concluiu que a presença de código de teste não afeta a decisão de integrar os *pull requests*. De acordo com os resultados, essa decisão é positivamente influenciada pelo fato de o *pull request* modificar um código que foi recentemente alterado no repositório. Por outro lado, em uma análise qualitativa de 350 *pull requests*, o estudo indica que 13% são rejeitados por questões técnicas e 27% pela dimensão tempo (ficou obsoleto, gerou conflito ou foi resolvido por outro *pull request*) [4].

No entanto, os resultados apresentados por Gousios *et al.* [4] foram obtidos a partir de uma única base, onde as instâncias representavam *pull requests* de projetos com características muito distintas. A base possuía desde projetos pequenos com poucos desenvolvedores e dezenas de *pull requests* até projetos grandes com muitos desenvolvedores

e milhares de *pull requests*. Além disso, esse estudo utilizou instâncias do futuro para predizer valores de instâncias do passado na avaliação dos classificadores. Esses fatores podem ter influenciado os resultados, revelando conclusões que não refletem a realidade de alguns projetos. Além disso, alguns atributos que podem ser extraídos a partir dos *pull requests* não foram explorados como, por exemplo, a localização dos arquivos alterados.

O trabalho de Tsay *et al.* [47] propôs um modelo estatístico que associa características sócio-técnicas dos solicitantes (*requesters*) com a probabilidade de integração de *pull requests* no desenvolvimento de software *open-source*. Os dados de 12.482 projetos e 659.501 *pull requests* foram extraídos com a ferramenta GitHub Archive. Os autores concluíram que os *pull requests* com vários comentários têm a probabilidade de aceitação (*merged*) reduzida em 54,6% a cada 6,656 comentários. Em relação aos fatores sociais, o estudo revelou que se o solicitante segue um desenvolvedor da equipe principal, a probabilidade de aceitar aumenta em 187,0%. Quanto maior o número de interações entre o solicitante e o projeto antes de enviar o *pull request*, como, por exemplo, a participação em *issues*, *pull requests*, e comentários no projeto, maior a probabilidade de aceitar. Nesses casos, para cada conjunto de oito interações, a probabilidade aumenta em 35,6%. Além disso, os solicitantes com muitos seguidores aumentam as chances de aceitar em 18,1%. Em geral, o aumento de algumas medidas associadas ao projeto, como popularidade, maturidade e quantidade de colaboradores, diminui a probabilidade de aceitação de *pull requests*.

Venn *et al.* [48] propuseram uma abordagem para selecionar *pull requests* que se encontram em estado aberto. Na abordagem, o tempo é dividido em janelas de um dia de duração. Em cada janela de tempo, a abordagem determina o valor de uma variável dependente que indica se o *pull request* recebeu uma ação dos desenvolvedores na próxima janela. Um algoritmo de classificação é treinado a partir de dados históricos para construir um modelo preditivo e prever se o *pull request* atual deve receber alguma atualização na próxima janela de tempo. Em 475 projetos *open-source*, o algoritmo *Random Forest* obteve 85% de acurácia na previsão de *pull requests* que deveriam receber a atenção dos desenvolvedores.

Estes trabalhos [4, 47, 48] exploraram problemas preditivos tendo como foco principal o cenário de integração dos *pull requests*. Entretanto, a decisão de aceitar ou rejeitar o código é uma das últimas atividades do ciclo de vida de um *pull request*. Assim, existem outros cenários preditivos no contexto de *pull requests* que podem ser explorados. Esta tese realizou estudos experimentais para investigar duas perspectivas de previsão. O primeiro estudo aplicou técnicas de classificação com o objetivo de prever os desenvolvedores mais

apropriados para integrar um novo *pull request*. O segundo estudo utilizou técnicas de regressão e classificação no intuito de predizer o tempo de vida de *pull requests*. Os trabalhos relacionados a esses dois estudos são discutidos nas seções a seguir.

3.3 Trabalhos sobre Previsão de Desenvolvedores

A atribuição de *issues* a desenvolvedores, também conhecida como triagem, diz respeito ao processo de encontrar o desenvolvedor mais adequado para lidar com uma determinada modificação no software. Em projetos que recebem grande quantidade de *issues* (*e.g.*, Mozilla e Eclipse), há muitas *issues* em aberto que precisam ser resolvidas, mas sem desenvolvedores alocados. Assim, existe um acúmulo de trabalho para os desenvolvedores, o que torna a triagem de *issues* uma atividade complexa [15, 54]. Em geral, essa atribuição requer tempo e conhecimento dos desenvolvedores, pois é realizada manualmente [55]. Baseados em dados de sistemas de controle de modificações (*e.g.*, Bugzilla e Mantis), existem trabalhos que buscam fazer previsões para facilitar a distribuição de *issues* entre os desenvolvedores de projetos *open-source* [56, 57, 58, 59].

Em projetos que recebem grande quantidade de *issues* (*e.g.*, Mozilla e Eclipse), há muitas *issues* em aberto que precisam ser resolvidas, mas sem desenvolvedores alocados. Assim, existe um acúmulo de trabalho para os desenvolvedores, o que torna a triagem de *issues* uma atividade complexa [15, 54].

No contexto do desenvolvimento baseado em *pull requets*, a integração do código enviado por um solicitante interno ou externo ao projeto depende da revisão e da decisão de um desenvolvedor da equipe principal do projeto (integrador). É desejável que o processo de revisão e integração do código enviado seja eficiente. Para isso, no momento em que um *pull request* é submetido ao projeto, algumas informações podem ser úteis na tomada de decisões, tais como, os desenvolvedores mais indicados para realizar comentários sobre a solicitação, chamados de revisores, e os desenvolvedores mais apropriados para concluir a revisão e decidir o estado final do *pull request*, denominados integradores. Em caso de aceitação, apenas os integradores do projeto possuem permissão para finalizar o processo de revisão de um *pull request*. Nos casos em que o código do *pull request* é rejeitado e não será integrado, o solicitante também pode encerrar o *pull request*.

Ferramentas de hospedagem de código, tais como o GitHub e BitBucket, oferecem funcionalidades de compartilhamento de código em conjunto com uma rede social que favorece a visualização e a participação dos desenvolvedores no processo que envolve a

revisão e integração de um *pull request*. Além disso, através do mecanismo de notificação dessas ferramentas, é possível notificar um desenvolvedor potencialmente interessado para participar das discussões de um *pull request*. No GitHub, a notificação consiste em colocar em um comentário o *login* do desenvolvedor precedido do símbolo @. A ferramenta cria um alerta para o desenvolvedor notificado, informando que seu nome foi citado por outro desenvolvedor em um comentário. Mecanismo semelhante também é utilizado pelos gerentes de projeto ao atribuir diretamente um desenvolvedor para integrar um *pull request*.

Qualquer desenvolvedor interno ou externo ao projeto pode acrescentar comentários (fazendo o papel de *reviewer*) e esses comentários podem fornecer informações importantes sobre o código alterado pelo *pull request*. Contudo, esses comentários não garantem, por exemplo, qual será o estado final de um *pull request*.

O problema de previsão de revisores no contexto de *pull requests* foi explorado em alguns trabalhos [49, 50, 51, 52, 60]. Em Yu *et al.* [49], com o objetivo de prever revisores, os autores aplicaram a técnica de recuperação da informação *Vector Space Model* (VSM) usando palavras dos títulos e das descrições dos *pull requests* para medir a similaridade entre um novo *pull request* e os anteriores. Em seguida, as informações sobre a quantidade de comentários de cada desenvolvedor em *pull requests* semelhantes foi utilizada na previsão de revisores apropriados para *pull requests*. Essa informação foi representada em uma rede de comentários (CN) e, com base nessa CN, sugere-se uma lista de revisores apropriados para realizar comentários sobre o *pull request*. O desempenho desta abordagem foi avaliado com um conjunto de dados contendo 10 projetos *open-source*. Os resultados obtiveram uma precisão média de 74% para a recomendação Top-1 e *recall* médio de 71% quando 10 revisores foram sugeridos. Neste trabalho, a previsão é considerada correta se o revisor sugerido tiver comentado pelo menos uma vez o *pull request*.

Em uma extensão da pesquisa apresentada em [49], Yu *et al.* [50] utilizaram o classificador SVM e a técnica de localização de arquivos proposta por Thongtanunam *et al.* [61] para melhorar o desempenho na previsão de revisores adequados para *pull requests*. Seus experimentos avaliaram 84 projetos *open-source* e um conjunto de dados com 105.123 *pull requests*, de 01/01/2012 a 31/05/2014. A avaliação foi realizada em um único modelo com 94% das instâncias no conjunto de treinamento e 6% das instâncias no conjunto de testes. O melhor desempenho foi obtido usando informações das duas abordagens (IR e CN): a recomendação Top-1 atingiu 67,3% de precisão e 25,2% de *recall*, enquanto a recomendação Top-10 obteve 33,8% de precisão e 79,0% de *recall*.

A abordagem proposta por Ying *et al.* [51] também explorou a previsão de potenciais revisores para *pull requests*. Os autores utilizaram dados já explorados por Yu *et al.* [49]: a similaridade das palavras do título e da descrição, e os comentários feitos nos *pull requests*. No entanto, a principal diferença foi o uso do algoritmo *Random Walk* para determinar a lista de revisores recomendados. Os experimentos foram realizados com um conjunto de dados de nove projetos *open-source*, onde 15.527 *pull requests* foram utilizados para treinamento e 1.156 para testes. Nos experimentos, a recomendação Top-1 atingiu 53,3% de precisão e 28,1% de *recall*, enquanto a Top-5 obteve 34,3% de precisão e 47,3% de *recall*, em média.

Rahman *et al.* [52] adaptaram a técnica de localização de arquivos para prever revisores adequados. Nessa adaptação, os dados sobre bibliotecas externas e tecnologias específicas foram extraídos do código-fonte enviado no *pull request*. Em seguida, bibliotecas e tecnologias utilizadas em *pull requests* abertos são comparadas com *pull requests* encerrados para calcular o valor de similaridade entre eles. Depois, esses valores de similaridade são atribuídos aos revisores que comentaram *pull requests* fechados pelo menos uma vez. Assim, a experiência de um revisor é determinada pela soma dos valores de similaridade de *pull requests* comentados por ele. Finalmente, uma lista dos revisores para comentar *pull requests* abertos, ordenados pela respectiva experiência, é recomendada. Nos experimentos, os autores utilizaram 10 projetos comerciais e seis projetos *open-source*, 10 bibliotecas, 10 tecnologias específicas e 17.115 *pull requests*. A recomendação Top-5 alcançou uma precisão que varia de 85% a 92%.

Em [60], Jiang *et al.* propuseram uma abordagem que analisa os *pull requests* anteriores e identifica os desenvolvedores que fizeram comentários nessas solicitações, com o objetivo de identificar possíveis revisores para um novo *pull request*. Os experimentos conduzidos empregaram quatro tipos diferentes de atributos extraídos a partir do *pull request* submetido e dos comentários realizados em *pull requests* anteriores. Para cada novo *pull requests*, os atributos são: (i) o nível de atividade dos revisores na realização de comentários em *pull requests* anteriores considerando uma janela de tempo (7 dias, 15 dias, 1 mês, 2 meses e sem limite de tempo); (ii) a semelhança das palavras utilizadas nos títulos dos *pull requests* anteriores com as palavras do título da nova solicitação; (iii) a semelhança entre os arquivos alterados por *pull requests* anteriores e arquivos alterados pela nova solicitação; e (iv) relações sociais entre os desenvolvedores. Os autores extraíram 19.543 *pull requests*, 206.664 comentários e 4.817 desenvolvedores que realizaram comentários (revisores) em oito projetos populares no GitHub. Para um novo *pull request*, a abordagem calcula uma pontuação para cada revisor no projeto com base nas solicitações de *pull*

request que cada revisor deixou comentários. Assim, uma lista com as pontuações mais altas é sugerida como os possíveis revisores para um *pull request*. O atributo baseado na atividade dos revisores, considerando a janela sem limite de tempo, obteve maior precisão e *recall* quando comparado com os outros atributos individualmente.

Esses trabalhos apresentados em [49, 50, 51, 52, 60] têm o objetivo de prever revisores para fazer comentários em *pull requests*. No entanto, uma pesquisa realizada com 749 integradores de projetos hospedados no GitHub revelou que a garantia da qualidade e o tempo consumido na revisão e integração são alguns dos principais desafios no que diz respeito ao desenvolvimento baseado em *pull requests* [6].

A qualidade do código a ser integrado no projeto é responsabilidade dos membros da equipe principal, especialmente nos casos em que o *pull request* altera arquivos importantes do projeto. Nesses casos, é absolutamente recomendável que um desenvolvedor com sólido conhecimento sobre o código alterado conduza o processo de integração. Portanto, um problema relevante neste cenário é prever quais integradores são mais adequados para a tarefa de decidir o estado final de um determinado *pull request*, ou seja, aqueles que encerram o processo e decidem se o código será aceito ou rejeitado. Caso os integradores sejam adequadamente sugeridos, o processo de integração pode ser realizado de forma mais eficiente.

Jiang *et al.* [7] propuseram a abordagem CoreDevRec com objetivo de prever integradores de *pull requests*. A proposta foi avaliada em cinco projetos (rails, scala, TrinityCore, xbmc e zf2). Top-5 integradores são recomendados para realizar a integração dos *pull requests* utilizando o classificador SVM. O conjunto de atributos preditivos usados na abordagem caracteriza os relacionamentos sociais, a localização dos arquivos alterados pelos *pull requests* e as atividades recentes dos membros da equipe principal.

Os experimentos foram conduzidos em um conjunto de dados com 18.651 *pull requests*, de 02/09/2010 a 07/31/2014. Na estratégia de avaliação, os *pull requests* foram divididos em partições, onde cada partição contém *pull requests* criados em um mês. Assim, em cada rodada, o classificador foi avaliado com instâncias de um mês para treinamento e do mês seguinte para teste, respeitando a ordem cronológica da criação dos *pull requests*, ou seja, somente informações anteriores foram utilizadas para a previsão dos integradores. Nessa estratégia, cada rodada depende do número de solicitações recebidas em cada projeto por mês. Uma vez que essa quantidade não é constante, as rodadas de teste são influenciadas pela quantidade de instâncias. A acurácia final da abordagem foi calculada pela média das acurácias nas rodadas de teste. Os resultados mostraram que a abordagem CoreDevRec

atingiu acurácias que variaram de 49,3% a 72,3% para a recomendação Top-1 e de 72,9% até 93,5% para a recomendação Top-3. A acurácia média foi de 57,0% e 82,2% para as recomendações Top-1 e Top-3, respectivamente [7].

É importante destacar que Jiang *et al.* [7] consideraram os desenvolvedores que aceitaram pelo menos um *pull request* no projeto como os integrantes da equipe principal. No entanto, podem existir desenvolvedores que têm permissão para aceitar *pull requests*, ou seja, aqueles que têm permissão de escrita no repositório principal do projeto e que poderiam ser recomendados, mas que ainda não aceitaram um *pull request* no momento em que o novo *pull request* foi recebido pelo projeto.

Na tarefa de prever integradores de *pull requests*, Jiang *et al.* [7] utilizaram atributos que contêm, para cada solicitação: (i) relações sociais entre o solicitante e os integradores, tais como, o número de *pull requests* enviados anteriormente pelo solicitante e integrados por cada integrador; e (ii) características que revelam o nível de atividade dos integradores no processo de integração de *pull requests* dentro de cada projeto, como, por exemplo, o número de *pull requests* integrados por cada integrador nos últimos 30 dias e o intervalo de tempo entre a submissão do *pull request* e a última integração de cada integrador. No entanto, a proposta anterior deixou de considerar vários outros atributos preditivos: (i) informações sobre o tamanho e a complexidade da solicitação como, por exemplo, o número de *commits*, arquivos e linhas; (ii) outras características sobre a experiência dos solicitantes, por exemplo, a sua taxa de aceitação e rejeição das solicitações; e (iii) informações fornecidas pelas relações sociais como, por exemplo, o número de desenvolvedores que seguem o solicitante e se o solicitante segue algum integrador.

3.4 Trabalhos sobre Tempo de Vida de *Pull Requests*

Uma tarefa importante no planejamento de projetos de software é prever o esforço e o tempo necessário para a correção de *bugs*. Essa estimativa pode melhorar o processo de triagem de *bugs*, permitindo, por exemplo, alocar o tempo de desenvolvedores mais experientes para correção de *bugs* de alta prioridade. Baseados em dados de sistemas de controle de modificações, existem trabalhos na literatura que exploraram a previsão do tempo de resolução de *bugs* [62, 63, 64].

O modelo baseado em *pull requests* e as modernas plataformas de desenvolvimento, tais como GitHub e BitBucket, oferecem várias funcionalidades para automatizar o processo de contribuição no desenvolvimento distribuído de software, incluindo testes auto-

matizados, fóruns de discussões, integração automática do código e recursos de rede social entre os desenvolvedores. Projetos *open-source* com grandes equipes (*e.g.*, rails, rosdistro, xbmc e docker) costumam adotar o modelo de *pull requests* para controlar as solicitações de mudanças e disponibilizam o código do projeto no GitHub. Nesses projetos, o volume de *pull requests* recebidos é bastante grande [4]. Uma das consequências é o aumento da carga de trabalho para os membros das equipes principais dos projetos que revisam e decidem sobre a integração do código no ramo principal de desenvolvimento. Assim, o tempo de revisão e integração de *pull requests* e a manutenção da qualidade do software se tornaram as principais preocupações dos integradores [6].

A integração de um *pull request* pode se tornar uma tarefa de alto custo, por diferentes motivos: código extenso, alteração de funcionalidades importantes e desconhecimento do desenvolvedor sobre os arquivos alterados pelo *pull request* [24]. Nesse contexto, prever o tempo necessário para o tratamento de *pull requests* pode ajudar no processo de triagem, permitindo por exemplo, alocar o tempo de desenvolvedores mais experientes para *pull requests* com alta prioridade. Baseado em dados de projetos armazenados no GitHub, algumas pesquisas recentes disponíveis na literatura já abordaram a previsão do tempo de vida (*lifetime*) de *pull requests* [4, 8].

Gousios *et al.* [4] avaliaram algoritmos de classificação em 141.468 *pull requests* de 291 projetos *open-source* para prever o tempo de vida, que foi discretizado em três classes ($C_1 \leq$ 1 hora $< C_2 \leq$ 1 dia $< C_3$). Considerando os projetos explorados, em média, um *pull request* demora 3,7 dias para ser encerrado. Nos experimentos, foram utilizados 12 atributos preditivos e um método de validação cruzada para aferir acurácias de 38% para o algoritmo *Naive Bayes* e 59% com o algoritmo *Random Forest*. Ao avaliar a importância dos atributos para o algoritmo *Random Forest*, em uma base de 83.442 *pull requests*, os autores afirmaram que a experiência do desenvolvedor, o tamanho do projeto, a presença de arquivos de testes, e a porcentagem de *commits* realizados a partir de *pull requests* no projeto parecem desempenhar um papel importante na qualidade da previsão do tempo de vida de *pull requests*. Além disso, os *pull requests* enviados por desenvolvedores da equipe principal não têm prioridade sobre os *pull requests* enviados por desenvolvedores externos. Por fim, sugerem que o tempo que um *pull request* leva para ser aceito é inversamente proporcional à taxa de *merge* do solicitante, ou seja, quanto mais alta a sua porcentagem de *pull requests* aceitos, em menos tempo um novo *pull request* será integrado [4].

Yu *et al.* [8] exploraram a influência de três conjuntos de atributos na latência de *pull requests* no GitHub. Em 103.284 *pull requests* extraídos de uma amostra de 40 pro-

jetos *open-source* que usam serviço de integração contínua[1], a técnica de regressão linear múltipla foi utilizada para modelar a latência da avaliação de *pull requests*. Três modelos lineares foram construídos: o primeiro apenas com atributos previamente utilizados na literatura [4, 47]; o segundo adicionando atributos relacionados ao processo de revisão e integração dos *pull requests*; e o terceiro acrescentando atributos sobre a utilização de integração contínua no processo.

O terceiro modelo, que contém a maior quantidade de atributos, incluindo os novos e aqueles já utilizados em trabalhos anteriores, obteve o melhor coeficiente de determinação ($R^2 = 0,587$). A partir dos modelos apresentados, os autores afirmam que: (i) a latência de *pull requests* é uma questão complexa, que exige muitas variáveis independentes para ser explicada adequadamente; (ii) a presença de integração contínua é um forte preditor positivo; (iii) o número de comentários é o melhor atributo preditivo; e (iv) o tamanho do *pull request* (linhas adicionadas, número de *commits*) é um forte atributo para determinar a latência de *pull requests* [8].

Mesmo com a existência de trabalhos que exploraram a previsão do tempo de vida de *pull requests*, esses trabalhos se diferenciam em vários aspectos referentes aos materiais e métodos utilizados: conjuntos de atributos preditivos, técnicas de classificação e regressão, processos experimentais e quantidade de projetos.

3.5 Considerações Finais

Este capítulo discutiu os trabalhos disponíveis na literatura que exploraram problemas preditivos em sistemas de controle de modificações com destaque para as previsões de integração, de desenvolvedores e tempo de vida de *pull requests*.

No cenário de recomendação de integradores de *pull requests*, a proposta de Jiang *et al.* [7] deixou de considerar vários atributos preditivos importantes. No cenário de previsão do tempo de vida de *pull requests*, os trabalhos anteriores [4, 8] utilizaram diferentes conjuntos de atributos preditivos, técnicas preditivas, processos experimentais e quantidade de projetos. Nesse sentido, esta tese visa contribuir com o estado da arte melhorando o desempenho das previsões.

[1] Quando um *pull request* é recebido em um projeto que utiliza integração contínua, o código é integrado em um ramo de teste e os conjuntos de testes são executados. Em caso de falha nos testes, um integrador pode rejeitar o *pull request* ou solicitar melhorias no código informando ao solicitante o que está inapropriado. Se os testes forem aprovados, um integrador pode revisar o código e decidir sobre a integração [8]. Um dos serviços de integração contínua mais utilizados é o TravisCI (`https://travis-ci.org/`).

Os resultados e as discussões sobre os experimentos no cenário de recomendação de integradores estão detalhados no Capítulo 4 e os experimentos no cenário de previsão do tempo de vida de *pull requests* estão reportados no Capítulo 5.

Capítulo 4

Previsão de Integradores de Pull Requests

4.1 Introdução

Neste capítulo, é apresentada a contribuição desta tese no cenário de previsão de integradores de *pull requests*.

Nesse cenário de previsão, propusemos em um estudo preliminar [65] a utilização de algoritmos de classificação a fim de sugerir os desenvolvedores apropriados para determinar seu estado final de *pull requests*. A tarefa de previsão foi explorada com 48.510 *pull requests* e 14 atributos preditivos extraídos através da ferramenta GHTorrent a partir de informações fornecidas pelos *pull requests* e pelas relações sociais entre os desenvolvedores. Em 21 projetos *open-source*, os classificadores foram avaliados com o método de validação cruzada (10-*fold cross validation*). A melhor acurácia foi alcançada pelo algoritmo *Random Forest* com um valor médio de 47,81% e 77,52% para as recomendações Top-1 e Top-3. Ao observar a diferença entre as acurácias obtidas pela proposta e por um *baseline*, as melhorias foram de 61% e 21% para as recomendações Top-1 e Top-3, em média.

Os experimentos foram ampliados com novos atributos preditivos e uma maior quantidade de projetos (106.168 *pull requests* obtidos de 32 projetos *open-source*), além da utilização de uma nova metodologia de avaliação de classificadores. O algoritmo *Random Forest*, após a aplicação de técnicas de discretização e estratégias de seleção de atributos, conseguiu uma acurácia média de 49,17%, 77,99%, e 88,38% para as recomendações Top-1, Top-3 e Top-5, respectivamente. Em comparação com estudos anteriores, realizada sobre a mesma base de dados (*pull requests* e projetos), visando uma comparação justa,

nossa abordagem obteve maiores acurácias e, consequentemente, maiores ganhos normalizados nos três tipos de recomendações. Para a recomendação Top-1, as propostas dos dois estudos anteriores [7, 65] alcançaram média de ganho normalizado de 12,75% e 12,90%, enquanto a nossa abordagem alcançou ganho normalizado de 19,93%. Isso representa um ganho de mais de 54%. Além disso, ganhos de mais de 32% e 24% também foram observados ao comparar a nossa abordagem com os resultados dos trabalhos anteriores para as recomendações Top-3 e Top-5, respectivamente. Vale notar que os nossos primeiros resultados [65] e os resultados apresentados por Jiang *et al.* [7] surgiram concomitantemente, em 2015.

