UNIVERSIDADE FEDERAL DO ACRE

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

PR-DISCOVER: UMA EXTENSÃO DA FERRAMENTA WEKA PARA EXTRAIR E VISUALIZAR REGRAS DE ASSOCIAÇÃO SOBRE PULL REQUESTS

RIO BRANCO

2018

JOSÉ MARCOS LOPES DAMASCENO

PR-DISCOVER: UMA EXTENSÃO DA FERRAMENTA WEKA PARA EXTRAIR E VISUALIZAR REGRAS DE ASSOCIAÇÃO SOBRE PULL REQUESTS

Monografia apresentada como exigência final para obtenção do grau de bacharel em Sistemas de Informação da Universidade Federal do Acre.

Orientador: Prof.Dr. Daricélio Moreira Soares

RIO BRANCO

2018

TERMO DE APROVAÇÃO

JOSÉ MARCOS LOPES DAMASCENO

PR-DISCOVER: UMA EXTENSÃO DA FERRAMENTA WEKA PARA EXTRAIR E VISUALIZAR REGRAS DE ASSOCIAÇÃO SOBRE PULL REQUESTS

Esta monografia foi apresentada como trabalho de conclusão de Curso de Bacharelado em Sistemas de Informação da Universidade Federal do Acre, sendo aprovado pela banca constituída pelo professor orientador e membros abaixo mencionados.

Compuseram a banca:

Prof. Daricélio Moreira Soares, Dr.
Curso de Bacharelado em Sistemas de Informação

Prof. Laura Costa Sarkis, Dra.
Curso de Bacharelado em Sistemas de Informação

Prof. Catarina de Souza Costa, Dra.
Curso de Bacharelado em Sistemas de Informação

Daniel Augusto N. da Silva, Bel.
Ministério Público Federal

Rio Branco, 21 de março 2018

Dedico este trabalho aos meus pais,
que me apoiaram e me direcionaram
ao caminho do conhecimento.

AGRADECIMENTOS

Agradeço aos meu pais, Sivaldo e Maria, que me deram condições e oportunidades para eu estar onde estou.

Agradeço ao meu orientador, Prof. Dr. Daricélio Moreira Soares, por ter me guiado durante a realização deste trabalho e por suas palavras que me motivaram a continuar. Agradeço também aos professores Olacir Rodrigues Castro Junior e Laura Costa Sarkis, pelo o apoio oferecido durante as disciplinas de TCC I e II. Assim como agradeço à toda equipe docente, por todo o conhecimento transmitido durante os 4 anos de curso.

Aos meu amigos de curso de Sistemas de Informação, pela ousadia e alegria para superarmos juntos os desafios encontrados durante o curso.

A todos envolvidos durante esta etapa da minha vida, meu muito obrigado.

"Ideias, e somente ideias,
podem iluminar a escuridão."
(Ludwig von Mises)

RESUMO

O processo de descoberta de conhecimento em bases de dados sobre Pull Requests a partir de repositórios no GitHub requer a execução de passos de pré-processamento e mineração de regras de associação, conforme mostrado por Soares (2017). Não existem ferramentas especificas para a automação desses passos. Este trabalho apresenta a proposta de uma solução complementar à ferramenta Weka, capaz de