O objetivo principal deste estudo foi investigar se a escolha de um conjunto adequado de atributos pode melhorar o desempenho da tarefa de previsão de integradores. Para conduzir essa investigação, diferentes conjuntos de atributos foram avaliados com diferentes algoritmos de classificação. Além disso, estratégias de seleção de atributos foram utilizadas em um extenso conjunto de atributos composto não apenas de atributos que já foram utilizados na literatura, mas também de novos atributos considerados significativos e úteis para previsão de integradores.

Uma contribuição secundária deste trabalho é a concepção e adoção de um método de avaliação de classificadores para o cenário de *pull requests*. Normalmente, a avaliação dos algoritmos de classificação para determinar suas acurácias é realizada utilizando o método de validação cruzada (*k-fold cross validation*) [2]. No entanto, esse método não é adequado para conjuntos de dados com *pull requests*, pois esses dados seguem a ordem cronológica de submissão dos *pull requests* nos repositórios. O problema é que, utilizando o método de validação cruzada, as instâncias do futuro podem ser utilizadas para prever o passado. Para resolver esse problema, propusemos um método para avaliar classificadores em conjuntos de dados com dependência temporal.

Este capítulo está dividido da seguinte forma. A Seção 4.2 descreve os dados, atributos e projetos utilizados nos experimentos. Os resultados e análises dos experimentos são detalhados na Seção 4.3. A Seção 4.4 discute as limitações e ameaças à validade dos resultados obtidos. Por fim, a Seção 4.5 apresenta as considerações finais sobre os experimentos.

4.2 Materiais e Métodos

Neste estudo experimental sobre previsão de integradores, os classificadores foram avaliados em três diferentes conjuntos de atributos preditivos (Conjuntos A, B e C). A Subseção

4.2.1 mostra, para cada conjunto, a descrição dos atributos utilizados. A Subseção 4.2.2 descreve os dados utilizados e as características dos projetos selecionados. A Subseção 4.2.3 informa quais algoritmos de classificação, técnicas de discretização e estratégias de seleção de atributos foram utilizados durante o processo experimental. Por fim, a Subseção 4.2.4 detalha como os experimentos foram conduzidos.

4.2.1 Conjuntos de Atributos

Nesta seção, são apresentados três conjuntos de atributos que foram utilizados para comparar o desempenho dos classificadores. O Conjunto A contém os atributos utilizados em nosso trabalho preliminar [65], o Conjunto B possui os atributos explorados por Jiang *et al.* [7] e o Conjunto C é composto pelos atributos dos conjuntos anteriores e por novos atributos, que caracterizam uma contribuição desta tese.

A Tabela 4.1 mostra o identificador, a categoria, o tipo, o nome e a descrição de cada atributo do Conjunto A. Os atributos estão organizados em três categorias: *Pull Request*, Solicitante e Social. Os atributos da categoria *Pull Request* representam informações sobre o tamanho e a complexidade das contribuições, tais como, quantidade de *commits* ($A.1$), de linhas adicionadas ($A.2$), de linhas excluídas ($A.3$) e de arquivos alterados ($A.4$).

Tabela 4.1: Atributos do Conjunto A.

Id	Categoria	Tipo	Nome	Descrição
$A.1$	*Pull Request*	Numérico	NumCommits	Quantidade de *commits* no pull request
$A.2$	*Pull Request*	Numérico	NumAddedLines	Quantidade de linhas de código adicionadas pelo *pull request*
$A.3$	*Pull Request*	Numérico	NumDeletedLines	Quantidade de linhas de código excluídas pelo *pull request*
$A.4$	*Pull Request*	Numérico	NumChangedFiles	Quantidade de arquivos alterados pelo *pull request*
$A.5$	Solicitante	Discreto	RequesterType	Tipo de solicitante (interno ou externo)
$A.6$	Solicitante	Numérico	RequesterNumPRs	Quantidade de *pull requests* enviados pelo solicitante
$A.7$	Solicitante	Numérico	RequesterMergedPRs	Quantidade de *pull requests* aceitos do solicitante
$A.8$	Solicitante	Numérico	RequesterRejectedPRs	Quantidade de *pull requests* rejeitados do solicitante
$A.9$	Solicitante	Numérico	RequesterAcceptRate	Taxa de aceitação dos *pull requests* enviados pelo solicitante
$A.10$	Solicitante	Numérico	RequesterRejectRate	Taxa de rejeição dos *pull requests* enviados pelo solicitante
$A.11$	Social	Numérico	RequesterNumFollowers	Quantidade de seguidores do solicitante
$A.12$	Social	Numérico	RequesterNumFollowing	Quantidade de desenvolvedores seguidos pelo solicitante
$A.13$	Social	Binário	RequesterFollowCT	Se o solicitante é seguidor de um desenvolvedor da equipe
$A.14$	Social	Binário	RequesterWatches	Se o solicitante é um observador do projeto no GitHub

Na categoria Solicitante, o atributo RequesterType indica se o solicitante é um membro da equipe principal do projeto. Os outros atributos dessa categoria são calculados no momento da submissão do *pull request* a partir das solicitações anteriores: a quantidade de *pull requests* submetidos pelo solicitante ($A.6$), a quantidade de *pull requests* aceitos do

solicitante ($A.7$), a quantidade de *pull requests* rejeitados do solicitante ($A.8$), e a taxa de aceitação ($A.9$) e rejeição ($A.10$) de *pull requests* do solicitante. Esses atributos representam a experiência dos solicitantes na utilização do desenvolvimento baseado em *pull requests*.

Os atributos do Conjunto A que envolvem relações sociais entre os desenvolvedores no GitHub estão representados na categoria Social: a quantidade de desenvolvedores que seguem o solicitante ($A.11$), a quantidade de desenvolvedores que o solicitante é seguidor ($A.12$), se o solicitante é seguido por algum integrador ($A.13$) e se o solicitante é um observador do repositório do projeto ($A.14$). Esses relacionamentos revelam informações sobre o interesse do solicitante nas atividades do projeto, o *status* social dos desenvolvedores e as relações sociais diretas entre solicitantes e integradores. Essas informações podem apoiar a tarefa de prever integradores de *pull requests*.

A Tabela 4.2 contém o identificador, a categoria, o tipo, o nome e a descrição dos atributos propostos por Jiang *et al.* [7] e representados pelo Conjunto B. Os atributos estão organizados em quatro categorias: Projeto, Social, Integrador e *Pull Request*. O primeiro atributo, único da categoria Projeto, representa o número de *pull requests* submetidos ao projeto no momento da submissão do *pull request* corrente ($B.1$). Outros atributos foram considerados para medir a dimensão do projeto, incluindo o número de integradores e de solicitantes. No entanto, esses atributos eram altamente correlacionados com o número total de *pull requests* (*Spearman* maior que 0,8). Por isso, os autores decidiram utilizar o número de *pull requests* como representante para descrever a dimensão do projeto.

O Conjunto B possui quatro atributos da categoria Social: o primeiro indica se o solicitante é um seguidor de algum integrador da equipe principal do projeto ($B.2$); o segundo mostra se algum integrador é seguidor do solicitante ($B.3$), indicando interesse da equipe em relação às atividades do solicitante; e os outros representam a quantidade de *pull requests* enviados pelo solicitante e integrados por cada integrador anteriormente ($B.4$) e nos últimos 30 dias ($B.5$).

Nos atributos do tipo vetor ($B.4$ até $B.10$), cada posição do vetor representa um desenvolvedor da equipe principal (potenciais integradores) e seu conteúdo é preenchido com um valor numérico correspondente ao significado do atributo. Vale ressaltar que os valores desses atributos são calculados considerando o momento em que o *pull request* é submetido.

Os atributos da categoria Integrador ($B.6$ até $B.10$) revelam o nível de atividade de integração dos *pull requests* de cada integrador. Os valores são calculados e atribuídos para cada integrador do projeto considerando o momento em que o *pull request* foi sub-

Tabela 4.2: Atributos do Conjunto B.

Id	Categoria	Tipo	Nome	Descrição
B.1	Projeto	Numérico	NumPRs	Número de *pull requests* recebidos pelo projeto no momento da submissão do *pull request* corrente
B.2	Social	Binário	RequesterFollowsCT	Se o solicitante é seguidor de algum integrador da projeto
B.3	Social	Binário	CTFollowsRequester	Se algum integrador do projeto é seguidor do solicitante
B.4	Social	Vetor[n_1]	PriorEvaluation	Número de *pull requests* enviados anteriormente pelo solicitante e integrados por cada integrador, onde n_1 representa o número de integradores do projeto
B.5	Social	Vetor[n_1]	RecentEvaluation	Número de *pull requests* enviados anteriormente pelo solicitante e integrados por cada integrador nos últimos 30 dias, onde n_1 representa o número de integradores do projeto
B.6	Integrador	Vetor[n_1]	EvaluationIntegrator	Número de *pull requests* integrados anteriormente por cada integrador, onde n_1 representa o número de integradores do projeto
B.7	Integrador	Vetor[n_1]	RecentEvalIntegrator	Número de *pull requests* integrados anteriormente por cada integrador nos últimos 30 dias, onde n_1 representa o número de integradores do projeto
B.8	Integrador	Vetor[n_1]	AvgEvalTime	A média de tempo entre a submissão do *pull request* e as integrações de cada integrador nos últimos 30 dias, onde n_1 representa o número de integradores do projeto
B.9	Integrador	Vetor[n_1]	TimeFirstEval	Intervalo de tempo entre o envio do *pull request* e a primeira integração de cada integrador, onde n_1 representa o número de integradores do projeto
B.10	Integrador	Vetor[n_1]	TimeLastEval	Intervalo de tempo entre o envio do *pull request* e a última integração de cada integrador, onde n_1 representa o número de integradores do projeto
B.11	*Pull Request*	Vetor[n_2]	FilePaths	TFIDF de cada *filepath* alterado pelo *pull request*, onde n_2 é o número de *filepaths* modificados no projeto

metido. Os atributos são: a quantidade de *pull requests* integrados anteriormente (*B*.6) e nos últimos 30 dias (*B*.7); a média de tempo entre a submissão do *pull request* e as integrações de cada integrador nos últimos 30 dias (*B*.8); o intervalo de tempo entre a submissão de cada *pull request* e a primeira integração de cada integrador (*B*.9); e o intervalo de tempo entre a submissão de cada *pull request* e a última integração de cada integrador (*B*.10). Esses atributos podem auxiliar na tarefa de prever integradores ativos e com experiência para *pull requests* no projeto.

O atributo FilePaths (*B*.11) é representado por um vetor de n_2 posições, onde n_2 é o número de *filepaths* do projeto. Para cada arquivo modificado pelo *pull request*, os diferentes *filepaths* são definidos da seguinte forma: o primeiro *filepath* é composto pelo diretório inicial do caminho completo do arquivo alterado pelo *pull request*. Os demais são gerados por um processo de concatenação, onde cada novo *filepath* é o resultado acumulativo da concatenação do diretório anterior com o próximo subdiretório até que o último elemento seja o caminho completo incluindo o nome do arquivo. Por exemplo, o caminho /dir1/dir2/filename produzirá os seguintes *filepaths*: dir1, /dir1/dir2 e dir1/dir2/filename. Dessa forma, cada *filepath* se torna um novo atributo do *pull request*. Em seguida, o

valor de cada *filepath* é determinado pelo *Term Frequency-Inverse Document Frequency* (TFIDF) [3], que indica a relevância de cada *filepath* e é calculado de acordo com a Equação 4.1.

$$TFIDF_{pr,w} = TF_{pr,w} \times log\left(\frac{N_{PR}}{N_{PR,w}}\right), \tag{4.1}$$

onde $TF_{pr,w}$ é a quantidade de vezes que o *filepath* w ocorreu no *pull request* pr, $N_{PR,w}$ é a quantidade de *pull requests* que possuem o *filepath* w e N_{PR} é a quantidade total de *pull requests* do projeto. Então, $TFIDF_{pr,w}$ é a relevância do *filepath* w para o *pull request* pr.

O Conjunto C é formado por atributos dos Conjuntos A, B e pelos atributos apresentados na Tabela 4.3. O objetivo de colocar todos os atributos no Conjunto C é dual: (i) avaliar se esse conjunto melhoraria os resultados obtidos com os conjuntos anteriores; e (ii) avaliar se um subconjunto de atributos identificados por uma estratégia de seleção de atributos poderia melhorar os resultados.

Tabela 4.3: Atributos do Conjunto C.

Id	Categoria	Tipo	Nome	Descrição
$C.1$	Projeto	Numérico	acceptanceRate	Taxa de aceitação de *pull requests* no projeto
$C.2$	Solicitante	Discreto	loginRequester	Nome de usuário do solicitante no GitHub
$C.3$	Solicitante	Numérico	ageRequester	Tempo do solicitante como usuário do GitHub (em dias)
$C.4$	Solicitante	Discreto	locationRequester	Localização geográfica do solicitante (estado ou cidade)
$C.5$	*Pull Request*	Numérico	recentCommits	Quantidade de *commits* nos arquivos alterados pelo *pull request* nos últimos sete dias
$C.6$	*Pull Request*	Discreto	authorMoreCommits	Integrador com a maior quantidade de *commits* nos arquivos alterados pelo *pull request* nos últimos sete dias
$C.7$	*Pull Request*	Numérico	totalLines	Quantidade de linhas de código alteradas pelo *pull request*
$C.8$	*Pull Request*	Vetor[n_1]	titleWords	TFIDF das palavras mais frequentes utilizadas nos títulos dos *pull requests* submetidos ao projeto, onde $n_1 = 30$
$C.9$	*Pull Request*	Vetor[n_2]	pullLanguages	TFIDF das linguagens de programação utilizas nos *pull requests* submetidos ao projeto, onde n_2 é o número de linguagens de programação utilizadas no projeto
$C.10$	*Pull Request*	Vetor[n_3]	fileNames	TFIDF dos arquivos mais frequentes modificados pelos *pull requests* submetidos ao projeto, onde $n_3 = 30$
$C.11$	*Pull Request*	Vetor[n_4]	changedDirectories	TFIDF dos diretórios que contém arquivos modificados pelos *pull requests* submetidos ao projeto, onde $n_4 = 30$

O primeiro atributo ($C.1$) na Tabela 4.3 representa a taxa de aceitação de *pull requests* do projeto no momento que o *pull request* foi submetido. Os atributos da categoria Solicitante identificam o *login* ($C.2$) do solicitante, o tempo que o solicitante é usuário do GitHub ($C.3$) e a localização geográfica (estado ou cidade) do solicitante ($C.4$). Dois atributos da categoria *Pull Request* indicam a atividade nos últimos sete dias envolvendo os arquivos que foram modificados pelo *pull request*: a quantidade de *commits* que atingiram esses arquivos e o desenvolvedor com a maior quantidade de *commits* nesses arquivos

($C.5$ e $C.6$). O atributo ($C.7$) representa a quantidade de linhas alteradas nos arquivos modificados pelo *pull request*. Esses atributos relativos ao solicitante e às características do *pull request* podem apoiar a previsão de integradores com maior experiência para integrar *pull requests*.

O atributo TitleWords ($C.8$) é representado por um vetor de n_1 posições, onde n_1 é o número de palavras diferentes utilizadas nos títulos dos *pull requests* submetidos ao projeto. Para esse atributo, números, caracteres especiais e *stop words* foram removidos dos títulos. O valor associado para cada palavra é calculado utilizando a equação $TFIDF_{pr,w}$, onde w representa uma palavra e pr o *pull request*. Para evitar a introdução de um grande número de atributos, n_1 foi limitado a 30. Consequentemente, o vetor contém o TFIDF das 30 palavras mais frequentes.

Para o atributo PullLanguages ($C.9$), cada posição do vetor representa uma linguagem de programação utilizada nos arquivos modificados pelos *pull requests* submetido ao projeto, onde n_2 é número de diferentes linguagens de programação utilizadas no projeto, identificadas pelas extensões dos arquivos modificados pelos *pull requests* recebidos no projeto. O valor associado para cada linguagem de programação é calculado utilizando a equação $TFIDF_{pr,w}$, onde w representa uma linguagem e pr o *pull request*.

O atributo FileNames ($C.10$) é representado por um vetor de n_3 posições, onde n_3 é número de arquivos modificados pelos *pull requests* submetidos ao projeto. O valor associado para cada arquivo modificado é calculado utilizando a equação $TFIDF_{pr,w}$, onde w representa um arquivo modificado e pr o *pull request*. Novamente, n_3 foi limitado a 30 e, consequentemente, o vetor contém o TFIDF dos 30 arquivos mais frequentes.

O atributo ChangedDirectories ($C.11$) é representado por um vetor de n_4 posições, onde n_4 é o número de diretórios que contêm arquivos modificados pelos *pull requests* submetidos ao projeto. O valor associado para cada diretório é calculado utilizando a equação $TFIDF_{pr,w}$, onde w representa um diretório e pr o *pull request*. Mais uma vez, n_4 foi limitado a 30 e, consequentemente, o vetor contém o TFIDF dos 30 diretórios mais frequentes.

4.2.2 Dados e Projetos

Para avaliar a previsão de integradores, 106.168 *pull requests* de 32 projetos *open-source* foram coletados a partir do GitHub utilizando a ferramenta GHTorrent [24]. Os projetos selecionados foram utilizados em trabalhos anteriores [7, 65] e possuem características

apropriadas para realizar experimentos no contexto do desenvolvimento baseado em *pull requests*: receberam mais de 1.000 *pull requests* e têm mais de cinco desenvolvedores na equipe que podem ser recomendados para integrar *pull requests*. Além disso, os projetos utilizam linguagens de programação populares como: C, C++, Java, JavaScript, Python, Ruby e Scala.

A extração dos atributos foi realizada a partir de cada projeto separadamente pois as equipes dos projetos são diferentes. Os *pull requests* autointegrados, ou seja, aqueles integrados pelos próprios solicitantes foram descartados. Essas decisões têm o objetivo de evitar a obtenção de resultados inconsistentes e simular mais fielmente o processo de integração de *pull requests*.

A Tabela 4.4 apresenta as características dos projetos selecionados. A coluna Projeto contém o nome dos projetos. A coluna Equipe representa o tamanho das equipes principais dos projetos. A coluna Top-1 contém a porcentagem de *pull requests* integrados pelo integrador que integrou a maior quantidade de *pull requests* no projeto. Esse integrador representa a classe majoritária do projeto. As colunas Top-3 e Top-5 contêm as porcentagens acumuladas de *pull requests* integrados pelos três e cinco integradores que mais integraram *pull requests* no projeto, respectivamente. A coluna Histograma mostra, para cada projeto, a distribuição dos *pull requests* integrados, onde cada barra representa a quantidade de *pull requests* integrados por cada integrador. A coluna Pull Requests contém, para cada projeto, o total de *pull requests* recebidos, a quantidade de *pull requests* selecionados, ou seja, aqueles que não foram autointegrados, e a quantidade de *pull requests* submetidos por solicitantes internos e externos. Por fim, a coluna Lifetime (dias) representa o tempo de vida médio, em dias, dos *pull requests* submetidos por solicitantes internos e externos considerando a datas de submissão e de definição do estado final.

Os projetos foram divididos em grupos de acordo com a característica das equipes principais em relação à quantidade de *pull requests* integrados. No Grupo 1 estão os projetos em que as classes majoritárias Top-3 integraram menos de 50% dos *pull requests*, isso indica que existem vários desenvolvedores que atuam como integradores nessas equipes. Os projetos do Grupo 2 têm classes majoritárias Top-3 responsáveis por mais de 50% dos *pull requests* integrados e a classe majoritária Top-1 responsável por menos de 50% dos *pull requests* integrados, indicando que esses projetos têm equipes menores para fazer a integração dos *pull requests*. O Grupo 3 contém os projetos em que a classe majoritária Top-1 integrou mais de 50% dos *pull requests*, indicando que nesses projetos existe um integrador que centraliza a atividade de integração dos *pull requests*.

Tabela 4.4: Características dos projetos.

			Classe Majoritária (%)				Pull Requests				Lifetime (dias)	
	Projeto	Equipe	Top-1	Top-3	Top-5	Histograma	Total	Selecionado	Interno	Externo	Interno	Externo
Grupo 1	titanium_mobile	44	12,87	30,27	38,78		6.931	6.044	110	5.934	80,43	10,31
	puppet	63	12,49	35,12	52,04		4.105	3.255	300	2.955	41,74	13,05
	docker	42	16,06	35,37	46,73		8.831	6.979	248	6.731	20,10	7,90
	katello	22	14,48	39,69	55,91		5.065	1.066	237	829	1,84	4,74
	phobos	26	21,82	42,71	57,33		3.465	2.744	90	2.654	42,28	15,25
	brackets	38	23,60	45,28	61,26		4.549	3.823	173	3.650	25,51	9,07
	rails	44	32,27	49,54	60,67		13.528	11.728	570	11.158	44,45	20,21
	node	28	23,09	49,67	65,10		1.376	510	216	294	10,26	9,42
Grupo 2	infinispan	15	25,47	51,90	70,49		3.601	3.114	191	2.923	16,75	4,33
	scikit-learn	15	19,85	52,02	76,23		2.800	1.018	146	872	29,37	17,78
	TrinityCore	39	31,70	52,56	60,29		2.793	2.173	95	2.078	17,36	10,95
	xbmc	53	26,49	54,96	60,43		7.486	3.816	70	3.746	152,10	28,23
	angular.js	32	29,75	55,79	74,07		5.739	3.802	566	3.236	43,24	30,23
	commcare-hq	27	35,21	56,45	69,91		7.611	6.986	20	6.966	2,72	1,47
	metasploit-framework	34	25,65	63,25	79,67		5.141	4.535	534	4.001	21,41	12,61
	scala	14	35,37	63,74	85,72		4.705	2.555	108	3.162	12,40	6,67
	akka	11	28,29	64,63	85,70		3.408	1.021	85	936	22,22	10,41
	Baystation12	18	34,15	68,01	75,94		6.493	5.195	219	4.976	1,67	1,18
	joomla-cms	18	33,22	69,68	82,50		5.851	960	155	805	179,84	125,37
	kuma	11	36,43	70,15	91,58		3.250	2.873	43	2.830	13,11	2,78
	jquery	15	49,45	73,85	84,19		2.013	1.202	249	953	25,84	15,01
	ipython	12	32.02	73,96	90,63		4.310	3.234	177	3.057	25,89	9,33
	boto	12	31,97	75,26	90,13		1.684	1.226	108	1.118	42,99	15,47
	rosdistro	7	34,64	82,72	99,44		8.896	7.207	57	7.150	1,23	0,24
	Nancy	7	42,41	92,17	98,54		1.191	893	75	818	48,99	28,27
Grupo 3	zf2	12	59,39	80,03	90,97		5.562	4.705	300	4.405	32,58	15,28
	diaspora	19	59,92	76,53	85,11		2.531	1.746	78	1.668	43,93	15,06
	pandas	12	63,73	84,18	91,63		3.830	2.233	236	1.997	28,69	21,64
	bitcoin	6	65,88	88,60	99,55		4.291	2.934	82	2.852	51,79	19,46
	netty	8	67,32	96,70	99,01		1.704	907	171	736	7,07	9,38
	django	30	67,72	76,36	79,68		5,228	3.731	766	2.965	42,49	26,85
	jekyll	5	77,22	97,76	99,52		1.721	1.238	119	1.119	93,78	34,39
	Média	**23,1**	**36,56**	**64,03**	**76,84**	–	**4.678**	**3.318**	**206**	**3.112**	**38,25**	**17,26**

Nos projetos selecionados, o tempo de vida médio dos *pull requests* foi de 17,26 dias quando foram submetidos por solicitantes externos e de 38,25 dias quando foram submetidos por solicitantes internos. Vale ressaltar que em alguns projetos como, por exemplo, katello (Grupo 1), commcare-hq, Baystation12 e rosdistro (Grupo 2), as equipes foram capazes de integrar *pull requests* que foram aceitos ou rejeitados em um tempo médio menor que cinco dias. Em outros projetos, tais como rails (Grupo 1), xbmc, angular.js, joomla-cms, Nancy (Grupo 2), django e jekyll (Grupo 3), o tempo de vida médio foi superior a 20 dias quando os *pull requests* foram submetidos por solicitantes externos e superior a 42 dias quando foram enviados por solicitantes internos. Em todos os projetos, exceto para katello (Grupo 1) e netty (Grupo 3), os *pull requests* submetidos por solicitantes externos tiveram tempo de vida médio menor do que os submetidos pela equipe principal.

O GitHub fornece uma funcionalidade que permite selecionar um integrador manualmente para integrar um *pull request* aberto. No entanto, entre 106.168 *pull requests* coletados em 32 projetos, 88% não usaram essa funcionalidade. Por exemplo, no projeto joomla-cms, a seleção de um integrador no GitHub não foi utilizada em nenhum *pull request* selecionado.

Por outro lado, existem projetos em que as equipes utilizam essa funcionalidade. Os projetos com mais seleção de integradores, em porcentagem de *pull requests*, foram: brackets (65%), netty (63%), zf2 (45%), metasploit-framework (33%) e angular.js (25%). Nesses projetos, as seleções possuem altas taxas de sucesso (quando os desenvolvedores sugeridos realmente integraram o *pull request*), brackets (95%), netty (84%), zf2 (94%), metaspoit-framework (96%) e angular.js (85%).

Com base nessas estatísticas, observa-se que a seleção de integradores apropriados para revisar e decidir o estado final de novos *pull requests* pode ser uma tarefa útil, mas ainda não é utilizada com frequência. No entanto, estamos considerando como integrador o desenvolvedor que de fato aceitou ou rejeitou o *pull request*, independentemente de ter sido selecionado anteriormente.

4.2.3 Algoritmos de Classificação, Discretização e Seleção de Atributos

Os seguintes algoritmos de classificação foram avaliados nos experimentos de previsão de integradores de *pull requests*: (i) *Naive Bayes* (NB), que é baseado no teorema de Bayes e calcula a probabilidade de cada instância pertencer a cada uma das classes [2, 66]; (ii) J48, que utiliza o conceito de árvores de decisão e implementa o algoritmo C4.5 [34]; (iii) *Random Forest* (RF), que representa o modelo *ensemble* de classificação, ou seja, combina as previsões de vários classificadores, onde cada classificador base é uma árvore de decisão. Para classificar uma instância o *Random Forest* realiza uma votação majoritária entre as classes previstas por cada árvore [2, 35]; (iv) *Instance-Based* (IBk) é uma implementação do algoritmo *k-Nearest Neighbors* (k-NN), que é baseado no conceito de aprendizado por analogia, ou seja, a classe de uma instância será determinada pelo conhecimento das classes de instâncias semelhantes do conjunto de dados de treinamento [2, 67]; e (v) *Sequential Minimal Optimization* (SMO), que é uma implementação do método *Support Vector Machine* (SVM) e busca encontrar uma fronteira de decisão (hiperplano) com a melhor margem de separação para instâncias de classes distintas. Se as instâncias do conjunto de treinamento tiverem duas classes linearmente separáveis,

então há um hiperplano que pode separar adequadamente as instâncias de treinamento das duas classes. O SVM também pode ser utilizado em situações nas quais instâncias do conjunto de treinamento não são linearmente separáveis. Nesse caso, um mapeamento não linear é realizado para transformar os dados de treinamento para outra dimensão, onde as instâncias podem ser separadas linearmente por um hiperplano [2, 68]. Para esse mapeamento, o SVM utiliza uma função de *kernel*.

Esses algoritmos pertencem a diferentes paradigmas de aprendizagem e são estratégias frequentemente sugeridas pelo seu desempenho competitivo [9]. As implementações desses algoritmos estão disponíveis na ferramenta Weka[1] [69]. Essa ferramenta oferece uma ampla gama de algoritmos de classificação, fornece vários filtros de pré-processamento, estratégias de seleção de atributos e o suporte necessário para as etapas de treinamento e teste do processo experimental descrito na Subseção 4.2.4.

Para melhorar a acurácia na tarefa de previsão de integradores de *pull requests*, os atributos com valores numéricos foram discretizados utilizando a estratégia supervisionada *Minimum Description Length* (MDL) [70], que é baseada na minimização da entropia. Essa heurística determina um ponto de corte que divide um intervalo de valores em dois subintervalos. O ponto de corte é definido utilizando a medida Ganho de Informação [2], o que resulta em subintervalos com maior ganho de informação. Em alguns casos, a discretização supervisionada é incapaz de gerar subintervalos, ou seja, é gerado um único valor para o atributo. Para esses casos, foi utilizada a estratégia de discretização não supervisionada *Equal Frequency Interval* (EFI) [3]. Essa estratégia divide o intervalo de um atributo contínuo em intervalos, onde cada intervalo deve conter a mesma quantidade de instâncias.

Além disso, estratégias de seleção de atributos foram utilizadas para identificar subconjuntos mais apropriados de atributos com objetivo de melhorar o desempenho preditivo dos algoritmos de classificação. Para isso, duas estratégias de seleção de atributos foram utilizadas: CFS (*Correlation-based Feature Selection*) [71] e IGAR (*Information Gain Attribute Ranking*) [3], sugeridas por Hall e Holmes [72], pois geralmente são rápidos, oferecem bom desempenho e não dependem do classificador avaliado.

[1] http://www.cs.waikato.ac.nz/ml/weka/

4.2.4 Processo Experimental

O método *k-fold cross validation* é amplamente utilizado para avaliar classificadores. Esse método divide aleatoriamente a base de dados em k partições do mesmo tamanho e o treinamento e teste são realizados em k rodadas de classificação. Em cada rodada, uma das dez partições é utilizada para teste e as demais para treinamento. Nesse caso, a acurácia é definida como a porcentagem de instâncias da base de dados cujas classes foram corretamente classificadas pelo algoritmo [2]. No entanto, esse método pode não ser adequado para bases de dados que têm dependência temporal, uma vez que, devido à geração aleatória dos conjuntos de treinamento e teste, instâncias do futuro poderiam ser utilizadas para prever o passado.

O método proposto nesta tese para avaliação dos classificadores, chamado *training-test sliding validation*, preserva a ordem cronológica de submissão dos *pull requests* na divisão de bases da dados em conjuntos de treinamento e teste. Nesse método, cada rodada de classificação consiste de 10% das instâncias para treinamento e 1% de instâncias imediatamente posteriores para teste. Por exemplo, considerando 100 subconjuntos ordenados de instâncias, na primeira rodada, o conjunto de treinamento é composto pelas instâncias do 1º até o 10º subconjuntos e o conjunto de teste é o 11º subconjunto. Na segunda rodada, o conjunto de treinamento é composto pelas instâncias do 2º ao 11º subconjuntos e o conjunto de teste é o 12º subconjunto, e assim por diante. O tamanho da janela de movimentação é 1%, o que permite que 90% das instâncias sejam testadas. A acurácia do método é estimada pela porcentagem de instâncias testadas cujas classes foram corretamente classificadas pelo algoritmo em todas as rodadas.

O conceito de recomendação Top-k representa o conjunto de k integradores sugeridos para integrar um *pull request*. Com o objetivo de fornecer uma recomendação Top-k para uma instância, primeiro o classificador calcula a probabilidade de cada integrador a ser recomendado. Dessa forma, o valor de probabilidade mais alto define a recomendação Top-1, os dois valores mais altos indicam a recomendação Top-2 e assim sucessivamente até a recomendação Top-k. Em seguida, a recomendação Top-k é comparada com o real integrador do *pull request*. Assim, a acurácia das recomendações Top-k é definida pelo número de recomendações Top-k em que o real integrador do *pull request* foi indicado pela recomendação Top-k para aquele *pull request*.

Para medir o desempenho preditivo dos classificadores, foram utilizadas as medidas de acurácia e de ganho normalizado da acurácia obtida por um classificador quando comparada com a classe majoritária (*baseline*). A ideia da medida de ganho normalizado é

avaliar o quanto um classificador aumenta a acurácia dentro da margem de melhoria. Essa margem é a diferença entre 100% e a acurácia da classe majoritária. A medida de ganho normalizado da acurácia de um classificador avaliado em um projeto p, isto é, sobre a base de dados que representa um projeto p, é denotada por f_p e definida na Equação 4.2 [73].

$$f_p = \begin{cases} \dfrac{Acurácia_p - Baseline_p}{1 - Baseline_p}, se\ Acurácia_p > Baseline_p \\ \\ \dfrac{Acurácia_p - Baseline_p}{Baseline_p}, em\ caso\ contrário \end{cases} \tag{4.2}$$

onde $Acurácia_p$ representa a acurácia obtida pelo algoritmo de classificação no projeto p e $Baseline_p$ representa o percentual da classe majoritária no projeto p, ou seja, a taxa de acerto quando a classe majoritária é sempre sugerida.

Finalmente, para a avaliação da significância estatística dos resultados [74], foram adotados o teste não-paramétrico de *Friedman* [75] e pós-teste de *Nemenyi* [76, 77]. Em ambos os testes estatísticos, o nível de significância foi de 0,05. Os resultados dos testes estatísticos foram obtidos com a ferramenta R[2].

4.3 Resultados e Discussões

Esta seção descreve e analisa os resultados dos experimentos realizados para avaliar atributos preditivos no problema de previsão de integradores de *pull requests*. Os resultados estão organizados da seguinte forma. A Subseção 4.3.1 apresenta o ajuste de parâmetros de entrada para alguns algoritmos utilizando o Conjunto C: o valor de k para o algoritmo IBk, o *kernel* do algoritmo SMO e o número de árvores de decisão utilizadas pelo algoritmo *Random Forest*. Também é realizado o ajuste do *kernel* para o algoritmo SMO com o Conjunto B, uma vez que os autores não divulgaram o *kernel* adotado em seu trabalho. A Subseção 4.3.2 compara o desempenho dos algoritmos de classificação utilizando o Conjunto C. Em seguida, na tentativa de melhorar os resultados obtidos, nas Subseções 4.3.3 e 4.3.4, o melhor algoritmo é avaliado novamente após a utilização de técnicas de discretização e estratégias de seleção de atributos. A Subseção 4.3.5 apresenta a avaliação e comparação de desempenho dos algoritmos de classificação para as recomendações Top-1, Top-3 e Top-5 considerando os três conjuntos de atributos, onde o Conjunto A representa o nosso trabalho em estágio preliminar [65], o Conjunto B representa a proposta de Jiang *et al.* [7] e o Conjunto C representa o estágio atual da abordagem

[2] `http://www.r-project.org/`

apresentada nesta tese. Por último, a Subseção 4.3.6 discute a relevância dos atributos na tarefa de prever integradores de *pull requests*.

4.3.1 Ajuste de Parâmetros dos Algoritmos de Classificação

Para identificar os melhores valores dos parâmetros de entrada de alguns algoritmos, foram realizadas experimentos preliminares utilizando o Conjunto C: a variação do valor de k (número de vizinhos) para o algoritmo IBk, a variação do *kernel* no algoritmo SMO (*Polynomial*, *Puk* e RBF) e a variação do número de árvores de decisão construídas pelo algoritmo *Random Forest*.

Nos próximos experimentos, cada tabela apresenta quatro medidas para indicar o desempenho dos algoritmos de classificação: (i) a Média das Acurácias, definida pela divisão entre a soma das acurácias de cada algoritmo e a quantidade de projetos; (ii) a média dos ganhos normalizados, ou simplesmente, Média dos Ganhos, que representa a soma dos ganhos normalizados de cada projeto dividida pelo número de projetos; (iii) a Média dos Rankings, calculada da seguinte forma: para cada projeto, cada algoritmo recebe um número de posto ordenado, 1 para a melhor acurácia, 2 para a segunda melhor e n para a pior acurácia, onde n é o número de algoritmos avaliados; em seguida, os postos de cada algoritmo são somados e o resultado é dividido pelo número de projetos; quanto menor a média melhor será o desempenho do algoritmo; e (iv) o Número de Vitórias, definido como o número de vezes em que cada algoritmo atingiu a melhor acurácia, considerando todos os projetos. Além disso, os melhores resultados para cada medida são destacados em negrito.

A Tabela 4.5 mostra os resultados quando k recebe números ímpares de 1 a 19. O algoritmo IBk utilizando o valor k igual 15 obteve os melhores resultados nas medidas Média das Acurácias, Média dos Ganhos, Média dos Rankings e o segundo melhor Número de Vitórias.

Tabela 4.5: Resultados obtidos pelo IBk com diferentes valores de k para o Conjunto C.

	Valor de k para IBk									
Medida	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**
Média das Acurácias	40,57%	42,05%	43,40%	43,92%	44,26%	44,52%	44,50%	**44,55%**	44,49%	44,35%
Média dos Ganhos	4,42%	7,77%	10,26%	11,33%	11,94%	12,36%	12,35%	**12,39%**	12,29%	11,09%
Média dos Rankings	8,81	8,66	6,98	6,02	4,66	3,88	3,64	**3,36**	4,25	4,75
Número de Vitórias	3	1	1	2	3	1	5	6	3	**7**

A Tabela 4.6 apresenta os resultados obtidos pelo SMO utilizando três *kernels* nos Conjuntos *B* e *C*. Os *kernels* também foram avaliados para a proposta de Jiang *et al.* [7], uma vez que os autores não informaram qual *kernel* foi utilizado pelo classificador SVM. Com base nos resultados, o *kernel* RBF apresentou o melhor desempenho, considerando todas as medidas, para os Conjuntos *B* e *C*.

Tabela 4.6: Resultados obtidos pelo SMO com *kernels Polynomial*, *Puk* e RBF para os Conjuntos B e C.

	Conjunto *B*			Conjunto *C*		
Medida	**Polynomial**	**Puk**	**RBF**	**Polynomial**	**Puk**	**RBF**
Média das Acurácias	41,81%	41,11%	**44,96%**	43,93%	40,76%	**46,15%**
Média dos Ganhos	6,03%	7,03%	**12,90%**	9,62%	6,31%	**14,85%**
Média dos Rankings	2,30	2,36	**1,34**	2,06	2,63	**1,31**
Número de Vitórias	8	3	**21**	9	1	**22**

Os resultados para a variação do número de árvores utilizadas pelo algoritmo *Random Forest* são apresentados na Tabela 4.7. Para esse parâmetro, foram utilizadas 50 árvores de decisão como valor mínimo e 550 para o máximo, com incremento de 50 árvores. O valor 450 obteve os melhores resultados considerando todas as medidas.

Tabela 4.7: Resultados obtidos pelo *Random Forest* com diferentes números de árvores para o Conjunto C.

	Número de árvores para Random Forest										
Medida	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**	**550**
Média das Acurácias	46,30%	46,95%	46,97%	47,07%	47,12%	47,20%	47,23%	47,23%	**47,34%**	47,29%	47,30%
Média dos Ganhos	14,89%	16,11%	16,07%	16,24%	16,30%	16,43%	16,50%	16,48%	**16,70%**	16,60%	16,65%
Média dos Rankings	9,83	7,58	7,17	6,84	6,48	5,25	4,84	4,98	**3,77**	4,69	4,56
Número de Vitórias	0	2	4	1	1	3	4	2	**7**	4	4

Com base nesses resultados, foram selecionados os seguintes parâmetros de entrada para os próximos experimentos: *kernel* RBF para o algoritmo SMO nos Conjuntos *B* e *C*, k igual a 15 para o algoritmo IBk e 450 árvores de decisão para o algoritmo *Random Forest* no Conjunto *C*.

4.3.2 Comparação dos Algoritmos de Classificação no Conjunto C

No experimento seguinte, foram avaliados cinco algoritmos de classificação utilizando o Conjunto *C*: IBk, J48, *Naive Bayes* (NB), *Random Forest* (RF) e SMO, com o objetivo de identificar o algoritmo que apresenta o melhor desempenho para esse conjunto de

atributos. De acordo com os resultados da Tabela 4.8, o algoritmo *Random Forest* obteve o melhor desempenho considerando todas as medidas.

Tabela 4.8: Resultados obtidos pelos diferentes classificadores no Conjunto C.

Medida	IBk ($k = 15$)	J48	Naive Bayes	Random Forest	SMO (RBF)
Média das Acurácias	44,50%	45,77%	37,77%	**47,34%**	46,15%
Média dos Ganhos	12,39%	13,19%	-1,03%	**16,70%**	14,85%
Média dos Rankings	3,41	3,20	4,47	**1,69**	2,23
Número de Vitórias	1	6	0	**17**	8

Com o objetivo de melhorar o desempenho e identificar subconjuntos adequados de atributos, foram avaliadas técnicas de discretização e estratégias de seleção de atributos no Conjunto C. Os resultados são descritos nas subseções seguintes (4.3.3 e 4.3.4).

4.3.3 Melhorando o Desempenho com Discretização

O objetivo do próximo experimento é verificar o desempenho do algoritmo RF utilizando o Conjunto C após a discretização dos atributos numéricos. Duas técnicas foram utilizadas para discretizar os atributos do Conjunto C, denominadas: *Minimum Description Length* (MDL) e *Equal Frequency Interval* (EFI).

Inicialmente, a primeira versão da base de dados com os atributos do Conjunto C assumindo valores numéricos foi discretizada pela técnica MDL. Nesse caso, os atributos foram discretizados somente se a técnica pudesse separar os valores numéricos em mais de um intervalo discreto. Em seguida, a partir da primeira versão, foi criada a segunda versão: para os atributos em que a técnica supervisionada não conseguiu obter mais de um intervalo, os dados foram discretizados pela técnica EFI com duas faixas de valores. Então, na segunda versão, todos os atributos com valores numéricos foram transformados em valores discretos.

A Tabela 4.9 mostra os resultados para o algoritmo RF utilizando atributos numéricos e discretos no Conjunto C. O algoritmo RF utilizando apenas valores discretos nos atributos preditivos obteve os melhores resultados.

Tabela 4.9: Resultados obtidos após a discretização de atributos no Conjunto C.

		Atributos Discretos	
Medidas	Atributos Numéricos	MDL	MDL e EFI
Média das Acurácias	47,34%	47,49%	**48,77%**
Média dos Ganhos	16,70%	16,81%	**18,61%**
Média dos Rankings	2,16	2,34	**1,50**
Número de Vitórias	10	1	**21**

4.3.4 Identificando Melhores Subconjuntos de Atributos

Utilizando o Conjunto C com valores discretos, duas estratégias de seleção de atributos foram aplicadas, CFS e IGAR, para avaliar a existência de subconjuntos de atributos capazes de melhorar o desempenho preditivo.

A estratégia CFS procura por subconjuntos de atributos que são mais correlacionados com a classe e menos correlacionados uns com os outros. Nessa estratégia, o algoritmo de busca *BestFirst* realiza uma pesquisa *forward*. Essa busca começa com o subconjunto vazio e avalia cada atributo, identificando o melhor. Em seguida, cada um dos atributos restantes é testado com o primeiro atributo para encontrar o melhor par de atributos. Na iteração seguinte, cada um dos atributos restantes é testado em conjunto com o melhor par para encontrar o melhor grupo de três atributos. A busca termina quando a adição de um novo atributo não melhora a avaliação do subconjunto corrente [72].

Já a estratégia IGAR retorna um número fixo de atributos (definido por um parâmetro de entrada) que individualmente estão mais correlacionados com a classe (os que possuem o maior ganho de informação) para serem utilizados pelo algoritmo de classificação.

Com a estratégia IGAR, foram considerados 13 subconjuntos de atributos, onde o primeiro subconjunto era composto por 50 atributos e o último por 650 atributos, com um incremento de 50 atributos. Vale ressaltar que o Conjunto C possui 35 atributos distintos. Entretanto, alguns desses atributos são vetores de atributos e isso pode levar a um total de 1.023 atributos individuais, dependendo do projeto.

A Tabela 4.10 apresenta os resultados para a variação do número de atributos retornados pela estratégia IGAR. O subconjunto com 300 atributos obteve o melhor desempenho considerando todas as medidas. Assim, esse valor foi escolhido para representar a estratégia IGAR.

Tabela 4.10: Resultados obtidos pelo algoritmo RF com diferentes números de atributos para a estratégia IGAR.

	Number of attributes for IGAR strategy												
Medidas	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**	**550**	**600**	**650**
Média das Acurácias	48,01%	49,09%	49,07%	49,12%	49,09%	**49,59%**	49,03%	49,06%	48,95%	48,86%	48,77%	48,80%	48,72%
Média dos Ganhos	17,35%	19,23%	19,09%	19,15%	19,17%	**19,93%**	19,01%	19,00%	18,94%	18,84%	18,65%	18,66%	18,56%
Média dos Rankings	9,56	6,83	6,97	7,14	6,73	**3,59**	6,17	6,31	6,80	6,94	7,95	7,70	8,30
Número de Vitórias	4	3	2	2	2	**6**	2	1	3	3	1	3	0

A Tabela 4.11 apresenta o desempenho das estratégias de seleção de atributos. O algoritmo RF utilizando a estratégia IGAR obteve o melhor desempenho considerando todas as medidas. Assim, o Conjunto C com valores discretos nos atributos, utilizando a es-

tratégia de seleção de atributos IGAR e utilizando o algoritmo RF obteve os melhores resultados. Essa combinação de técnicas, estratégias e conjuntos de atributos representa nossa abordagem e será comparada com as abordagens de trabalhos anteriores na seção seguinte.

Tabela 4.11: Resultados obtidos após seleção de atributos no Conjunto C.

			Seleção de Atributos	
Medidas	**Atributos Numéricos**	**Atributos Discretos**	**CFS**	**IGAR**
Média das Acurácias	47,34%	48,77%	48,68%	**49,59%**
Média dos Ganhos	16,70%	18,61%	18,49%	**19,93%**
Média dos Rankings	3,06	2,45	2,80	**1,69**
Número de Vitórias	4	4	9	**15**

4.3.5 Avaliação das Diferentes Abordagens

O experimento seguinte compara a capacidade de previsão dos algoritmos considerando três abordagens. Essa comparação tem como objetivo determinar qual abordagem, isto é, que combinação de conjunto de atributos e algoritmo, fornece os melhores resultados para o problema de previsão de integradores. Para os Conjuntos A e B, foram utilizados os algoritmos propostos nos trabalhos anteriores. Para o Conjunto A, avaliamos o algoritmo RF com 450 árvores de decisão, esse número foi identificado pelo ajuste de parâmetros de entrada (número de árvores para o algoritmo *Random Forest*) nesse conjunto de atributos. Dessa forma, chamamos de **Abordagem A** [65], proposta preliminar do nosso trabalho, o uso do algoritmo RF com 450 árvores de decisão no Conjunto A. Para o Conjunto B, tendo em vista que os autores não informaram sobre o *kernel* adotado para o algoritmo SVM, escolhemos aquele que apresentou os melhores resultados em nosso ajuste: RBF. Portando, denominamos **Abordagem B** [7], o uso do algoritmo SMO com o *kernel* RBF no Conjunto B. Para o Conjunto C, utilizamos os resultados apresentados pelo algoritmo RF, depois de ajustar o número de árvores de decisão e considerando os processos de discretização e seleção de atributos. Assim, chamamos de **Abordagem C**, proposta nesta tese, o uso de RF com 450 árvores, atributos com valores discretizados pelas técnicas MDL e EFI, e 300 atributos selecionados com a estratégia IGAR no Conjunto C. As abordagens são avaliadas e comparadas a seguir.

A Tabela 4.12 mostra, para cada abordagem, os resultados das recomendações Top-1, Top-3 e Top-5. A Abordagem C apresentou os melhores resultados para todas as medidas quando comparadas com as Abordagens A e B.

Tabela 4.12: Resultados obtidos pelas três abordagens para as recomendações Top-1, Top-3 e Top-5.

	Top-1 Abordagens			Top-3 Abordagens			Top-5 Abordagens		
Medidas	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
Média das Acurácias	45,18%	44,96%	**49,59%**	73,93%	75,00%	**78,60%**	85,39%	86,84%	**88,73%**
Média dos Ganhos	12,75%	12,90%	**19,93%**	25,93%	31,74%	**41,91%**	33,22%	40,74%	**52,60%**
Média dos Rankings	2,53	2,34	**1,13**	2,72	2,25	**1,03**	2,78	1,91	**1,31**
Número de Vitórias	1	2	**29**	0	1	**31**	0	9	**23**

No cenário Top-1, a Abordagem B alcançou a maior acurácia em dois projetos e a Abordagem A alcançou a maior acurácia em apenas um projeto. Note que, para as Abordagens A e B, os algoritmos obtiveram médias de ganho normalizado muito semelhantes, embora tenham sido treinados com diferentes conjuntos de atributos. A Abordagem C alcançou a maior acurácia em 29 projetos. Além disso, no teste de Friedman, a hipótese nula foi rejeitada com p-valor $= 7,901 \times 10^{-9}$, indicando que há uma diferença estatística nos resultados representados pelas acurácias. No pós-teste de Nemenyi, a Abordagem C obteve acurácias estatisticamente maiores quando comparada com a Abordagem A (p-valor $= 5,6 \times 10^{-8}$) e com a Abordagem B (p-valor $= 3,3 \times 10^{-6}$).

Para a recomendação Top-3, a Abordagem C obteve maior acurácia em 31 projetos, maior média dos *rankings* e maior média de ganho normalizado que a Abordagem A e que a Abordagem B. O teste de Friedman indicou diferença estatística com p-valor $= 2,85 \times 10^{-11}$ para os resultados da recomendação Top-3 em termos de acurácia. No pós-teste de Nemenyi, a Abordagem C obteve acurácias estatisticamente maiores quando comparada com as Abordagen A (p-valor $= 4,4 \times 10^{-11}$) e com a Abordagen B (p-valor $= 3,3 \times 10^{-6}$).

No contexto da recomendação Top-5, a Abordagem B alcançou as maiores acurácias em nove projetos. No entanto, a Abordagem C obteve as maiores acurácias em 23 projetos e obteve média de ganho normalizado de 52,60%. No teste de Friedman, a hipótese nula foi rejeitada com p-valor $= 2,591 \times 10^{-8}$, indicando que há diferença estatística entre os resultados das recomendações Top-5 em termos de acurácia. No pós-teste de Nemenyi, a diferença estatística foi novamente observada entre as Abordagens A e C (p-valor $= 1,3 \times 10^{-8}$) e entre as Abordagens B e C (p-valor $= 0,0462$).

Para detalhar um pouco mais a avaliação das abordagens, as acurácias das recomendações Top-1, Top-3 e Top-5 foram comparadas com três *baselines*, representadas pelas porcentagens da classe majoritária, das três classes e das cinco classes majoritárias.

A Tabela 4.13 apresenta, para cada projeto, os ganhos normalizados obtidos utilizando as Abordagens A, B e C. As colunas abaixo do título Baseline contêm, para cada projeto, a porcentagem de *pull requests* integrados pelo integrador que integrou a maior quantidade de *pull requests* (Top-1) e as porcentagens acumuladas de *pull requests* integrados pelos três e cinco integradores que mais integraram *pull requests* no projeto, Top-3 e Top-5, respectivamente. As colunas de cada abordagem indicam, em cada projeto, os ganhos normalizados das recomendações Top-1, Top-3 e Top-5. Além disso, o melhor ganho normalizado em cada recomendação Top-k está destacado em negrito.

Tabela 4.13: Ganho normalizado das Abordagens A, B e C por projeto

					Ganho Normalizado								
		Baseline			Abordagem A			Abordagem B			Abordagem C		
	Projeto	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Grupo 1	titanium_mobile	12,87	30,27	38,78	33,54	54,87	66,04	27,12	46,91	60,68	**39,57**	**62,25**	**74,88**
	puppet	12,49	35,12	52,04	23,63	37,11	39,85	17,59	29,86	40,26	**28,97**	**44,20**	**51,38**
	docker	16,06	35,37	46,73	9,33	28,22	45,94	12,51	38,45	56,20	**19,78**	**46,63**	**65,72**
	katello	14,48	39,69	55,91	16,80	25,39	38,81	10,59	18,12	28,65	**21,43**	**35,58**	**46,84**
	phobos	21,82	42,71	57,33	12,48	27,82	35,72	13,99	34,00	42,30	**19,20**	**35,50**	**46,47**
	brackets	23,60	45,28	61,26	6,81	17,60	26,92	5,59	15,79	30,64	**14,88**	**27,91**	**45,20**
	rails	32,27	49,54	60,67	0,94	15,83	23,09	8,08	25,33	32,70	**8,62**	**28,00**	**39,49**
	node	23,09	49,67	65,10	1,99	-1,31	3,04	**3,41**	**13,53**	18,11	2,55	**13,53**	**15,59**
	Média do Grupo 1				13,19	25,69	34,92	12,36	27,75	38,69	**19,38**	**38,31**	**48,19**
Grupo 2	infinispan	25,47	51,90	70,49	13,48	33,89	52,22	6,13	28,25	47,78	**15,70**	**41,14**	**57,74**
	scikit-learn	19,85	52,02	76,23	4,30	6,11	-1,68	3,48	**11,13**	8,46	**11,43**	10,21	**9,84**
	TrinityCore	31,70	52,56	60,29	-6,47	1,24	23,80	-0,03	**14,40**	24,91	**4,11**	14,19	**44,04**
	xbmc	26,49	54,96	60,43	13,09	25,16	40,31	**45,03**	17,90	32,17	23,11	**42,87**	**59,06**
	angular.js	29,75	55,79	74,07	0,36	9,91	10,03	7,73	27,64	35,94	**13,05**	**30,94**	**37,56**
	commcare-hq	35,21	56,45	69,91	6,10	28,98	38,82	7,33	19,70	25,52	**12,69**	**32,08**	**45,46**
	metasploit-framework	25,65	63,25	79,67	21,59	19,29	14,02	11,96	13,44	17,02	**25,53**	**25,55**	**28,18**
	scala	35,37	63,74	85,72	16,68	59,43	70,31	13,55	62,41	79,55	**27,28**	**67,21**	**83,54**
	akka	28,29	64,63	85,70	27,82	49,36	55,45	31,35	**55,27**	**73,15**	**39,46**	54,62	70,00
	Baystation12	34,15	68,01	75,94	10,54	36,01	71,61	8,28	45,17	80,47	**14,76**	**51,98**	**83,17**
	joomla-cms	33,22	69,68	82,50	2,08	4,55	16,63	3,80	9,89	**37,83**	**9,36**	**13,32**	31,26
	kuma	36,43	70,15	91,58	33,32	68,54	63,66	23,06	46,16	53,92	**35,33**	**69,58**	**75,18**
	jquery	49,45	73,85	84,19	-11,43	20,00	46,17	-1,52	27,04	11,19	**2,20**	**34,49**	**57,31**
	ipython	32,02	73,96	90,63	21,70	32,64	36,18	23,12	29,34	43,76	**27,42**	**48,54**	**64,78**
	boto	31,97	75,26	90,13	41,41	63,38	63,32	47,01	65,60	**80,85**	**53,93**	**75,87**	78,01
	rosdistro	34,64	82,72	99,44	7,97	14,47	-0,37	6,88	26,74	**75,00**	**9,59**	**31,66**	16,07
	Nancy	42,41	92,17	98,54	35,09	-0,46	-0,68	21,20	16,99	-0,79	**36,19**	**24,90**	**73,29**
	Média do Grupo 2				13,98	27,79	35,28	13,60	31,26	43,51	**21,24**	**42,54**	**53,79**
Grupo 3	zf2	59,39	80,03	90,97	21,00	48,37	41,09	14,01	54,73	47,51	**23,91**	**61,04**	**56,26**
	diaspora	59,92	76,53	85,11	14,65	47,12	65,21	15,44	57,73	**79,72**	**29,64**	**62,68**	70,79
	pandas	63,73	84,18	91,63	23,30	44,75	61,29	31,27	60,49	**82,08**	**34,99**	**65,87**	80,88
	bitcoin	65,88	88,60	99,55	2,37	10,53	-0,58	12,81	13,60	15,56	**17,38**	**27,28**	**66,67**
	netty	67,32	96,70	99,01	**2,91**	-0,03	-0,37	1,01	**25,15**	**13,13**	-1,16	13,94	-0,49
	django	67,72	76,36	79,68	-5,20	1,27	17,77	**-0,31**	7,28	6,84	-0,52	**14,89**	**32,04**
	jekyll	77,22	97,76	99,52	5,75	-0,21	-0,43	8,56	43,30	62,50	**17,34**	**59,82**	**77,08**
	Média do Grupo 3				9,25	21,69	26,28	11,83	37,47	43,90	**17,37**	**44,51**	**54,75**
	Média Geral				12,75	25,93	33,22	12,90	31,74	42,39	**19,93**	**41,91**	**52,60**

Os projetos no Grupo 1 têm equipe principal com mais de 22 integradores e as classes majoritárias Top-3 integraram menos de 50% dos *pull requests*. Assim, vários desenvolvedores da equipe principal muitas vezes desempenham o papel de integrador. A Abordagem C atingiu médias de ganho normalizado de 19,38%, 38,31% e 48,19% para as recomendações Top-1, Top-3 e Top-5. Essas médias são melhores do que os resultados apresentados pelas Abordagens A e B. Considerando as recomendações Top-1, Top-3 e Top-5 para os oito projetos desse grupo, a Abordagem C obteve a melhor acurácia em 23 (sendo um empate) dos 24 casos. A Abordagem B obteve apenas duas melhores acurácias (sendo um empate).

No Grupo 2, a equipe principal dos projetos têm entre 7 e 53 integradores, as três classes majoritárias (Top-3) são sempre responsáveis por mais de 50% dos *pull requests*, mas a classe majoritária (Top-1) nunca é responsável por mais de 50% das *pull requests*, ou seja, um pequeno número de integradores realiza a integração dos *pull requests*. Nesse grupo, a Abordagem C apresentou as melhores médias de ganho normalizado quando comparadas com as médias das Abordagens A e B para todas as recomendações. Considerando as recomendações Top-1, Top-3 e Top-5 para os 17 projetos desse grupo, a Abordagem C obteve a melhor acurácia em 43 casos de 51. A Abordagem B obteve apenas oito melhores resultados. Em alguns projetos do Grupo 2, como scala e Baystation12, que têm seus *pull requests* distribuídos entre equipes com menos integradores (14 e 18), as Top-5 classes majoritárias são 85,72% e 75,94%, respectivamente. No entanto, a Abordagem C obteve acurácias na recomendação Top-5 de 97,65% e 95,95%, proporcionando ganhos normalizados de 83,54% e 83,17%. Isso mostra que em equipes menores, onde os *pull requests* são integrados por poucos integradores, a recomendação Top-5 alcançou um ganho normalizado acima de 83% e acurácias acima de 95%.

O Grupo 3 é composto por projetos em que a classe majoritária (Top-1) integrou mais de 50% dos *pull requests*, isto é, mais da metade dos *pull requests* foram integrados por um único integrador. Isso mostra que esses projetos têm um integrador responsável por realizar grande parte dessa tarefa. No entanto, alguns *pull requests* ainda devem ser atribuídos a outros integradores que não são a classe majoritária. Considerando o comportamento geral, a Abordagem C obteve o melhor desempenho quando comparada com as Abordagens A e B. Nos sete projetos desse grupo, a Abordagem C obteve a melhor acurácia em 15 casos de 21 para as recomendações Top-1, Top-3 e Top-5. As Abordagens A e B juntas obtiveram apenas seis melhores acurácias. Nesse grupo, destacamos as recomendações para o projeto pandas feitas pela Abordagem C, onde os ganhos normalizados foram 34,99%, 65,87% e 80,88% para as recomendações Top-1, Top-3 e Top-5, respectivamente. Por outro

lado, os projetos netty e django foram os únicos em que as recomendações Top-1 e Top-5 utilizando a Abordagem C não melhoraram as *baselines*.

4.3.6 Importância dos Atributos

Nesta subseção, discutimos a importância dos atributos para a tarefa de previsão de integradores em três projetos, selecionados com valores elevados de ganho normalizado, um de cada grupo: titanium_mobile (Grupo 1), boto (Grupo 2) e pandas (Grupo 3). Para cada atributo, sua importância é avaliada pela medida de ganho de informação. Os atributos com os maiores valores de ganho de informação são considerados os mais relevantes para a tarefa de classificação. Para cada classificador gerado, os atributos são classificados de acordo com o valor de seu ganho de informação. O atributo com o valor mais alto recebe a posição 1 e com o valor mais baixo recebe a última posição. Em seguida, somamos as posições de cada atributo e dividimos o resultado pelo número de classificadores gerados. Esse valor representa a média das posições para cada atributo. Para os atributos do tipo vetor, existem duas médias: a primeira considera a melhor média entre os elementos no vetor e a segunda considera a média geral de todos os elementos no vetor.

A Tabela 4.14 apresenta duas colunas para cada projeto. A coluna Atributo contém o identificador de cada atributo e está ordenada de acordo com o valor da coluna Média, que representa a média das posições de cada atributo e, para os atributos do tipo vetor, a melhor média entre os elementos do vetor e a média geral para todos os elementos do vetor. Em outras palavras, os atributos são ranqueados de acordo com sua importância nas bases de treinamento de cada projeto. Com isso, os atributos mais importantes se encontram na parte de cima e os menos importantes na parte de baixo da tabela. As colunas abaixo do título Legenda contêm o identificador e nome de cada atributo apenas para facilitar a identificação dos mesmos.

O atributo RequesterLogin ($C.2$) foi considerado muito importante, uma vez que apareceu como o melhor atributo em quase todas as bases de treinamento dos três projetos. Além disso, os atributos RequesterLocation ($C.3$), FilePaths ($B.11$) e ActiveDeveloper ($C.6$) estão entre os quatro mais importantes nos projetos boto e pandas. No projeto titanium_mobile, a previsão dos integradores de *pull requests* é fortemente influenciada por atributos que caracterizam o solicitante que enviou a contribuição e suas relações sociais ($C.2$, $A.11$ e $B.4$). Por outro lado, alguns atributos da categoria *Pull Request*, como NumCommits ($A.1$), NumAddedLines ($A.2$) e NumDeleteLines ($A.3$) estão entre aqueles que pouco contribuem para a previsão do integrador mais adequado. Além disso, o atributo menos

importante nos três projetos foi o ProjectAcceptRate ($C.1$). Em resumo, é possível notar que vários atributos da parte de cima da tabela em todos os projetos foram introduzidos pelo Conjunto C, representando uma contribuição importante desta tese para futuras ferramentas que visam prever desenvolvedores para *pull requests*.

Tabela 4.14: Importância dos atributos.

titanium_mobile		boto		pandas		Legenda	
Atributo	**Média [Média Vetor]**	**Atributo**	**Média [Média Vetor]**	**Atributo**	**Média [Média Vetor]**	**Id**	**Nome**
$C.2$	1,04	$C.2$	1,00	$C.2$	1,00	$A.1$	NumCommits
$A.11$	2,98	$C.3$	2,52	$C.3$	9,70	$A.2$	NumAddedLines
$B.4$	3,87 [212,24]	$B.11$	11,09 [523,73]	$C.6$	11,63	$A.3$	NumDeletedLines
$A.8$	11,64	$C.6$	17,73	$B.11$	17,02 [577,68]	$A.4$	NumChangedFiles
$C.4$	12,73	$A.13$	33,82	$C.11$	38,56 [103,99]	$A.5$	RequesterType
$C.3$	19,31	$A.5$	35,17	$A.13$	40,57	$A.6$	RequesterNumPRs
$A.9$	20,18	$C.11$	45,34 [274,22]	$A.14/B.2$	43,19	$A.7$	RequesterMergedPRs
$B.7$	20,18 [358,08]	$A.14/B.2$	58,93	$A.5$	44,64	$A.8$	RequesterRejectPRs
$A.10$	20,80	$B.4$	82,72 [257,02]	$B.3$	64,47	$A.9$	RequesterAcceptRate
$A.12$	25,63	$B.6$	87,92 [239,76]	$C.8$	76,64 [250,37]	$A.10$	RequesterRejectRate
$B.11$	30,45 [629,39]	$B.9$	118,089 [199,11]	$B.6$	110,16 [195,43]	$A.11$	RequesterNumFollowers
$A.6$	31,33	$B.10$	124,21 [203,87]	$B.8$	110,47 [215,57]	$A.12$	RequesterNumFollowing
$C.11$	38,877 [403,42]	$B.7$	124,53 [227,00]	$C.10$	120,95 [381,81]	$A.13$	RequesterWatches
$C.9$	42,07 [192,72]	$B.8$	131,60 [223,59]	$C.9$	122,52 [330,55]	$A.14/B.2$	RequesterFollowsCT
$A.7$	42,72	$C.10$	134,92 [368,18]	$B.5$	134,31 [239,57]	$B.1$	NumPRs
$C.6$	44,16	$B.5$	144,75 [272,14]	$B.7$	135,76 [215,28]	$B.3$	CTFollowsRequester
$C.8$	61,27 [241,72]	$C.4$	185,58	$B.4$	137,95 [253,04]	$B.4$	PriorEvaluation
$B.10$	92,90 [167,15]	$B.3$	190,19	$A.4$	196,31	$B.5$	RecentEvaluation
$B.9$	93,59 [256,26]	$B.1$	211,09	$B.9$	199,76 [279,59]	$B.6$	EvaluationIntegrator
$A.13$	101,47	$C.9$	221,40 [312,66]	$A.8$	209,86	$B.7$	RecenteEvalIntegrator
$B.1$	135,63	$C.8$	222,15 [344,05]	$A.6$	210,93	$B.8$	AvgEvalTime
$A.14/B.2$	151,89	$A.11$	222,53	$C.7$	222,40	$B.9$	TimeLastEval
$B.6$	157,14 [326,47]	$A.12$	233,44	$A.7$	225,51	$B.10$	TimeFirstEval
$B.3$	165,70	$C.5$	248,70	$B.10$	227,42 [276,92]	$B.11$	FilePaths
$C.10$	171,45 [277,99]	$A.6$	259,53	$A.2$	232,76	$C.1$	ProjectAcceptRate
$B.5$	175,80 [338,27]	$A.7$	264,53	$A.12$	233,11	$C.2$	RequesterLogin
$A.5$	184,32	$A.4$	268,33	$A.11$	238,18	$C.3$	RequesterLocation
$B.8$	191,70 [416,86]	$A.8$	269,52	$C.5$	239,70	$C.4$	RequesterAge
$C.5$	337,28	$A.9$	274,18	$A.9$	244,84	$C.5$	CommitsFiles
$A.2$	362,19	$A.10$	275,24	$A.3$	247,46	$C.6$	ActiveDeveloper
$A.3$	363,66	$C.7$	275,51	$B.1$	248,92	$C.7$	TotalLines
$C.7$	364,26	$A.2$	277,31	$C.4$	249,19	$C.8$	TitleWords
$A.1$	371,80	$A.3$	277,67	$A.1$	259,64	$C.9$	PullLanguages
$A.4$	375,19	$A.1$	280,40	$A.10$	263,38	$C.10$	FileNames
$C.1$	785,39	$C.1$	548,14	$C.1$	422,42	$C.11$	ChangedDirectories

4.4 Limitações e Ameaças à Validade dos Resultados

Nesta seção, destacamos algumas decisões e aspectos do trabalho que poderiam limitar ou comprometer alguns resultados. Para cada *pull request*, utilizamos as informações de quem realmente integrou o *pull request* como uma referência para indicar se a previsão

está correta ou não. No entanto, não há garantia de que o integrador mais adequado seja quem realmente integrou o *pull request*. Isso pode acontecer por várias razões: o integrador mais adequado pode estar de férias, fora do projeto ou com uma carga de trabalho muito grande. Essas situações podem levar à escolha de outro integrador para integrar o *pull request* que não o mais adequado. Para mitigar essa ameaça, analisamos também as recomendações Top-3 e Top-5, ou seja, quando o desenvolvedor sugerido é o mais adequado mas não integrou o *pull request* devido às razões acima mencionadas, e o desenvolvedor que realmente integrou o *pull request* estava entre as recomendações Top-3 ou Top-5, sendo considerada então, nesses casos, correta a previsão.

As instâncias utilizadas para avaliar as Abordagens A, B e C também foram utilizadas no ajuste de parâmetros dessas abordagens. Embora isso possa fornecer alguma vantagem para a Abordagem C, as outras duas abordagens também tiveram alguns parâmetros ajustados com essas instâncias: o número de árvores utilizadas pela Abordagem A e o *kernel* utilizado pela Abordagem B.

Nossos resultados foram obtidos com o uso de dados extraídos de projetos *open-source*. Portanto, não podemos garantir que os resultados sejam os mesmos para projetos industriais. No entanto, selecionamos projetos de várias linguagens de programação, com diferentes tamanhos de equipes de desenvolvimento (pelo menos cinco desenvolvedores) e distribuições diferentes de classes majoritárias. Além disso, todos os projetos possuem maturidade no uso do modelo de desenvolvimento baseado em *pull requests*.

Para os atributos do Conjunto B, implementamos a extração sugerida pelo estudo de Jiang *et al.* [7], pois os autores não disponibilizaram os dados. Assim, os resultados relacionados a este estudo podem não ser exatamente os mesmos resultados originais. Além disso, os autores não definiram o *kernel* utilizado pelo classificador SVM. Para solucionar esse problema, avaliamos três *kernels* e atribuímos o melhor para a Abordagem B.

4.5 Considerações Finais

Neste capítulo, foram utilizados algoritmos de classificação com objetivo de avaliar o desempenho da tarefa de prever integradores de *pull requests*. Nesses experimentos, propusemos um método para avaliar classificadores em bases de dados nas quais a ordem das instâncias é importante. Além disso, fizemos uma sólida comparação de cinco algoritmos de classificação (IBk, J48, *Naive Bayes*, *Random Forest* e SMO) utilizando os atributos definidos nos Conjuntos A, B e C em 32 projetos *open-source*. Para a Abordagem A, que re-

presenta os nossos resultados preliminares [65], adotamos o algoritmo RF com 450 árvores de decisão e o Conjunto A; para a Abordagem B, que representa os resultados de Jiang *et al.* [7], uma vez que os autores não informaram sobre o *kernel* utilizado para o algoritmo SVM, escolhemos o *kernel* que apresentou as melhores acurácias em nosso ajuste de parâmetros (RBF) e o Conjunto B; e para a Abordagem C, que representa a proposta desta tese, adotamos o algoritmo RF, depois de ajustar o número de árvores de decisão (450), considerando os processos de discretização (utilizando as técnicas MDL e EFI) e seleção de atributos (300 atributos selecionados utilizando a estratégia IGAR), e o Conjunto C.

Na grande maioria dos casos, a Abordagem C apresentou o melhor desempenho. Nossa abordagem atingiu uma média da ganho normalizado de 19,93%, 41,91% e 52,60% para as recomendações Top-1, Top-3 e Top-5, respectivamente. Essas médias são melhores do que os resultados apresentados quando as Abordagens A e B são utilizadas. Além disso, a Abordagem C apresentou as melhores acurácias em 29 dos 32 projetos considerando a recomendação Top-1.

O melhor ganho normalizado para as recomendações Top-1 e Top-3 ocorreu no projeto boto (53,93% e 75,87%), onde a acurácia foi de 68,66% e 93,85%, respectivamente. Para a recomendação Top-5, o maior ganho (83,54%) ocorreu no projeto scala com uma acurácia de 97,56%. Em contrapartida, apenas dois projetos (netty e django) utilizando a Abordagem C não obtiveram ganhos para a recomendação Top-1.

Em uma análise mais aprofundada sobre a relevância dos atributos nos três conjuntos de atributos, observamos que alguns dos principais atributos foram introduzidos pelo Conjunto C. Essa é uma contribuição fundamental da pesquisa, uma vez que futuros desenvolvedores de ferramentas podem considerar a adoção desses atributos para melhorar a acurácia de suas ferramentas.

Com base nos resultados, podemos concluir que a abordagem de considerar todos os atributos possíveis e aplicar estratégias de seleção de atributos pode contribuir de forma consistente no problema de previsão de integradores de *pull requests*. Podemos também observar que nossa abordagem apresentou os melhores ganhos normalizados em projetos onde a integração dos *pull requests* é alocada entre vários integradores. No entanto, também conseguimos ganhos em projetos com poucos integradores.

Capítulo 5

Previsão do Tempo de Vida de Pull Requests

5.1 Introdução

Este capítulo apresenta as contribuições da tese no cenário de previsão do tempo de vida (*lifetime*) de *pull requests*.

O tempo de vida de um *pull request* é representado pelo resultado da diferença entre a data em que um integrador encerrou o processo de revisão, aceitando ou rejeitando, e a data em que o desenvolvedor submeteu o *pull request*. Assim, o tempo de vida reflete o tempo de duração e não exatamente o tempo de integração de um *pull request*. Em geral, os próprios integradores gerenciam sua disponibilidade para revisar e integrar *pull requests* recebidos em projetos *open-source*. Por isso, o tempo consumido efetivamente na revisão e integração do código de um *pull request* é difícil de medir.

Em muitos projetos *open-source* (*e.g.*, rails e rosdistro), a quantidade de *pull requests* recebidos é bastante alta e as contribuições podem demorar um tempo consideravelmente elevado para receber a atenção dos integradores. Nesse contexto, uma informação que se torna importante é conhecer antecipadamente quanto tempo um *pull request* pode demorar no estado aberto. Essa informação pode ser utilizada por gerentes da equipe principal na triagem de *pull requests* e por integradores no gerenciamento do tempo disponível.

Na Figura 5.1, é possível observar a quantidade de *pull requests* por mês no projeto rails, no período de setembro de 2011 até abril de 2017. Cada barra vertical corresponde a um mês do ano e são compostas da seguinte forma: (i) quantidade de *pull requests* abertos que foram revisados e integrados no mesmo mês (vermelho); (ii) quantidade de *pull*

requests abertos que não foram revisados e integrados no mesmo mês (verde); e (iii) quantidade de *pull requests* revisados e integrados no mês e abertos em meses anteriores (azul).

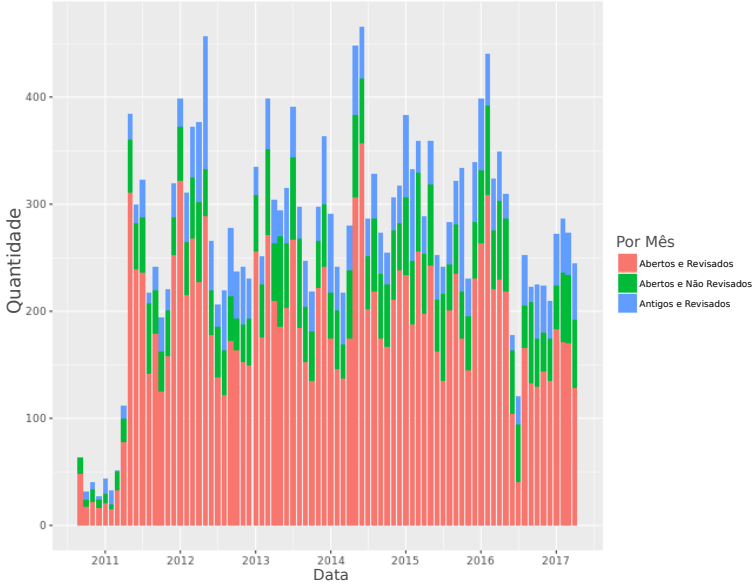

Figura 5.1: Quantidade de *pull requests* por mês no projeto rails [4].

O projeto rails possui uma comunidade bastante ativa na utilização do modelo de desenvolvimento baseado em *pull requests*. No entanto, a Figura 5.1 mostra que uma quantidade considerável dos *pull requests* recebidos pelo projeto não são integrados no mesmo mês da submissão. De acordo com Gousios *et al.* [6], o tempo necessário para fazer a integração de *pull requests* é um dos fatores que dificulta a priorização daqueles que necessitam ser integrados. Esse tipo de projeto ilustra o contexto onde a previsão do tempo de vida se torna relevante.

Nos experimentos apresentados neste capítulo, foram utilizados 97.603 *pull requests* coletados a partir projetos *open-source* por meio da ferramenta GHTorrent. Com o objetivo de melhorar o desempenho da tarefa de prever o tempo de vida de *pull requests*, esta tese propõe a utilização de algoritmos de classificação e regressão para avaliar dois conjuntos de atributos: o primeiro representado pelos atributos propostos por Yu *et al.* [8] e o segundo constituído por todos os atributos do primeiro conjunto além de um novo subconjunto que acreditamos ser útil na tarefa de previsão do tempo de vida de *pull requests*. Além disso, nossa proposta defende que estratégias de seleção de atributos podem identificar subconjuntos de atributos capazes de aprimorar o desempenho dos algoritmos. Na comparação com a proposta anterior [8], nossa abordagem obteve a melhor acurácia em 18 dos 20 projetos avaliados. A acurácia média foi de 45,28% para prever o tempo de vida de *pull requests*, o que proporcionou um ganho normalizado médio de 14,68% em relação à classe majoritária.

Este capítulo está organizado da seguinte forma. A Seção 5.2 apresenta os materiais e métodos utilizados nos experimentos. A Seção 5.3 discorre sobre os resultados obtidos pelos algoritmos de regressão e classificação na tarefa de prever o tempo de vida. A Seção 5.4 discute limitações e ameaças à validade deste estudo experimental. Por último, a Seção 5.5 apresenta as considerações finais do capítulo.

5.2 Materiais e Métodos

Nos experimentos de previsão do tempo de vida de *pull requests*, os algoritmos de regressão e de classificação foram avaliados em dois conjuntos de atributos preditivos (Conjuntos D e E). Na Subseção 5.2.1, são apresentas as descrições dos atributos em cada conjunto. Os projetos utilizados na avaliação são descritos na Subseção 5.2.2. A Subseção 5.2.3 descreve os algoritmos de classificação, regressão e as estratégias de seleção de atributos utilizados durante o processo experimental. Por fim, os métodos adotados na condução dos experimentos estão detalhados na Subseção 5.2.4.

5.2.1 Conjuntos de Atributos

Esta seção apresenta os dois conjuntos de atributos preditivos avaliados no cenário do tempo de vida de *pull requests*. O Conjunto D contém os atributos propostos pela abordagem de Yu *et al.* [8]. Já o Conjunto E é composta pelos atributos do conjunto anterior adicionado de novos atributos, que representam a proposta desta tese.

A Tabela 5.1 apresenta o identificador, a categoria, o tipo, o nome e a descrição dos atributos que compõem o Conjunto D. Esses atributos estão organizados em três categorias: Projeto, *Pull Request* e Solicitante. Na categoria Projeto, os atributos representam informações sobre a idade do projeto ($D.1$), o tamanho das equipes de desenvolvimento ($D.2$), a razão entre a média de *commits* que atingiram arquivos alterados pelo *pull request* e o total de *commits* do projeto nos últimos três meses ($D.3$), e a quantidade de *pull requests* que estão abertos no projeto ($D.4$).

Os atributos da categoria *Pull Request* identificam algumas características sobre as contribuições, tais como: a quantidade de palavras presentes no título e na descrição do *pull request* ($D.5$), o número de *commits* ($D.6$), a quantidade de linhas adicionadas e excluídas ($D.7$), a quantidade de comentários ($D.8$), o intervalo de tempo entre a submissão e a primeira interação de um desenvolvedor no *pull request* ($D.9$), se o *pull request* utilizou

integração contínua[1] (*D*.10), se o *pull request* possui arquivos de teste (*D*.11), se o *pull request* foi submetido em uma sexta-feira (*D*.12), e se o título ou a descrição do *pull request* possui alguma referência para *issues* (*D*.13) ou desenvolvedores (*D*.14).

Os atributos *D*.8 e *D*.9, propostos por Yu *et al.* [8], são desconhecidos no momento em que o *pull request* é submetido, ou seja, seus valores são atualizados após a submissão. Esse fato torna inapropriada a utilização desses atributos na tarefa de prever o tempo de vida de *pull request* no momento que eles são recebidos no projeto. Desta forma, desconsideramos ambos os atributos para reduzir as ameaças à validade do nosso estudo.

Tabela 5.1: Atributos do Conjunto D.

Id	Categoria	Tipo	Nome	Descrição
D.1	Projeto	Numérico	*ProjectAge*	Idade do projeto, em minutos, no momento da submissão do *pull request*
D.2	Projeto	Numérico	*TeamSize*	Quantidade de integradores ativos (integrou ao menos um *pull request* de outro solicitante) nos três meses anteriores à submissão do *pull request*
D.3	Projeto	Numérico	*AreaHotness*	Número médio de *commits* nos arquivos alterados pelo *pull request* dividido pelo número de *commits* realizados no projeto, considerando os últimos três meses
D.4	Projeto	Numérico	*Workload*	Quantidade de *pull requests* abertos em cada projeto no momento da submissão do *pull request* atual
D.5	*Pull Request*	Numérico	*DescriptionSize*	Quantidade de palavras no título e na descrição do *pull request*
D.6	*Pull Request*	Numérico	*NumCommits*	Quantidade de *commits* incluídos no *pull request*
D.7	*Pull Request*	Numérico	*NumChurn*	Quantidade de linhas adicionadas e excluídas pelo *pull request*
~~*D*.8~~	~~*Pull Request*~~	~~Numérico~~	~~*NumComments*~~	~~Quantidade de comentários no *pull request*~~
~~*D*.9~~	~~*Pull Request*~~	~~Numérico~~	~~*FirstResponse*~~	~~Intervalo de tempo em minutos entre a submissão do *pull request* e o primeiro comentário dos revisores~~
D.10	*Pull Request*	Binário	*UseCI*	Indica se o *pull request* utilizou integração contínua
D.11	*Pull Request*	Binário	*TestInclusion*	Indica se o *pull request* possui pelo menos um arquivo de teste (com base no nome do arquivo e expressões regulares do caminho)
D.12	*Pull Request*	Binário	*FridayEffect*	Indica se o *pull request* foi recebido em uma sexta-feira
D.13	*Pull Request*	Binário	*IssueTag*	Indica a presença de notificações para *issues* (*e.g. Ticket* #7) no título ou na descrição do *pull request*
D.14	*Pull Request*	Binário	*MentionTag*	Indica a presença de notificações para desenvolvedores (*e.g.* @mlimeira) no título ou na descrição do *pull request*
D.15	Solicitante	Binário	*RequesterType*	Indica se o solicitante é um integrador do projeto
D.16	Solicitante	Numérico	*RequesterAcceptRate*	Taxa de aceitação dos *pull requests* submetidos pelo solicitante
D.17	Solicitante	Numérico	*RequesterNumFollowers*	Quantidade de desenvolvedores do GitHub que são seguidores do solicitante

[1]Integração contínua é uma prática de desenvolvimento de software onde os membros de uma equipe integram seu trabalho com frequência, geralmente cada desenvolvedor integra pelo menos diariamente – podendo haver múltiplas integrações por dia. As integrações são verificadas por testes automatizados para detectar erros de integração de forma rápida [78].

A categoria Solicitante possui atributos que indicam informações sobre a reputação do solicitante (*requester*), ou seja: a informação que indica se o solicitante tem permissão de escrita no repositório principal, ou seja, se pode integrar o código enviado por um *pull request* ($D.15$), a informação que representa a porcentagem de *pull requests* submetidos pelo solicitante cujo estado final foi aceito ($D.16$), e a informação que representa a quantidade de seguidores do solicitante ($D.17$).

A Tabela 5.2 contém os atributos que, nesta tese, consideramos que podem ajudar na previsão do tempo de vida de *pull requests*. O Conjunto E contém os atributos do Conjunto D e os atributos dessa tabela. Esses atributos estão organizados nas seguintes categorias: Projeto, *Pull Request*, Solicitante e *Social*. Os atributos da categoria Projeto representam: a quantidade de *pull requests* recebidos no projeto antes da submissão do *pull request* corrente ($E.1$), a porcentagem de *commits* realizados no projeto por meio de *pull requests* ($E.2$), a quantidade de linhas de código do projeto ($E.3$), e a quantidade de observadores do projeto, ou seja, o número de estrelas ($E.4$).

Os outros atributos da categoria *Pull Request* representam algumas características do *pull request*, tais como: o número de arquivos adicionados ($E.5$), excluídos ($E.6$) e modificados ($E.7$), a quantidade total de arquivos alterados ($E.8$), o número de arquivos de código fonte ($E.9$), de documentação ($E.10$) e outros tipos de arquivos alterados ($E.11$), e o número de linhas de testes alteradas ($E.12$).

Alguns atributos da categoria Solicitante caracterizam o perfil do desenvolvedor como, por exemplo, a quantidade de *pull requests* submetidos ($E.13$), a quantidade de desenvolvedores seguidos pelo solicitante ($E.14$), o tempo como usuário do GitHub ($E.15$) e se é um observador do projeto ($E.16$). Os demais atributos dessa categoria revelam se o solicitante participou de algum evento (envio de comentários, *issues*, *pull requests* e *commits*) no projeto antes do envio do *pull request* corrente: quantidade de interações em *issues* ($E.17$), quantidade de comentários em *issues* ($E.18$), quantidade de interações em *pull requests* ($E.19$), quantidade de comentários em *pull requests* ($E.20$), quantidade de *commits* ($E.21$), quantidade de comentários em *commits* ($E.22$).

Na categoria *Social*, os atributos indicam se o solicitante é seguidor de algum integrador da equipe de desenvolvimento ($E.23$) e se algum integrador do projeto é seguidor do solicitante ($E.24$).

Tabela 5.2: Atributos adicionados ao Conjunto D para formar o Conjunto E.

Id	Categoria	Tipo	Nome	Descrição
E.1	Projeto	Numérico	*PrevPullProject*	Quantidade de *pull requests* recebidos pelo projeto até a submissão do *pull request* atual
E.2	Projeto	Numérico	*ExternalContribs*	Porcentagem de *commits* realizados por *pull requests* no projeto até um mês antes
E.3	Projeto	Numérico	*Sloc*	Quantidade de linhas de código no projeto
E.4	Projeto	Numérico	*Stars*	Número de estrelas do projeto no momento em que o *pull request* foi submetido.
E.5	*Pull Request*	Numérico	*NumAddedFiles*	Quantidade de arquivos adicionados pelo *pull request*
E.6	*Pull Request*	Numérico	*NumDeletedFiles*	Quantidade de arquivos excluídos pelo *pull request*
E.7	*Pull Request*	Numérico	*NumModifiedFiles*	Quantidade de arquivos modificados pelo *pull request*
E.8	*Pull Request*	Numérico	*NumChangedFiles*	Quantidade total de arquivos alterados (adicionado, excluído e modificado) pelo *pull request*
E.9	*Pull Request*	Numérico	*NumSrcFiles*	Quantidade de arquivos de código alterados pelo *pull request*
E.10	*Pull Request*	Numérico	*NumDocFiles*	Quantidade de arquivos de documentação alterados pelo *pull request*
E.11	*Pull Request*	Numérico	*NumOtherFiles*	Quantidade de outros arquivos alterados pelo *pull request*
E.12	*Pull Request*	Numérico	*NumTestChurn*	Quantidade de linhas de teste alteradas (adicionadas e excluídas) pelo *pull request*
E.13	Solicitante	Numérico	*NumPrevPullRequester*	Quantidade de *pull requests* submetidos pelo solicitante até o momento da submissão
E.14	Solicitante	Numérico	*RequesterNumFollowing*	Quantidade de desenvolvedores seguidos pelo solicitante
E.15	Solicitante	Numérico	*RequesterAge*	Tempo do solicitante como usuário do GitHub (em minutos)
E.16	Solicitante	Binário	*WatcherProject*	Indica se o solicitante é um observador do projeto
E.17	Solicitante	Numérico	*IssueEvents*	Quantidade de interações do solicitante em *issues* antes da submissão
E.18	Solicitante	Numérico	*IssueComments*	Quantidade de comentários do solicitante em *issues* antes da submissão
E.19	Solicitante	Numérico	*PullEvents*	Quantidade de interações do solicitante em *pull requests* antes da submissão
E.20	Solicitante	Numérico	*PullComments*	Quantidade de comentários do solicitante em *pull requests* antes da submissão
E.21	Solicitante	Numérico	*NumPrevCommits*	Quantidade de *commits* do solicitante antes da submissão
E.22	Solicitante	Numérico	*NumCommitComments*	Quantidade de comentários do solicitante em *commits* antes da submissão
E.23	*Social*	Binário	*RequesterFollowsCT*	Indica se o solicitante é seguidor de algum integrador
E.24	*Social*	Binário	*CTFollowsRequester*	Indica se algum integrador é seguidor do solicitante

5.2.2 Dados e Projetos

Os experimentos no cenário de previsão do tempo de vida envolveram 97.603 *pull requests* coletados a partir de 30 projetos *open-source* por meio da ferramenta GHTorrent. Os projetos selecionados receberam mais de 1.000 *pull requests* e possuem atividade constante no processo de integração de *pull requests*. Além disso, os projetos são desenvolvidos predominantemente com algumas das linguagens de programação mais populares[2] no GitHub: Java, JavaScript, Python, Ruby e Scala.

[2]O GitHut permite visualizar e explorar a popularidade das linguagens de programação entre desenvolvedores do GitHub. Disponível em: `http://githut.info/`.

A Figura 5.2 mostra a distribuição do tempo de vida dos *pull requests* em *boxplots* na escala log_{10}, de acordo com o estado final do *pull request*. A distribuição do tempo de vida dos *pull requests* nos *boxplots* está representada em minutos. No entanto, esses valores também podem ser interpretados de forma discreta, utilizando outras representações de tempo. Por exemplo, entre os *pull requests* com estado final aceito, pode-se observar a seguinte distribuição para os diferentes intervalos de tempo: 23,88% são integrados em minutos (de 1 até 60 minutos), 34,37% em horas (de 1 até 24 horas), 25,30% em dias (de 1 até 7 dias), 10,98% em semanas (de 1 até 4 semanas) e 5,47% em meses (acima de 1 mês). Nessa discretização, cada intervalo possui uma quantidade significativa de *pull requests*.

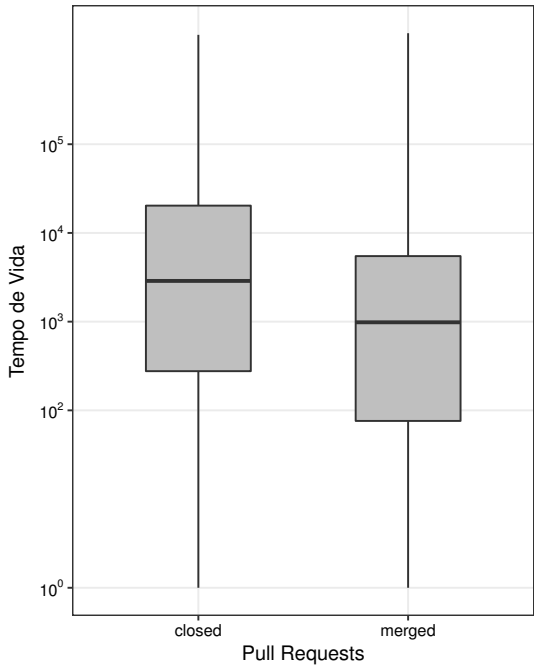

Figura 5.2: Tempo de vida dos *pull requests* nos projetos selecionados.

Nos projetos selecionados, os *pull requests* cujo código foi aceito no repositório principal do projeto representam 76,5% do total de *pull requests*. Observando os *boxplots* da Figura 5.2, os *pull requests* com estado final aceito (*merged*) são tratados mais rapidamente que os *pull requests* cujo o código foi rejeitado (*closed*). Entre os *pull requests* aceitos, a média do tempo de vida foi de 11.735 minutos, aproximadamente oito dias, e a mediana foi de 941 minutos, ou seja, em torno de 15 horas. Por outro lado, nos *pull requests* rejeitados, a média atingiu 42.442 minutos, aproximadamente 29 dias, e a mediana alcançou 2.716 minutos, isto é, quase dois dias.

A Figura 5.3 apresenta, para cada um dos 30 projetos selecionados, a distribuição do tempo de vida dos *pull requests*. Cada *boxplot* representa a distribuição do tempo de

vida dos *pull requests* no projeto em escala log_{10}. Os pontos em vermelho nos *boxplots* de alguns projetos são considerados *outliers*. Pode-se observar que todos os projetos têm *pull requests* com tempo de vida nos cinco intervalos de tempo considerados, representados do lado esquerdo da figura. Além disso, a maioria dos projetos tem maior concentração de *pull requests* com tempo de vida nos intervalos de horas e dias, exceto para os projetos appium e rosdistro, onde cerca de 45% e 70% dos *pull requests* possuem tempo de vida no intervalo de minutos, respectivamente.

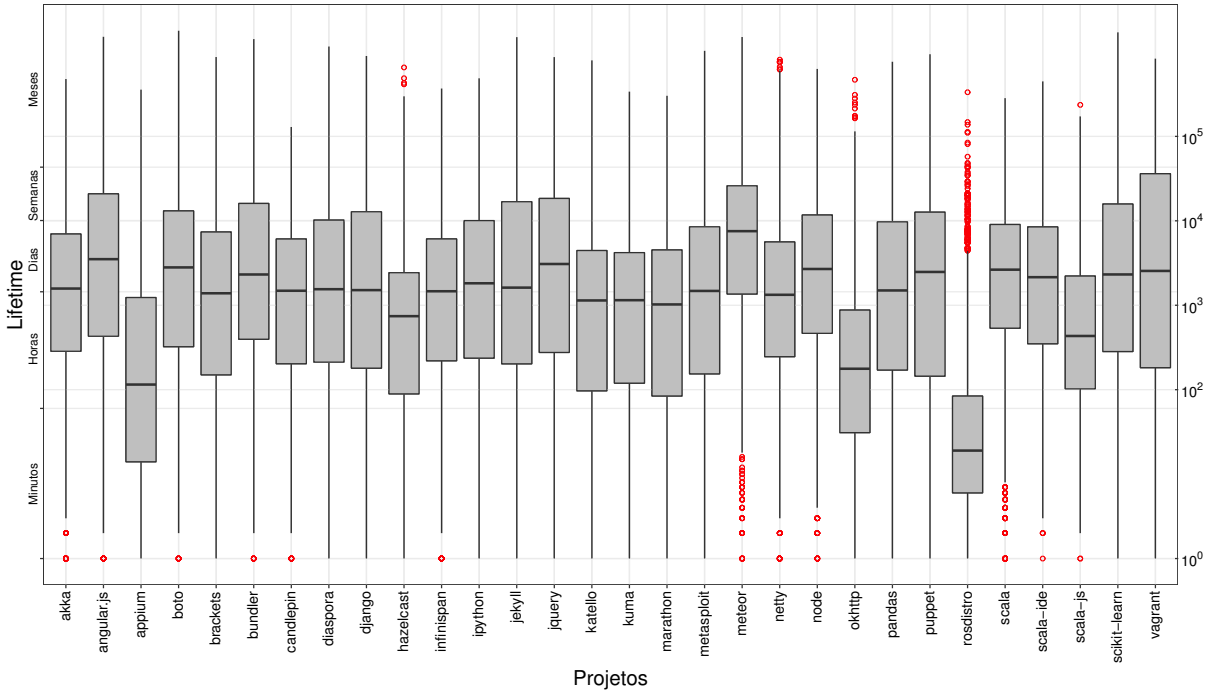

Figura 5.3: Tempo de vida de *pull requests* por projeto.

A Tabela 5.3 contém as características dos projetos utilizados. Para cada projeto, a coluna Linguagem indica a linguagem de programação predominante entre os arquivos dos projetos. A coluna PRs representa, para cada projeto, o número total de *pull requests*. As colunas abaixo do título Status indicam, para cada projeto, as porcentagens de *pull requests* aceitos e rejeitados. As colunas abaixo do título Média de Tempo indicam, para cada projeto, as médias de tempo de vida dos *pull requests* de acordo com o estado final, em dias. As colunas abaixo do título Tempo em Classes representam, em porcentagem, a distribuição dos *pull requests* em relação ao tempo de vida, onde cada classe é definida por um intervalo de tempo. Além disso, a classe majoritária de cada projeto está destacada em negrito.

A maioria dos projetos selecionados possui mais *pull requests* aceitos que rejeitados, exceto os projetos: netty, angular.js, jquery e node. Em média, todas as equipes dos projetos

Tabela 5.3: Características dos projetos.

			Status (%)		Média de Tempo		Tempo em Classes (%)				
Linguagem	**Projetos**	**PRs**	**Aceito**	**Rejeitado**	**Aceito**	**Rejeitado**	**Minutos**	**Horas**	**Dias**	**Semanas**	**Meses**
Java	candlepin	1.093	87,64	12,36	3,55	5,03	14,93	34,23	**36,42**	12,82	1,60
	hazelcast	1.832	91,27	8,73	2,13	18,79	20,92	**47,88**	23,92	4,87	2,41
	infinispan	3.990	67,14	32,86	5,14	4,25	13,58	**36,44**	33,87	13,02	3,09
	netty	2.028	49,70	50,30	6,20	15,17	15,34	**37,97**	29,73	10,50	6,46
	okhttp	1.194	88,27	11,73	1,38	17,47	35,37	**46,58**	10,89	4,61	2,55
	Média	2.027	76,80	23,20	3,68	12,14	20,03	40,62	26,97	9,16	3,22
JavaScript	angular.js	6.190	36,39	63,61	27,20	40,48	12,27	**26,89**	25,93	17,24	17,67
	appium	1.984	87,85	12,15	1,01	11,81	**45,77**	33,47	15,73	3,33	1,71
	brackets	4.575	87,23	12,77	7,54	31,53	19,21	**33,07**	27,41	12,66	7,65
	jquery	2.032	30,91	69,09	15,03	16,41	16,45	**25,74**	24,47	19,45	13,89
	meteor	1.299	57,51	42,49	22,75	51,30	10,55	16,86	**30,10**	22,79	19,71
	node	2.851	33,36	66,64	11,08	17,32	12,03	27,57	**33,22**	15,22	11,96
	Média	2.548	59,37	40,63	11,482	25,67	20,80	27,34	26,18	14,69	10,98
Python	boto	1.711	87,61	12,39	14,66	76,04	14,68	26,89	**29,94**	16,96	11,53
	django	6.006	68,88	31,12	15,72	38,54	15,88	**33,85**	22,88	12,29	15,10
	ipython	4.429	87,33	12,67	7,67	31,68	14,74	**31,56**	28,96	15,24	9,49
	kuma	3.565	89,20	10,80	2,69	9,23	19,24	**39,64**	30,52	8,92	1,68
	pandas	4.476	77,12	22,88	9,80	36,82	17,82	**32,49**	25,61	13,14	10,94
	rosdistro	10.155	93,75	6,25	0,20	1,40	**70,72**	26,12	2,49	0,55	0,12
	scikit-learn	3.213	80,82	19,18	17,37	87,24	14,63	**30,25**	25,15	14,60	15,38
	Média	5.168	85,64	14,36	7,546	33,27	27,43	32,01	22,55	10,49	7,52
Ruby	bundler	1.327	77,32	22,68	24,34	40,06	11,15	**33,38**	25,32	14,54	15,60
	diaspora	2.612	76,65	23,35	9,20	39,26	14,97	**34,11**	25,92	15,31	9,69
	jekyll	1.908	74,21	25,79	17,45	16,41	16,43	**31,99**	21,98	12,18	17,42
	katello	5.170	90,48	9,52	3,46	12,42	21,03	**37,90**	29,41	8,99	2,67
	metasploit-framework	5.341	86,20	13,80	10,62	35,33	17,66	**32,39**	27,45	12,08	10,43
	puppet	4.592	82,51	17,49	10,26	28,70	19,80	**26,24**	25,98	16,64	11,35
	vagrant	1.622	77,01	22,99	21,50	67,65	20,22	**25,40**	16,83	15,29	22,26
	Média	3.727	82,08	17,92	12,658	32,10	19,03	30,78	24,33	13,03	12,82
Scala	akka	4.175	71,62	28,38	4,32	9,14	9,08	**38,83**	34,92	13,56	3,62
	marathon	1.360	87,06	12,94	3,10	16,86	22,28	**36,99**	27,57	9,41	3,75
	scala	4.774	81,38	18,62	5,47	8,94	6,35	**35,13**	36,18	18,12	4,23
	scala-ide	1.050	86,95	13,05	5,93	15,62	9,45	**35,46**	34,32	15,18	5,59
	scala-js	1.049	94,19	5,81	1,61	15,59	16,97	**52,24**	25,17	4,19	1,43
	Média	2.482	84,24	15,76	4,09	13,23	12,82	39,73	31,63	12,09	3,72
	Média Geral	3.253	76,25	23,75	9,61	27,22	18,98	33,59	26,28	12,46	8,70

selecionados aceitaram os *pull requests* em menos de um mês. No entanto, as equipes de integração dos projetos angular.js, brackets, meteor, boto, django, ipython, pandas, scikit-learn, bundler, diaspora, metasploit-framwork e vagrant levam, em média, mais de um mês para rejeitar *pull requests*. Os projetos jekyll e infinispan foram os únicos onde os *pull requests* aceitos têm a média do tempo de vida maior que os *pull requests* rejeitados.

Na distribuição do tempo de vida em valores discretos, é possível observar que, os integradores geralmente integram os *pull requests* entre uma e 24 horas, considerando os projetos selecionados. Por outro lado, nos projetos candlepin, meteor, node e boto, a classe de 1 até 7 dias contém a maior porcentagem de *pull requests* integrados.

O tempo de vida de *pull requests* pode variar bastante. Enquanto a maioria dos *pull requests* é integrada em minutos, poucas horas e até em alguns dias, existe uma quantidade significativa de *pull requests* com valor do tempo de vida muito alto, acima de 30 dias. Além disso, o valor do tempo de vida de *pull requests* de projetos *open-source* pode ser influenciado por diferentes fatores, tais como, a utilização de integração contínua, a existência de conflito entre o código enviado no *pull request* e o código do repositório principal, o tamanho da equipe de integradores, a sobrecarga de atividade dos integradores e a falta de comunicação entre o solicitante e a equipe de integração.

5.2.3 Algoritmos de Classificação, Regressão e Seleção de Atributos

Nos experimentos com os valores numéricos para o tempo de vida, os seguintes algoritmos foram avaliados: (i) *Linear Regression* [3] representando a técnica de regressão linear múltipla; (ii) M5P (M5Prime) [3] como implementação do algoritmo de árvore de modelos lineares; (iii) *Random Forest* [35] representando um método *ensemble*; e (iv) SMOReg (*Sequential Minimal Optimization*) [79] como correspondente do algoritmo SVM para regressão.

Já nos experimentos onde o tempo de vida assume valores discretos, os seguintes algoritmos foram avaliados: (i) IBk (*Instance-Based*) [67] representando o algoritmo k-NN; (ii) J48 [34] referente ao algoritmo de árvore de decisão; (iii) *Naive Bayes* [66] como correspondente do paradigma bayesiano; (iv) *Random Forest* [35] representando um método *ensemble*; e (v) SMO [68] como representante do algoritmo SVM para classificação.

O intervalo de distribuição do tempo de vida dos *pull requests* em projetos *open-source* é bastante amplo. Enquanto muitos *pull requests* são integrados em poucos minutos, vários demoram semanas ou até meses. Com isso, a tarefa de realizar previsões utilizando algoritmos de regressão com valores numéricos para o atributo alvo pode se tornar propensa a erros de grande magnitude. Nesse contexto, em vez de prever um valor numérico, que pode apresentar uma alta taxa de erro, a equipe principal de desenvolvimento pode estar interessada em uma previsão mais acurada do tempo de vida. Assim, a previsão pode ser realizada utilizando algoritmos de classificação com valores discretos representados por intervalos de tempo.

Estratégias de seleção de atributos foram empregadas para identificar se existe um subconjunto de atributos com melhor desempenho e, consequentemente, capaz de melhorar a capacidade preditiva dos algoritmos. Para isso, três estratégias sugeridas por

Hall e Holmes [72] foram avaliadas: CFS (*Correlation-based Feature Selection*) [71], IGAR (*Information Gain Attribute Ranking*) [3] e Relief-F [80].

A estratégia CFS e IGAR já foram explicadas na Subseção 4.3.4. A estratégia Relief-F têm como característica a identificação de um subconjunto de atributos de tamanho fixo a partir de um ranqueamento (estratégia *Ranker*) dos atributos, definido a partir da capacidade individual dos atributos de determinar a classe [72]. O tamanho do subconjunto é definido como parâmetro de entrada da estratégia Relief-F.

5.2.4 Processo Experimental

A avaliação dos algoritmos de regressão e classificação foi realizada com o método adaptado para bases com dependência temporal, *traning-test sliding validation*, proposto na subseção 4.2.4 do Capítulo 4.

Para verificar o desempenho preditivo dos algoritmos de regressão foram utilizados o coeficiente de correlação não-paramétrico de *Spearman* (ρ) e medidas de erro médio: RMSE (*Root Mean Squared Error*) e NRMSE (*Normalized Root Mean Squared Error*). Já para os algoritmos de classificação, a acurácia foi utilizada para medir e comparar a capacidade preditiva.

A medida de ganho normalizado definida na Subseção 4.2.4 foi utilizada para avaliar o quanto o algoritmo melhora o desempenho preditivo de uma estratégia *baseline* .

Para identificar a significância estatística dos resultados [74], foram adotados o teste não-paramétrico de *Friedman* [75] e o pós-teste de *Nemenyi* [76, 77]. Em ambos os testes estatísticos o nível de significância foi de 0,05. Os resultados dos testes estatísticos foram obtidos com a ferramenta R.

Os experimentos foram realizados utilizando a linguagem de programação Java em conjunto com a API da ferramenta Weka como base para a execução dos algoritmos de regressão e classificação.

5.3 Resultados dos Experimentos

Esta seção apresenta os resultados e as discussões sobre os experimentos de previsão do tempo de vida e está organizada da seguinte forma. A Subseção 5.3.1 contém os resultados alcançados com o modelo de regressão linear utilizando os atributos do Conjunto D. A Subseção 5.3.2 apresenta os resultados obtidos com outros algoritmos de regressão con-

siderando a seguinte discretização do atributo alvo: de um até 60 minutos, de uma até 24 horas, de um até sete dias, de uma até quatro semanas e mais de um mês. A Subseção 5.3.3 descreve o procedimento de ajuste dos parâmetros de entrada para alguns algoritmos de classificação. Na Subseção 5.3.4, são apresentadas as acurácias obtidas com os algoritmos de classificação utilizando o Conjunto E. A Subseção 5.3.6 compara os resultados atingidos pelo algoritmo *Random Forest* considerando os dois conjuntos. A Subseção 5.3.5 mostra os resultados alcançados com a utilização de estratégias de seleção de atributos. A Subseção 5.3.6 compara os resultados e apresenta os ganhos normalizados obtidos a partir das acurácias em cada projeto. Por fim, a Subseção 5.3.7 apresenta a importância dos atributos para o cenário de tempo de vida em cinco projetos.

5.3.1 Previsão do Tempo de Vida com Regressão Linear

O primeiro experimento teve como objetivo avaliar o poder preditivo dos atributos propostos por Yu *et al.* [8], que compõem o Conjunto D. Para isso, o algoritmo de regressão linear, utilizado pelos autores, foi avaliado com a metodologia *training-test sliding validation*. Nesse experimento, o valor do atributo alvo, que representa o tempo de vida dos *pull requests*, possui valores numéricos, em minutos.

Na Tabela 5.4, para cada projeto, a coluna Linguagem contém a linguagem de programação predominante entre os arquivos dos projetos. A coluna Spearman representa os resultados para o coeficiente de correlação de *Spearman* entre o tempo de vida real e o valor previsto pelo algoritmo. As colunas RMSE e NRMSE apresentam os valores de erro médio e erro médio normalizado obtidos pelo algoritmo *Linear Regression* (LR). Os maiores valores para o coeficiente de *Spearman* indicam maior correlação entre os *rankings* dos valores reais e previstos. Os menores valores para as medidas RMSE e NRMSE representam maior índice de acerto do modelo de regressão.

Apesar de alcançar valores acima de 0,7 para o coeficiente de correlação em alguns projetos, os valores de RMSE e NRMSE apresentados pelos modelos de regressão linear são muito altos. O menor erro médio foi obtido no projeto rosdistro: RMSE = 4.843. Isso significa que a diferença média entre os valores reais e os valores previstos é de 4.843 minutos. A avaliação dos modelos lineares apresentou resultados de erro médio bem limitados, o que motivou a avaliação de outros algoritmos de regressão com atributos do Conjunto D.

Tabela 5.4: Resultados obtidos com regressão linear utilizando o Conjunto D.

		Linear Regression		
Linguagem	**Projeto**	**Spearman**	**RMSE**	**NRMSE**
Java	candlepin	0,406	9.677	397
	hazelcast	0,426	105.717	1.408
	infinispan	0,457	15.824	953
	netty	0,416	159.372	6.331
	okhttp	0,319	65.202	20.648
JavaScript	angular.js	0,712	105.052	5.286
	appium	0,370	12.579	1.184
	brackets	0,693	63.178	6.475
	jquery	0,444	57.530	3.725
	meteor	0,606	203.590	14.312
	node	0,779	29.079	1.928
Python	boto	0,512	67.149	3.519
	django	0,734	62.940	5.535
	ipython	0,694	130.832	2.104
	kuma	0,498	12.890	550
	pandas	0,645	60.869	4.719
	rosdistro	0,220	4.843	226
	scikit-learn	0,682	68.864	4.404
Ruby	bundler	0,425	344.875	16.959
	diaspora	0,499	72.067	3.783
	jekyll	0,506	83.058	4.114
	katello	0,574	18.257	1.107
	metasploit-framework	0,683	65.815	4.573
	puppet	0,713	47.522	5.214
	vagrant	0,437	513.906	11.284
Scala	akka	0,563	20.452	1.299
	marathon	0,413	40.960	1.534
	scala	0,425	157.333	803
	scala-ide	0,385	38.955	139
	scala-js	0,602	13.304	843

5.3.2 Previsão do Tempo de Vida com Algoritmos de Regressão

Nesta subseção, avaliamos o desempenho de outros algoritmos de regressão na previsão do tempo de vida de *pull requests* utilizando o Conjunto *D*. Para isso, foram considerados os seguintes algoritmos: *Random Forest*, M5P e SMOReg com o *kernel Polynomial*. Os algoritmos de regressão foram avaliados utilizando a metodologia *training-test sliding validation* com os atributos do Conjunto *D*. Nesse experimento, o valor do tempo de vida dos *pull requests* também assume valores numéricos em minutos.

A Tabela 5.5 apresenta, para cada projeto, o coeficiente de correlação de *Spearman*, as medidas de erro médio e erro médio normalizado obtidas com os algoritmos de regressão nas colunas RMSE e NRMSE. Em cada projeto, os melhores resultados para a medida NRMSE estão marcados em negrito.

Tabela 5.5: Resultados obtidos com algoritmos de regressão utilizando o Conjunto D.

		M5P			Random Forest			SMOReg		
Linguagem	**Projeto**	**Spearman**	**RMSE**	**NRMSE**	**Spearman**	**RMSE**	**NRMSE**	**Spearman**	**RMSE**	**NRMSE**
Java	candlepin	0,413	60.055	1.162	0,360	7.610	456	0,408	9.043	**264**
	hazelcast	0,509	114.414	790	0,390	11.649	512	0,418	34.820	**475**
	infinispan	0,462	14.315	**360**	0,424	12.735	521	0,484	13.313	535
	netty	0,422	88.996	8.263	0,415	37.360	2.174	0,459	47.325	**771**
	okhttp	0,391	110.273	19.296	0,381	12.679	**316**	0,410	21.192	7.753
JavaScript	angular.js	0,714	78.579	**1.391**	0,550	80.849	2.172	0,730	105.671	2.527
	appium	0,397	26.343	560	0,407	6.515	443	0,408	7.244	**148**
	brackets	0,707	19.350	**395**	0,560	23.958	1.104	0,739	35.091	1.246
	jquery	0,458	54.675	2.238	0,406	45.896	2.091	0,479	47.834	**1.226**
	meteor	0,627	64.203	2.618	0,476	75.380	**2.613**	0,632	180.619	4.103
	node	0,797	75.082	**206**	0,519	25.694	1.126	0,798	28.023	961
Python	boto	0,545	48.022	**1.051**	0,351	55.895	1.590	0,521	55.970	1.885
	django	0,700	88.260	**1.805**	0,622	49.376	1.938	0,738	68.398	2.447
	ipython	0,671	202.679	**518**	0,604	23.346	688	0,714	93.862	770
	kuma	0,497	87.936	344	0,449	9.457	345	0,502	9.924	**187**
	pandas	0,663	86.800	**837**	0,621	39.438	999	0,674	57.813	2.259
	rosdistro	0,218	25.053	424	0,211	2.318	124	0,229	2.215	**35**
	scikit-learn	0,688	42.683	**562**	0,566	67.427	1.939	0,685	71.519	3.053
Ruby	bundler	0,485	195.259	**2.493**	0,390	87.145	3.563	0,441	437.675	16.845
	diaspora	0,518	57.068	**1.103**	0,422	45.350	1.853	0,534	59.010	1.385
	jekyll	0,528	78.415	2.049	0,459	72.924	2.090	0,540	77.779	**1.332**
	katello	0,550	414.327	1.215	0,460	14.089	569	0,557	15.354	**332**
	metasploit-framework	0,660	124.126	**1.207**	0,588	34.491	1.513	0,686	52.340	1.720
	puppet	0,681	62.011	4.568	0,566	33.806	2.699	0,722	39.184	**1.822**
	vagrant	0,461	138.044	**4.730**	0,409	73.786	5.665	0,451	145.508	6.138
Scala	akka	0,566	18.430	1.047	0,485	17.255	812	0,571	18.418	**449**
	marathon	0,436	219.106	818	0,460	14.686	**504**	0,461	18.528	759
	scala	0,563	56.204	564	0,479	10.980	608	0,600	12.637	**519**
	scala-ide	0,465	165.888	825	0,458	18.058	**470**	0,457	41.807	544
	scala-js	0,395	13.709	64	0,414	8.060	**37**	0,468	12.567	64

Na grande maioria dos casos, as medidas de erro médio obtidas pelos algoritmos M5P, *Random Forest* e SMOReg são melhores que as apresentadas pelo algoritmo *Linear Regression* utilizado por Yu *et al.* [8]. Considerando os valores da medida NRMSE, os algoritmos M5P e SMOReg atingiram os melhores desempenhos, obtendo, junto, o melhor resultado em 25 dos 30 projetos. O algoritmo *Random Forest* conseguiu o melhor resultado em quatro projetos e o algoritmo *Linear Regression* obteve o melhor resultado em apenas um projeto. Os menores NRMSE foram obtidos nos projetos appium (148), node (206), rosdistro (35) e scala-js (37).

Apesar de os algoritmos avaliados nesta tese (M5P, *Random Forest* e SMOReg) terem melhorado os resultados da estratégia adotada por Yu *et al.* [8] (*Linear Regression*), os erros ainda estão muito altos, o que limita a utilização da previsão. No processo de tomada de decisão, pode ser mais simples e útil para solicitantes e integradores estimar o tempo de vida de *pull requests* com valores discretos, utilizando cinco intervalos de tempo na ordem de: minutos (de um até 60 minutos), horas (de uma até 24 horas),

dias (de um até sete dias), semanas (de uma até quatro semanas) e meses (mais de um mês). Com isso, pretende-se obter modelos preditivos mais efetivos e úteis. Dessa forma, a partir da próxima subseção, serão utilizados e avaliados algoritmos de classificação com os Conjuntos D e E.

5.3.3 Ajuste de Parâmetros dos Algoritmos de Classificação

Experimentos preliminares foram conduzidos com o intuito de calibrar os parâmetros de entrada dos algoritmos IBk, SMO e *Random Forest*. Esses experimentos foram realizados utilizando os atributos do Conjunto E em 10 projetos, dois representantes com o menor número de *pull requests* em cada linguagem: candlepin e okhttp (Java); appium e meteor (JavaScript); boto e scikit-learn (Python); bundler e vagrant (Ruby); scala-ide e scala-js (Scala). Esses projetos não foram posteriormente utilizados para avaliar e comparar o desempenho dos algoritmos.

Foram calibrados os seguintes parâmetros de entrada: o valor de k (número de vizinhos) no algoritmo IBk, o *kernel* do algoritmo SMO (*Polynomial*, Puk e RBF) e o número de árvores construídas no algoritmo *Random Forest*. Os algoritmos de classificação foram avaliados pela metodologia *training-test sliding validation* com valores discretos para o tempo de vida.

Nos próximos experimentos, cada tabela apresenta quatro medidas para indicar o desempenho dos algoritmos: (i) a Média das Acurácias, definida pela divisão entre a soma das acurácias de cada algoritmo e a quantidade de projetos; (ii) a média dos ganhos normalizados, ou simplesmente, Média dos Ganhos, que representa a soma dos ganhos normalizados (melhoria da acurácia comparada com a classe majoritária) obtidos pelas acurácias de cada projeto dividida pelo número de projetos; (iii) a Média dos Rankings, calculada da seguinte forma: para cada projeto, cada algoritmo recebe um número de posto ordenado, 1 para melhor acurácia, 2 para segunda melhor e n para pior acurácia, onde n é o número de algoritmos; em seguida, os postos de cada algoritmo são somados e o resultado é dividido pelo número de projetos; quanto menor a média, melhor o desempenho do algoritmo; e (iv) o Número de Vitórias, definido como o número de vezes em que o algoritmo obteve a melhor acurácia, considerando todos os projetos.

A Tabela 5.6 mostra os resultados obtidos para o parâmetro k do IBk variando com valores ímpares de 1 até 29. O algoritmo IBk usando o valor k igual a 23 obteve os melhores resultados considerando todas as medidas.

Tabela 5.6: Resultados obtidos com a variação do valor de k no IBk.

	Valor de k											
Medidas	**1**	**3**	**5**	**7**	**9**	**11**	...	**21**	**23**	**25**	**27**	**29**
Média das Acurácias	35,06%	35,20%	36,11%	36,41%	37,14%	37,20%		38,30%	**38,68%**	38,16%	37,90%	37,94%
Média dos Ganhos	-1,01%	-0,76%	2,12%	2,63%	4,28%	3,68%		4,73%	**5,86%**	4,75%	4,38	4,03%
Média dos Rankings	13,70	13,90	11,75	11,20	9,40	9,70		5,00	**2,70**	4,10	5,75	6,20
Número de Vitórias	0	0	0	0	0	0		0	**4**	0	0	1

A Tabela 5.7 apresenta os resultados alcançados pelo algoritmo SMO usando os três *kernels* (*Polynomial*, Puk e RBF). O *kernel Polynomial* apresentou o melhor desempenho considerando todas as medidas.

Tabela 5.7: Resultados obtidos com a variação do *kernel* no SMO.

	kernel SMO		
Medidas	**Polynomial**	**Puk**	**RBF**
Média das Acurácias	**40,19%**	38,41%	36,76%
Média dos Ganhos	**4,99%**	2,20%	-1,02%
Média dos Rankings	**1,20**	2,00	2,80
Número de Vitórias	**9**	1	0

Os resultados para a variação do número de árvores usadas pelo *Random Forest* são apresentados na Tabela 5.8. Para esse parâmetro, foram utilizadas 50 árvores como valor mínimo e 500 para o máximo, com incremento de 50 árvores. O valor 300 conseguiu a melhor acurácia em três projetos e atingiu os melhores resultados para a média das acurácias, dos ganhos e dos *rankings*.

Tabela 5.8: Resultados obtidos com a variação do número de árvores no *Random Forest*.

	Número de árvores									
Medidas	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Média das Acurácias	42,64%	42,78%	43,09%	43,00%	43,03%	**43,20%**	43,01%	42,91%	42,93%	42,90%
Média dos Ganhos	6,21%	6,75%	7,46%	7,35%	7,27%	**7,50%**	7,24%	7,07%	7,00%	7,13%
Média dos Rankings	7,40	6,65	4,95	5,10	4,95	**3,25**	5,15	5,75	5,70	6,10
Número de Vitórias	0	2	**3**	0	1	**3**	0	1	0	0

Com base nos resultados do ajuste de parâmetros, o valor de k igual a 23, o *kernel Polynomial* e 300 árvores de decisão foram utilizados como parâmetros de entrada para os algoritmos IBk, SMO e *Random Forest* no experimento de comparação entre os algoritmos de classificação utilizando os atributos do Conjunto E, apresentado na seção seguinte.

5.3.4 Previsão do Tempo de Vida com Classificação

O experimento desta subseção consiste em comparar a capacidade preditiva dos algoritmos de classificação utilizando os atributos preditivos do Conjunto *E* para prever o tempo de vida de *pull requests*. Os parâmetros de entrada dos algoritmos IBk, *Random Forest* (RF) e SMO receberam os valores definidos pelo experimento anterior. Os algoritmos J48 e *Naive Bayes* foram avaliados com valores *default* para os parâmetros de entrada. Foram utilizados os 20 projetos que não participaram do experimento de ajuste de parâmetros. Os algoritmos de classificação foram avaliados pela metodologia *training-test sliding validation* e o tempo de vida dos *pull requests* assumiu cinco valores discretos ($C_1 \leq$ 60 minutos $< C_2 \leq$ 24 horas $< C_3 \leq$ 7 dias $< C_4 \leq$ 4 semanas $< C_5$).

Na Tabela 5.9, a coluna Baseline representa, para cada projeto, a porcentagem da classe majoritária, ou seja, o intervalo de tempo que possui a maior quantidade de *pull requests* integrados. As colunas abaixo do título Acurácia apresentam, para cada projeto, as acurácias obtidas com os algoritmos de classificação na previsão do tempo de vida utilizando o Conjunto *E*. Além disso, para cada projeto e para cada medida, o melhor resultado está destacado em negrito.

Tabela 5.9: Resultados obtidos com algoritmos de classificação utilizando o Conjunto E

			Acurácia (%)				
Linguagem	**Projetos**	**Baseline**	**IBk**	**J48**	**Naive Bayes**	**RF**	**SMO**
Java	hazelcast	47,01	47,52	41,55	41,12	**47,98**	46,28
	infinispan	35,37	38,60	36,89	32,09	**44,08**	43,89
	netty	38,88	36,85	38,37	31,14	**42,70**	41,36
JavaScript	angular.js	26,91	30,97	**43,96**	33,11	42,44	40,57
	brackets	31,03	32,90	40,39	31,26	**41,19**	39,54
	jquery	26,30	27,61	28,59	27,39	31,30	**32,28**
	node	34,80	34,94	**50,70**	31,91	48,64	45,42
Python	django	35,37	34,67	43,28	28,97	**46,74**	45,33
	ipython	31,83	31,14	38,01	30,04	**41,96**	40,51
	kuma	39,08	39,81	36,77	31,99	43,51	**44,13**
	pandas	32,39	34,98	42,79	37,85	**44,64**	43,01
	rosdistro	**71,52**	70,49	61,77	54,49	68,39	70,99
Ruby	diaspora	35,30	30,68	36,05	29,12	**38,06**	37,95
	jekyll	34,75	30,94	31,87	31,23	**37,42**	36,32
	katello	38,53	39,75	37,25	31,39	44,54	**45,72**
	metasploit-framework	31,34	32,14	39,25	31,29	**43,20**	42,59
	puppet	27,05	29,37	35,43	27,85	**38,17**	36,64
Scala	akka	38,11	39,58	38,92	35,05	44,75	**47,22**
	marathon	37,01	36,52	33,93	33,04	**37,73**	36,92
	scala	38,28	40,82	39,82	23,89	47,41	**47,62**
	Média das Acurácias		37,01%	39,78%	32,71%	**43,74%**	43,21%
	Média dos Ganhos		-0,11%	3,95%	-9,94%	**10,82%**	10,00%
	Média dos Rankings		3,65	3,15	4,85	**1,45**	1,90
	Número de Vitórias		0	2	0	**12**	6

O algoritmo RF apresentou os melhores resultados considerando as quatro medidas. As acurácias obtidas pelo algoritmo RF proporcionaram média de ganho normalizado de 10,82% e obteve a melhor acurácia em 12 dos 20 projetos. Com base nesses resultados, o algoritmo RF foi selecionado para realizar o experimento empregando estratégias de seleção de atributos sobre o Conjunto E.

5.3.5 Melhorando o Desempenho com Seleção de Atributos

Com o intuito de avaliar se existe um subconjunto de atributos capaz de melhorar o desempenho dos classificadores, os atributos do Conjunto E foram submetidos a estratégias de seleção de atributos. Três estratégias foram utilizadas: CFS, IGAR e Relief-F. As estratégias IGAR e Relief-F têm como parâmetro de entrada o tamanho do subconjunto de atributos que deverá ser retornado. Assim, um experimento foi realizado para calibrar esse parâmetro. Utilizando os 20 projetos do experimento anterior, as estratégias foram avaliadas, considerando em todas as bases de treinamento, subconjuntos de 1 até 40 atributos.

A Tabela 5.10 mostra os resultados obtidos pela estratégia Relief-F. O melhor desempenho foi alcançado pelos subconjuntos com os 33 atributos preditivos mais relevantes. Algumas colunas da tabela, que representam outras quantidades de atributos, apresentaram resultados inferiores e foram suprimidas para melhorar a apresentação dos resultados.

Tabela 5.10: Resultados obtidos para a estratégia Relief-F.

	Número de Atributos										
Medidas	**1**	**2**	**...**	**20**	**21**	**22**	**...**	**32**	**33**	**34**	**...**	**40**
Média das Acurácias	37,57%	37,67%		43,80%	43,75%	43,75%		44,15%	**44,27%**	44,02%		44,02%
Média dos Ganhos	-0,18%	-0,14%		10,78%	10,72%	10,73%		11,36%	**11,57%**	11,13%		11,18%
Média dos Rankings	37,55	38,98		14,53	15,40	16,25		12,25	**9,73**	11,85		11,65
Número de Vitórias	0	0		1	0	0		2	**3**	2		1

A Tabela 5.11 mostra os resultados obtidos pela estratégia IGAR. O melhor desempenho foi alcançado pelos subconjuntos com os 33 atributos preditivos mais relevantes. Novamente, algumas colunas da tabela foram suprimidas para melhorar a apresentação dos resultados.

Subconjuntos com 33 atributos para as estratégias Relief-F e IGAR apresentaram os melhores resultados. Portanto, esses valores serão utilizados como parâmetros de entrada no próximo experimento que compara o desempenho das estratégias de seleção de atributos.

Tabela 5.11: Resultados obtidos para a estratégia IGAR.

	Número de Atributos										
Medidas	**1**	**2**	**...**	**20**	**21**	**22**	**...**	**32**	**33**	**34**	**...**	**40**
Média das Acurácias	38,22%	39,13%		43,81%	43,88%	43,90%		45,04%	**45,28%**	44,98%		44,93%
Média dos Ganhos	1,42%	3,23%		11,20%	11,30%	11,31%		12,74%	**13,10%**	12,63%		12,55%
Média dos Rankings	38,68	38,93		19,88	18,63	18,10		8,38	**6,20**	8,73		8,78
Número de Vitórias	0	0		0	0	1		2	**5**	2		3

A Tabela 5.12 mostra, para cada projeto, as acurácias obtidas com o algoritmo *Random Forest* após a utilização das estratégias de seleção de atributos no Conjunto *E*. A coluna Baseline representa o percentual da classe majoritária. Além disso, para cada projeto e para cada medida, o melhor resultado está destacado em negrito.

Tabela 5.12: Resultados obtidos com seleção de atributos utilizando o Conjunto E.

			Acurácia (%) Random Forest		
Linguagem	**Projetos**	**Baseline**	**CFS**	**IGAR (33)**	**Relief-F (33)**
Java	hazelcast	47,01	47,28	47,58	**47,95**
	infinispan	35,37	41,80	**45,57**	42,06
	netty	38,88	40,11	**44,62**	43,67
JavaScript	angular.js	26,91	39,24	**45,46**	45,08
	brackets	31,03	41,81	**44,84**	43,92
	jquery	26,30	30,43	**31,96**	31,52
	node	34,80	50,50	52,95	**53,80**
Python	django	31,83	41,74	**47,38**	46,09
	ipython	31,83	39,69	**44,99**	42,86
	kuma	39,08	41,46	43,88	**43,92**
	pandas	32,39	44,32	**50,46**	47,04
	rosdistro	**71,52**	64,15	67,97	67,11
Ruby	diaspora	35,30	35,88	39,90	**40,24**
	jekyll	34,75	33,60	**38,86**	36,36
	katello	37,90	58,87	**60,36**	58,38
	metasploit-framework	31,34	41,09	**44,37**	42,98
	puppet	27,05	35,63	39,52	**39,83**
Scala	akka	38,11	43,16	**44,97**	43,85
	marathon	37,01	**38,54**	37,81	37,41
	scala	38,28	42,85	**47,86**	46,07
	Média das Acurácias		41,68	**48,28**	44,27
	Média dos Ganhos		7,58	**13,10**	11,57
	Média dos Rankings		2,90	**1,30**	1,80
	Número de Vitórias		1	**13**	5

O algoritmo *Random Forest* utilizando a estratégia de seleção de atributos IGAR, com os 33 atributos mais relevantes, obteve resultados superiores em relação às demais estratégias de seleção de atributos, considerando todas as medidas.

5.3.6 Comparando o Desempenho das Diferentes Abordagens

O experimento desta subseção tem o objetivo de comparar o desempenho de três abordagens para prever o tempo de vida de *pull requests*. Cada abordagem considera a combinação de um algoritmo de classificação e um conjunto de atributos. A **Abordagem A**, representa o algoritmo *Random Forest*, com o valor fixo de 300 árvores de decisão como parâmetro de entrada, utilizando os atributos preditivos do Conjunto D propostos por Yu *et al.* [8]. A **Abordagem B**, representa a combinação do algoritmo *Random Forest*, também com 300 árvores de decisão como parâmetro de entrada, utilizando os atributos preditivos do Conjunto E, proposta nesta tese. A **Abordagem C**, representa o algoritmo *Random Forest*, novamente com 300 árvores de decisão como parâmetro de entrada, e a utilização da estratégia de seleção de atributos IGAR no Conjunto E.

Na Tabela 5.13, a sumarização dos resultados de cada abordagem é apresentada. Para cada medida, o melhor resultado está destacado em negrito.

Tabela 5.13: Resultados obtidos pelas Abordagens A, B e C.

	Abordagens		
Medidas	**Abordagem A**	**Abordagem B**	**Abordagem C**
Média das Acurácias	41,48%	43,74%	**45,28%**
Média dos Ganhos	8,94%	12,40%	**14,68%**
Média dos Rankings	2,90	2,00	**1,10**
Número de Vitórias	0	2	**18**

Observa-se que os subconjuntos de atributos da Abordagem C atingiram os melhores resultados em todas as medidas. Na comparação entre as três abordagens, considerando os valores de acurácia obtidos para cada um dos 20 projetos, o teste de *Friedman* rejeitou a hipótese nula com $p\text{-}valor = 9,214 \times 10^{-8}$, indicando que existe uma diferença significativa entre as acurácias das abordagens. No pós-teste de *Nemenyi*, os resultados obtidos com a Abordagem C são estatisticamente melhores quando comparados com as acurácias da Abordagem A ($p\text{-}valor = 3,8 \times 10^{-8}$) e da Abordagem B ($p\text{-}valor = 0,012$).

Para detalhar um pouco mais a avaliação das abordagens, a Tabela 5.14 apresenta o ganho normalizado para cada projeto. Os resultados apresentados mostram que a utilização da estratégia de seleção de atributos IGAR no Conjunto E (Abordagem C) proporcionou ganhos normalizados superiores quando comparados com as demais abordagens. Por exemplo, nos projetos angular, brackets, node e pandas, os ganhos normalizados foram superiores a 20%. Além disso, a Abordagem C obteve o melhor ganho normalizado em 18 dos 20 projetos. Esses resultados mostram que a identificação de um subconjunto apropriado

de atributos trouxe um aumento da capacidade de prever o tempo de vida na maioria dos projetos avaliados.

Tabela 5.14: Ganho normalizado obtido pelas abordagens, por projeto.

		Ganho Normalizado (%)		
Linguagem	**Projetos**	**Abordagem A**	**Abordagem B**	**Abordagem C**
Java	hazelcast	-3,68	**1,83**	1,08
	infinispan	6,48	13,48	**15,68**
	netty	3,17	6,25	**9,39**
JavaScript	angular.js	20,89	21,25	**25,38**
	brackets	13,53	14,73	**20,02**
	jquery	5,02	6,78	**7,68**
	node	24,03	21,23	**27,84**
Python	django	10,82	17,59	**18,58**
	ipython	10,09	14,86	**19,30**
	kuma	-1,77	7,27	**7,88**
	pandas	19,88	18,12	**26,73**
	rosdistro	-11,21	**-4,38**	-4,96
Ruby	diaspora	0,12	4,27	**7,11**
	jekyll	-4,63	4,09	**6,30**
	katello	4,20	9,78	**9,86**
	metasploit-framework	14,30	17,27	**18,98**
	puppet	13,75	15,24	**17,09**
Scala	akka	8,76	10,73	**11,08**
	marathon	-0,16	1,14	**1,27**
	scala	9,02	14,79	**15,52**

5.3.7 Atributos Relevantes na Previsão do Tempo de Vida

Nesta subseção, identificamos os atributos do Conjunto E mais relevantes na previsão do tempo de vida. Para isso, os projetos que obtiveram o melhor ganho normalizado em cada linguagem foram selecionados: infinispan, metasploit-framework, node, pandas e scala. Para cada base de treinamento utilizada na avaliação dos classificadores, os atributos foram ordenados de acordo com o valor calculado pela medida Ganho de Informação. O primeiro posto representa o atributo com maior valor e o último posto representa o atributo com menor valor. Em seguida, os postos de cada atributo foram somados e o resultado divido pela quantidade de bases de treinamento avaliadas. Esse valor representa a média dos postos de cada atributo.

A Tabela 5.15 apresenta a relevância dos atributos nos cinco projetos avaliados. Cada projeto possui duas colunas que representam o ranqueamento de todos os atributos. A coluna Atributo contém o identificador de cada atributo e está ordenada de acordo com o valor da coluna Média, que representa a média dos postos considerando os valores de Ganho de Informação para cada atributo. Com isso, os atributos mais importantes se encontram na parte de cima e os menos importantes na parte de baixo da tabela. As

colunas abaixo do título Legenda contêm o identificador e nome de cada atributo apenas para facilitar a identificação dos mesmos.

Tabela 5.15: Importância dos atributos.

	Projetos										Legenda	
	infinispan		metasploit		node		pandas		scala			
Posto	Atributo	Média	Atributo	Média	Atributo	Média	Atributo	Média	Atributo	Média	Id	Nome
1	*D*.4	1,00	*D*.4	1,00	*D*.4	2,67	*D*.4	1,00	*D*.4	1,00	*D*.1	ProjectAge
2	*D*.14	6,94	*D*.14	4,88	*D*.7	5,74	*D*.14	2,77	*D*.14	3,87	*D*.2	TeamSize
3	*D*.15	9,33	*D*.7	5,04	*D*.14	5,86	*D*.6	3,72	*D*.15	8,89	*D*.3	AreaHotness
4	*E*.16	9,98	*D*.6	5,84	*E*.5	7,88	*D*.7	7,90	*E*.16	9,06	*D*.4	Workload
5	*D*.12	10,44	*E*.19	7,19	*E*.9	8,80	*E*.12	9,70	*E*.24	9,89	*D*.5	DescriptionSize
6	*D*.7	11,89	*E*.13	10,07	*D*.6	8,83	*E*.19	10,76	*D*.12	10,18	*D*.6	NumCommits
7	*E*.23	13,42	*E*.17	10,73	*E*.12	9,00	*E*.18	11,33	*E*.23	10,42	*D*.7	NumChurn
8	*D*.6	13,50	*E*.18	11,92	*E*.8	9,03	*E*.17	11,36	*D*.7	12,32	*D*.10	UseCI
9	*E*.24	14,56	*E*.5	13,47	*E*.7	11,00	*E*.13	12,90	*D*.6	14,32	*D*.11	TestInclusion
10	*E*.12	14,88	*D*.15	14,46	*D*.12	11,13	*E*.21	13,14	*D*.11	16,69	*D*.12	FridayEffect
11	*E*.11	16,16	*E*.20	14,99	*E*.23	12,23	*D*.17	14,7	*E*.12	17,00	*D*.13	IssueTag
12	*D*.11	16,36	*D*.16	15,32	*E*.16	12,57	*E*.20	14,81	*E*.22	17,79	*D*.14	MentionTag
13	*E*.10	17,50	*D*.17	15,67	*E*.24	12,66	*D*.5	16,37	*E*.11	18,39	*D*.15	RequesterType
14	*E*.5	17,98	*E*.21	16,23	*D*.15	12,70	*D*.15	17,01	*D*.3	18,47	*D*.16	RequesterAcceptRate
15	*E*.22	18,60	*E*.8	16,28	*E*.11	16,39	*E*.16	18,26	*D*.13	19,33	*D*.17	RequesterNumFollowers
16	*E*.19	18,94	*D*.5	16,36	*D*.3	16,80	*E*.8	18,36	*E*.5	19,36	*E*.1	PrevPullProject
17	*D*.3	19,43	*E*.24	17,98	*D*.11	16,96	*E*.9	18,69	*E*.1	19,69	*E*.2	ExternalContribs
19	*E*.6	19,46	*E*.16	18,96	*D*.13	17,78	*E*.7	18,81	*E*.6	19,97	*E*.3	Sloc
20	*E*.9	19,79	*E*.23	19,34	*E*.6	19,42	*E*.23	19,49	*E*.2	20,16	*E*.4	Stars
21	*E*.7	19,83	*D*.12	20,84	*E*.10	20,57	*E*.22	19,70	*E*.10	20,24	*E*.5	NumAddedFiles
22	*D*.13	19,84	*E*.22	21,59	*E*.22	20,86	*D*.12	20,48	*E*.7	20,90	*E*.6	NumDeletedFiles
23	*E*.3	20,38	*E*.9	23,90	*E*.1	22,29	*D*.16	20,81	*E*.8	20,96	*E*.7	NumModifiedFiles
24	*E*.8	20,92	*D*.11	23,94	*E*.2	26,01	*E*.24	22,14	*E*.9	21,22	*E*.8	NumChangedFiles
25	*E*.18	21,74	*E*.10	24,58	*E*.15	26,62	*E*.11	24,04	*E*.17	22,82	*E*.9	NumSrcFiles
26	*D*.10	23,14	*E*.7	24,60	*E*.18	27,03	*E*.5	24,49	*D*.17	23,32	*E*.10	NumDocFiles
27	*E*.1	23,33	*E*.6	25,20	*E*.19	27,10	*E*.10	25,07	*E*.18	25,74	*E*.11	NumOtherFiles
28	*E*.13	25,04	*E*.11	25,04	*E*.20	27,13	*E*.6	25,28	*E*.19	25,74	*E*.12	NumTestChurn
29	*E*.2	25,54	*D*.3	26,21	*E*.17	27,68	*E*.14	26,09	*E*.21	26,06	*E*.13	NumPrevPullRequester
30	*E*.20	26,12	*E*.15	26,27	*D*.17	28,46	*D*.11	26,64	*E*.15	26,10	*E*.14	RequesterNumFollowing
31	*E*.14	26,34	*E*.12	26,53	*D*.1	29,53	*E*.15	27,59	*D*.1	26,50	*E*.15	RequesterAge
32	*E*.17	27,41	*E*.14	29,11	*E*.14	30,21	*D*.10	31,11	*E*.14	27,54	*E*.16	WatcherProject
33	*E*.21	29,26	*E*.3	29,93	*E*.13	31,91	*D*.3	32,17	*E*.4	27,87	*E*.17	IssueEvents
34	*E*.15	29,49	*E*.4	30,99	*E*.3	32,14	*D*.2	32,29	*E*.20	28,30	*E*.18	IssueComments
35	*D*.1	29,92	*D*.2	31,49	*D*.16	32,48	*E*.3	32,49	*D*.16	28,68	*E*.19	PullEvents
36	*E*.4	30,00	*D*.10	32,17	*D*.5	33,24	*E*.4	32,78	*E*.3	29,12	*E*.20	PullComments
37	*D*.16	30,77	*E*.1	33,04	*E*.4	33,34	*E*.1	32,83	*D*.10	31,18	*E*.21	NumPrevCommits
38	*D*.17	32,51	*D*.1	33,72	*E*.21	34,62	*D*.13	33,20	*D*.2	31,94	*E*.22	NumCommitComments
39	*D*.2	34,83	*D*.13	34,30	*D*.2	36,49	*D*.1	33,32	*D*.15	32,82	*E*.23	RequesterFollowsCT
40	*D*.5	36,04	*E*.2	34,82	*D*.10	38,34	*E*.2	33,78	*D*.5	38,58	*E*.24	CTFollowsRequester

Nos cinco projetos apresentados, os atributos Workload (*D*.4) e MentionTag (*D*.14) aparecem entre as três melhores médias. Por outro lado, alguns atributos sobre as características do projeto, tais como, ProjectAge (*D*.1), TeamSize (*D*.2), PrevPullProject (*E*.1) e ExternalContribs (*E*.2) estão entre os que pouco contribuem para prever o valor do tempo de vida.

Alguns atributos propostos no Conjunto *E* se mostraram relevantes para determinar o

tempo de vida como, por exemplo: NumAddedFiles (E.5), NumTestChurn (E.12) e WatcherProject (E.16). Nos projetos metasploit e pandas, os atributos que caracterizam interações anteriores dos desenvolvedores no projeto (E.17 até E.22) também demonstraram ser bons indicadores para prever o tempo de vida.

Além disso, algumas informações sobre o tamanho do *pull request* como, por exemplo, os atributos NumCommits (D.6) e NumChurn (D.7) também mostraram ter uma boa correlação com o tempo de vida. O atributo RequesterType (D.15), também parece influenciar o tempo de tratamento de *pull requests*.

5.4 Limitações e Ameaças à Validade dos Resultados

No estudo apresentado, o tempo de vida foi tratado como o tempo entre a submissão e a finalização da revisão do *pull request* que culmina na integração (aceitando ou rejeitando) do código. Dessa forma, uma limitação desse estudo consiste na dificuldade de determinar o tempo que o integrador leva efetivamente realizando a integração de um *pull request*. Portanto, o tempo de vida considerado neste trabalho não é exatamente o tempo de integração do *pull request*. Isso porque o *pull request* pode permanecer aberto por motivos alheios ao processo de integração como, por exemplo, sobrecarga de atividade dos integradores e falta de interação do solicitante.

Destacamos como outra ameaça o fato de os resultados terem sido obtidos com a utilização de dados de projetos *open-source*. Portanto, não é possível garantir que os resultados obtidos também sejam válidos para projetos industriais. No entanto, os projetos selecionados para os experimentos possuem muitos *pull requests* e são desenvolvidos com algumas das linguagens mais populares entre os projetos hospedados no GitHub.

5.5 Considerações Finais

Este capítulo apresentou os experimentos realizados com o objetivo de avaliar e comparar abordagens de previsão do tempo de vida de *pull requests*. Neste cenário, foram explorados 30 projetos *open-source* coletados com a ferramenta GHTorrent, dois diferentes conjuntos de atributos preditivos, representados pelos Conjuntos D e E, algoritmos de regressão (*Linear Regression*, M5P, *Random Forest* e SMOReg) e algoritmos de classificação (IBk, J48, *Naive Bayes*, *Random Forest* e SMO). A Abordagem A é representada pela avaliação do algoritmo *Random Forest* com os atributos do Conjunto D propostos por Yu *et al.* [8]. A

Abordagem B corresponde aos atributos do Conjunto E, proposto nesta tese, e o algoritmo *Random Forest*. A Abordagem C, também proposta nesta tese, representa a avaliação do algoritmo *Random Forest* utilizando os 20 atributos mais relevantes do Conjunto E, de acordo com a seleção de atributos IGAR.

Os resultados obtidos pelos algoritmos de regressão utilizando o Conjunto D apresentaram valores de erro médio e erro médio normalizado muito altos. Por isso, os algoritmos de classificação foram avaliados com valores discretos representados por intervalos de tempo.

Na comparação com a proposta anterior [8], a Abordagem C apresentou os melhores ganhos normalizados, obtendo a melhor acurácia em 18 dos 20 projetos avaliados. A acurácia média foi de 45,28% para prever o tempo de vida de *pull requests*, o que proporcionou um ganho normalizado médio de 14,68% em relação à classe majoritária. Vale destacar que a Abordagem C utiliza o Conjunto E, contendo atributos cuja utilização como atributos preditivos é uma proposta e contribuição desta tese.

Entre os projetos que apresentaram os melhores ganhos normalizados, observamos que alguns atributos propostos no Conjunto E se mostraram relevantes para determinar o tempo de vida de *pull requests* como, por exemplo: NumAddedFiles ($E.5$), NumTestChurn ($E.12$), WatcherProject ($E.16$) e os que caracterizam interações anteriores dos desenvolvedores no projeto ($E.17$ até $E.22$).

Capítulo 6

Conclusão

6.1 Resultados e Contribuições

O modelo de colaboração baseado em *pull requests* vem sendo amplamente adotado no âmbito do desenvolvimento distribuído de software, em especial nos projetos *open-source*, com o objetivo de controlar o processo de colaboração. Devido à grande quantidade de *pull requests* recebida em projetos populares, as equipes de integradores estão sempre preocupadas em minimizar o tempo de integração dos *pull requests* e garantir a qualidade do código que será aceito e integrado ao repositório do projeto. Quando o código enviado pelo *pull request* modifica arquivos importantes do projeto, é sempre interessante que um integrador experiente conduza o processo de integração de forma rápida e eficiente.

Os sistemas de Gerência de Configuração de Software e as ferramentas de hospedagem de código poderiam se beneficiar em oferecer funções capazes de prever, por exemplo: quais são os integradores mais adequados para integrar um *pull request* e o tempo que pode demorar a integração de um *pull request*. Nesse contexto, as principais contribuições desta tese são:

1. **Uma abordagem para prever integradores de *pull requests***, considerando os processos de discretização (utilizando as técnicas *Minimum Description Length* e *Equal Frequency Interval*) e seleção de atributos (300 atributos selecionados utilizando a estratégia *Information Gain Attribute Ranking*) sobre o Conjunto C (contendo os atributos de [65], de [7] e os novos atributos propostos) e a utilização do algoritmo de classificação *Random Forest* com ajuste do número de árvores de decisão (450). Nos experimentos, nossa abordagem apresentou o melhor desempenho, atingindo média da ganho normalizado de 19,93%, 41,91% e 52,60% para as reco-

mendações Top-1, Top-3 e Top-5, respectivamente. Essas médias são melhores que os resultados apresentados quando as outras abordagens são utilizadas. Além disso, nossa abordagem apresentou a melhor acurácia em 29 dos 32 projetos considerando a recomendação Top-1. Em relação aos atributos mais relevantes, observou-se que alguns dos principais atributos foram introduzidos pelo Conjunto C: RequesterLogin ($C.2$), RequesterLocation ($C.3$) e ActiveDeveloper ($C.6$).

2. **Uma abordagem para prever o tempo de vida de *pull requests***, composta pelos atributos mais relevantes do Conjunto E, de acordo com a seleção de atributos *Information Gain Attribute Ranking* e pelo algoritmo de classificação *Random Forest* com ajuste do número de árvores de decisão (300). É importante ressaltar que a avaliação dos algoritmos de regressão utilizando o Conjunto D apresentou valores de erro médio e erro médio normalizado muito altos. Isso motivou a avaliação dos algoritmos de classificação considerando o atributo alvo com valores discretizados. Na comparação com a proposta anterior [8], nossa abordagem apresentou os melhores ganhos normalizados, obtendo a melhor acurácia em 18 dos 20 projetos avaliados. A acurácia média foi de 45,28% na tarefa de prever o tempo de vida de *pull requests*, o que proporcionou um ganho normalizado médio de 14,68% em relação à classe majoritária. Além disso, observou-se que alguns atributos propostos no Conjunto E se mostraram relevantes para determinar o tempo de vida de *pull requests* como, por exemplo: NumAddedFiles ($E.5$), NumTestChurn ($E.12$), WatcherProject ($E.16$) e os que caracterizam interações anteriores dos desenvolvedores no projeto ($E.17$ até $E.22$).

3. **Definição de um método de avaliação adequado para previsões envolvendo dados de *pull requests***, chamado *training-test sliding validation*. Esse método divide as bases de treinamento e teste preservando a ordem cronológica de submissão dos *pull requests*. Com isso, cada rodada de avaliação dos algoritmos preditivos utiliza apenas *pull requests* do passado para treinamento e *pull requests* do futuro para testes. Esse método de avaliação foi implementado por [81] em uma extensão para ferramenta Weka e disponibilizado para *download* no GitHub[1].

[1] `https://github.com/mlimeira/training-test-sliding-validation`

Vale mencionar que o artigo intitulado *Automatic Assignment of Integrators for Pull Requests Review: The Importance of Selecting Appropriate Attributes*, sobre o cenário de previsão de integradores, foi submetido em coautoria com os orientadores para o periódico *Journal of Systems and Software* e está em fase de revisão. Esse resultado foi apresentado no Capítulo 4.

6.2 Trabalhos Futuros

Uma sugestão de trabalho futuro é um estudo com o objetivo de prever se o *pull request* vai ser aceito ou rejeitado. Assim como nos cenários de prever integradores e o tempo de vida de *pull requests*, prever o estado final de *pull requests* pode auxiliar o processo de tomada de decisão dos integradores. Esse estudo seria análogo aos que foram propostos nesta tese, utilizando o método *training-test sliding validation* para avaliar algoritmos de classificação em diferentes conjuntos de atributos e estratégias de seleção de atributos para aprimorar o desempenho dos algoritmos.

Estudos semelhantes também podem ser voltados para tentar prever outras informações sobre *pull requests* como, por exemplo: o tipo do código recebido (nova funcionalidade, correção de *bug* ou refatoração) e se o *pull request* deve ser encarado com prioridade ou não. No entanto, os valores das informações históricas (já existentes nos repositórios) teriam que ser configurados e classificados pelas equipes dos projetos e por heurísticas predefinidas.

Para melhorar o desempenho das abordagens de previsão no contexto de *pull requests*, poderiam ser adicionados novos atributos preditivos coletados a partir de outras fontes de dados como, por exemplo: informações sobre métricas de código e características sobre o processo de integração contínua.

Outra proposta é a implementação de ferramentas de apoio às equipes de integradores durante o processo de integração de *pull requests*. Essas ferramentas poderiam ser integradas em plataformas como o GitHub sugerindo os integradores mais adequados para integrar um *pull request* e estimando o tempo de vida do *pull request*.

Referências

[1] CHACON, S.; STRAUB, B. *Pro Git*. 2. ed. Berkely, CA, USA: Apress, 2014. ISBN 1-4842-0077-2.

[2] HAN, J.; KAMBER, M.; PEI, J. *Data Mining: Concepts and Techniques*. 3. ed. Boston: Morgan Kaufmann, 2011. (The Morgan Kaufmann Series in Data Management Systems). ISBN 978-0-12-381479-1.

[3] WITTEN, I. H.; FRANK, E.; HALL, M. A.; PAL, C. J. *Data Mining: Practical Machine Learning Tools and Techniques*. 4. ed. Amsterdam: Elsevier Science, 2016. (The Morgan Kaufmann Series in Data Management Systems). ISBN 978-0-12-804357-8.

[4] GOUSIOS, G.; PINZGER, M.; DEURSEN, A. V. An Exploratory Study of the Pull-based Software Development Model. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India: ACM, 2014. p. 345–355. ISBN 978-1-4503-2756-5.

[5] BARR, E. T.; BIRD, C.; RIGBY, P. C.; HINDLE, A.; GERMAN, D. M.; DEVANBU, P. Cohesive and Isolated Development with Branches. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Tallinn, Estonia: Springer-Verlag, 2012. p. 316–331. ISBN 978-3-642-28871-5.

[6] GOUSIOS, G.; ZAIDMAN, A.; STOREY, M.-A.; DEURSEN, A. van. Work Practices and Challenges in Pull-based Development: The Integrator's Perspective. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy: IEEE, 2015. p. 358–368. ISBN 978-1-4799-1934-5.

[7] JIANG, J.; HE, J.-H.; CHEN, X.-Y. CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation. *Journal of Computer Science and Technology*, v. 30, n. 5, p. 998–1016, set. 2015. ISSN 1000-9000, 1860-4749.

[8] YU, Y.; WANG, H.; FILKOV, V.; DEVANBU, P.; VASILESCU, B. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In: *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*. Florence, Italy: IEEE, 2015. p. 367–371.

[9] LESSMANN, S.; BAESENS, B.; MUES, C.; PIETSCH, S. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, v. 34, n. 4, p. 485–496, jul. 2008. ISSN 0098-5589.

[10] GUYON, I.; ELISSEEFF, A. An Introduction to Variable and Feature Selection. *The Journal of Machine Learning Research*, v. 3, p. 1157–1182, mar. 2003. ISSN 1532-4435.

[11] DART, S. Concepts in Configuration Management Systems. In: *Proceedings of the 3rd International Workshop on Software Configuration Management (SCM)*. Trondheim, Norway: ACM, 1991. p. 1–18. ISBN 0-89791-429-5.

[12] ESTUBLIER, J. Software Configuration Canagement: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering (ICSE)*. Limerick, Ireland: ACM, 2000. p. 279–289. ISBN 1-58113-253-0.

[13] Institute of Electrical and Electronics Engineers. *Std 828 - IEEE Standard for Software Configuration Management Plans*. New York, NY: IEEE, 2005. ISBN 0-7381-4765-6.

[14] MURTA, L. G. P. *Gerência de Configuração no Desenvolvimento baseado em Componentes*. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2006.

[15] CAVALCANTI, Y. C.; NETO, P. A. M. S.; MACHADO, I. C.; VALE, T. F.; ALMEIDA, E. S.; MEIRA, S. R. L. Challenges and Opportunities for Software Change Request Repositories: A Systematic Mapping Study. *Journal of Software: Evolution and Process*, v. 26, n. 7, p. 620–653, jul. 2014. ISSN 2047-7481.

[16] CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. *ACM Computing Surveys*, v. 30, n. 2, p. 232–282, jun. 1998. ISSN 0360-0300.

[17] MENS, T. A State-of-The-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, v. 28, n. 5, p. 449–462, maio 2002. ISSN 0098-5589.

[18] ESTUBLIER, J. Objects Control for Software Configuration Management. In: *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE)*. London, UK: Springer-Verlag, 2001. p. 359–373. ISBN 3-540-42215-3.

[19] PRUDÊNCIO, J. G.; MURTA, L.; WERNER, C.; CEPÊDA, R. To Lock, or Not to Lock: That is The Question. *Journal of Systems and Software*, v. 85, n. 2, p. 277–289, fev. 2012. ISSN 0164-1212.

[20] BIRD, C.; ZIMMERMANN, T. Assessing the Value of Branches with What-if Analysis. In: *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*. Cary, NC, USA: ACM, 2012. p. 45:1–45:11. ISBN 978-1-4503-1614-9.

[21] BITBUCKET. *Pull Requests*. 2014. Disponível em: <https://www.atlassian.com/git/tutorials/making-a-pull-request/how-it-works>. Acesso em: 07 de dez. 2014.

[22] GITHUB. *Using pull requests - User Documentation*. 2014. Disponível em: <https://help.github.com/articles/using-pull-requests/>. Acesso em: 07 de dez. 2014.

[23] GOUSIOS, G.; SPINELLIS, D. GHTorrent: Github's Data From a Firehose. In: *Proceedings on the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. Zürich, Switzerland: IEEE, 2012. p. 12–21.

[24] GOUSIOS, G. The GHTorent Dataset and Tool Suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. San Francisco, USA: IEEE, 2013. p. 233–236. ISBN 978-1-4673-2936-1.

[25] GOUSIOS, G. *The GHTorrent Project*. Disponível em: <http://ghtorrent.org/>. Acesso em: 14 de mar. 2017.

[26] GRIGORIK, I. *The GitHub Archive*. Disponível em: <https://www.githubarchive.org/>. Acesso em: 29 de jan. 2015.

[27] TAN, P. N.; STEINBACH, M.; KUMAR, V.; KARPATNE, A. *Introduction to Data Mining*. 2. ed. Boston, MA, USA: Addison-Wesley, 2013. 792 p. ISBN 978-0-13-312890-1.

[28] FAYYAD, U. M.; PIATETSKY-SHAPIRO, G.; SMYTH, P.; UTHURUSAMY, R. (Ed.). *Advances in Knowledge Discovery and Data Mining*. 1. ed. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1996. ISBN 0-262-56097-6.

[29] FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, v. 17, n. 3, p. 37, mar. 1996. ISSN 0738-4602.

[30] ZAKI, M. J.; JR, W. M. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. 1. ed. New York, NY, USA: Cambridge University Press, 2014. ISBN 0-521-76633-8.

[31] BRIER, G. W.; ALLEN, R. A. Verification of Weather Forecasts. In: MALONE, T. F. (Ed.). *Compendium of Meteorology*. [S.l.]: American Meteorological Society, 1951. p. 841–848. ISBN 978-1-940033-70-9. DOI: 10.1007/978-1-940033-70-9_68.

[32] SIEGEL, S.; CASTELLAN, N. *Estatística Não-Paramétrica para Ciências do Comportamento*. 2. ed. Porto Alegre: Artmed, 2006. (Métodos de pesquisa). ISBN 9788536307299.

[33] DUDA, R. O.; HART, P. E.; STORK, D. G. *Pattern Classification*. 2. ed. [S.l.]: John Wiley & Sons, 2012. ISBN 978-1-118-58600-6.

[34] QUINLAN, J. R. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1-55860-240-2.

[35] BREIMAN, L. Random Forests. *Machine Learning*, v. 45, n. 1, p. 5–32, out. 2001. ISSN 0885-6125.

[36] HEARST, M.; DUMAIS, S.; OSMAN, E.; PLATT, J.; SCHOLKOPF, B. Support vector machines. *IEEE Intelligent Systems and their Applications*, v. 13, n. 4, p. 18–28, jul. 1998. ISSN 1094-7167.

[37] COVER, T.; HART, P. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, v. 13, n. 1, p. 21–27, jan. 1967. ISSN 0018-9448.

[38] QUINLAN, J. R. Induction of Decision Trees. *Machine Learning*, v. 1, n. 1, p. 81–106, mar. 1986. ISSN 0885-6125, 1573-0565.

[39] BOSER, B. E.; GUYON, I. M.; VAPNIK, V. N. A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the 5th Annual Workshop on Computational Learning Theory*. Pittsburgh, PA, USA: ACM, 1992. p. 144–152. ISBN 978-0-89791-497-0.

[40] ÜSTÜN, B.; MELSSEN, W. J.; BUYDENS, L. M. C. Facilitating the Application of Support Vector Regression by Using a Universal Pearson VII Function Based kernel. *Chemometrics and Intelligent Laboratory Systems*, v. 81, n. 1, p. 29–40, mar. 2006. ISSN 0169-7439.

[41] WANG, Y.; WITTEN, I. H. Induction Model Trees for Continuous Classes. In: *Proceedings of the 9th European Conference on Machine Learning Poster Papers*. Prague, Czech Republic: Springer, 1997. p. 128–137.

[42] QUINLAN, R. J. Learning With Continuous Classes. In: *5th Australian Joint Conference on Artificial Intelligence*. Singapore: World Scientific, 1992. p. 343–348.

[43] VAPNIK, V. N. An Overview of Statistical Learning Theory. *IEEE Transactions on Neural Networks*, v. 10, n. 5, p. 988–999, set. 1999. ISSN 1045-9227.

[44] EIN-DOR, P.; FELDMESSER, J. *UCI Machine Learning Repository Computer Hardware Data Set*. 2013. Disponível em: <https://archive.ics.uci.edu/ml/datasets/Computer+Hardware>. Acesso em: 16 de mar. 2017.

[45] SMOLA, A. J.; SCHöLKOPF, B. A Tutorial on Support Vector Regression. *Statistics and Computing*, v. 14, n. 3, p. 199–222, ago. 2004. ISSN 0960-3174.

[46] BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. TravisTorrent: Synthesizing Travis CI and GitHub for Full-stack Research on Continuous Integration. In: *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, 2017. p. 447–450. ISBN 978-1-5386-1544-7.

[47] TSAY, J.; DABBISH, L.; HERBSLEB, J. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India: ACM, 2014. p. 356–366. ISBN 978-1-4503-2756-5.

[48] VEEN, E. V. D.; GOUSIOS, G.; ZAIDMAN, A. Automatically Prioritizing Pull Requests. In: *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*. Florence, Italy: IEEE, 2015. p. 357–361. ISBN 978-0-7695-5594-2.

[49] YU, Y.; WANG, H.; YIN, G.; LING, C. X. Reviewer Recommender of Pull-requests in GitHub. In: *Proceedings on the 30th International Conference on Software Maintenance and Evolution (ICSME)*. Victoria, BC, Canada: IEEE, 2014. p. 609–612.

[50] YU, Y.; WANG, H.; YIN, G.; WANG, T. Reviewer Recommendation for Pull-requests in GitHub: What Can We Learn From Code Review and Bug Assignment? *Information and Software Technology*, v. 74, p. 204–218, jan. 2016. ISSN 0950-5849.

[51] YING, H.; CHEN, L.; LIANG, T.; WU, J. EARec: Leveraging Expertise and Authority for Pull-request Reviewer Recommendation in GitHub. In: *Proceedings of the 3rd International Workshop on Crowd Sourcing in Software Engineering*. Austin, Texas: ACM, 2016. p. 29–35. ISBN 978-1-4503-4158-5.

[52] RAHMAN, M. M.; ROY, C. K.; COLLINS, J. A. CoRReCT: Code Reviewer Recommendation in GitHub Based on Cross-project and Technology Experience. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas: ACM, 2016. p. 222–231. ISBN 978-1-4503-4205-6.

[53] BACCHELLI, A.; BIRD, C. Expectations, Outcomes, and Challenges of Modern Code Review. In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. San Francisco, USA: IEEE, 2013. p. 712–721. ISBN 978-1-4673-3076-3.

[54] CAVALCANTI, Y. C.; MACHADO, I. C.; NETO, P. A. M. S.; ALMEIDA, E. S. Towards Semi-automated Assignment of Software Change Requests. *Journal of Systems and Software*, v. 115, p. 82–101, maio 2016. ISSN 0164-1212.

[55] ANVIK, J. Automating Bug Report Assignment. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. Shanghai, China: ACM, 2006. p. 937–940. ISBN 1-59593-375-1.

[56] ČUBRANIĆ, D. Automatic Bug Triage Using Text Categorization. In: *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Banff, Alberta, Canadá: KSI Press, 2004. p. 92–97.

[57] CANFORA, G.; CERULO, L. Supporting Change Request Assignment in Open Source Development. In: *Proceedings of the 21st Symposium on Applied Computing (SAC)*. Dijon, France: ACM, 2006. p. 1767–1772. ISBN 1-59593-108-2.

[58] ANVIK, J.; MURPHY, G. C. Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions. *ACM Transactions on Software Engineering and Methodology*, v. 20, n. 3, p. 10:1–10:35, ago. 2011. ISSN 1049-331X.

[59] XUAN, J.; JIANG, H.; REN, Z.; ZOU, W. Developer Prioritization in Bug Repositories. In: *Proceedings on the 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, 2012. p. 25–35.

[60] JIANG, J.; YANG, Y.; HE, J.; BLANC, X.; ZHANG, L. Who Should Comment on This Pull Request? Analyzing Attributes for More Accurate Commenter Recommendation in Pull-based Development. *Information and Software Technology*, v. 84, p. 48–62, abr. 2017. ISSN 0950-5849.

[61] THONGTANUNAM, P.; TANTITHAMTHAVORN, C.; KULA, R. G.; YOSHIDA, N.; IIDA, H.; MATSUMOTO, K.-i. Who Should Review My Code? A File Location-based Code-reviewer Recommendation Approach for Modern Code Review. In: *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Montreal, QC, Canada: IEEE, 2015. p. 141–150.

[62] PANJER, L. Predicting Eclipse Bug Lifetimes. In: *Proceedings on the 4th International Workshop on Mining Software Repositories (MSR)*. Minneapolis, Minnesota, USA: IEEE, 2007. p. 29–32.

[63] MARKS, L.; ZOU, Y.; HASSAN, A. E. Studying the Fix-time for Bugs in Large Open Source Projects. In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE)*. Banff, AB, Canada: ACM, 2011. p. 11:1–11:8. ISBN 978-1-4503-0709-3.

[64] BHATTACHARYA, P.; NEAMTIU, I. Bug-fix Time Prediction Models: Can We Do Better? In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. Waikiki, Honolulu, HI, USA: ACM, 2011. p. 207–210. ISBN 978-1-4503-0574-7.

[65] JÚNIOR, M. L. de L.; SOARES, D. M.; PLASTINO, A.; MURTA, L. Developers Assignment for Analyzing Pull Requests. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*. Salamanca, Spain: ACM, 2015. p. 1567–1572. ISBN 978-1-4503-3196-8.

[66] JOHN, G. H.; LANGLEY, P. Estimating Continuous Distributions in Bayesian Classifiers. In: *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. p. 338–345. ISBN 1-55860-385-9.

[67] AHA, D. W.; KIBLER, D.; ALBERT, M. K. Instance-Based Learning Algorithms. *Machine Learning*, v. 6, n. 1, p. 37–66, jan. 1991. ISSN 0885-6125.

[68] PLATT, J. C. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Technical Report MSR-TR-98-14, 1998. 21 p.

[69] HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WITTEN, I. H. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, v. 11, n. 1, p. 10–18, nov. 2009. ISSN 1931-0145.

[70] FAYYAD, U. M.; IRANI, K. B. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*. Chambéry, France: Morgan Kaufmann, 1993. p. 1022–1029.

[71] HALL, M. A. *Correlation-based Feature Selection for Machine Learning*. Tese (Doutorado) — University of Waikato, Hamilton, New Zealand, 1999.

[72] HALL, M. A.; HOLMES, G. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. *IEEE Transactions on Knowledge and Data Engineering*, v. 15, n. 6, p. 1437–1447, nov. 2003. ISSN 1041-4347.

[73] PAPPA, G. L.; FREITAS, A. A. Automatically Evolving Rule Induction Algorithms. In: *Proceedings of the 17th European Conference on Machine Learning (ECML)*. Berlin, Heidelberg: Springer-Verlag, 2006. p. 341–352. ISBN 978-3-540-45375-8.

[74] DEMŠAR, J. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research*, v. 7, p. 1–30, dez. 2006. ISSN 1532-4435.

[75] FRIEDMAN, M. A Comparison of Alternative Tests of Significance for the Problem of m Rankings. *The Annals of Mathematical Statistics*, v. 11, n. 1, p. 86–92, mar. 1940. ISSN 0003-4851.

[76] NEMENYI, P. *Distribution-free Multiple Comparisons*. Tese (Doutorado) — Princeton University, 1963.

[77] POHLERT, T. *The Pairwise Multiple Comparison of Mean Ranks Package (PMCMR)*. [S.l.: s.n.], 2014. R package.

[78] FOWLER, M. *Continuous Integration*. 2006. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 07 de abr. 2017.

[79] SHEVADE, S. K.; KEERTHI, S. S.; BHATTACHARYYA, C.; MURTHY, K. R. K. Improvements to The SMO Algorithm for SVM Regression. *IEEE Transactions on Neural Networks*, v. 11, n. 5, p. 1188–1193, set. 2000. ISSN 1045-9227.

[80] KONONENKO, I. Estimating Attributes: Analysis and Extensions of RELIEF. In: *Proceedings of the 7th European Conference on Machine Learning (ECML)*. Catania, Italy: Springer-Verlag, 1994. p. 171–182. ISBN 978-3-540-57868-0 978-3-540-48365-6.

[81] LIMA, M. W. S. *Uma Extensão da Ferramenta WEKA para Avaliação de Tarefas Preditivas*. Monografia (Trabalho de Conclusão de Curso) — Universidade Federal do Acre, Rio Branco, 2017.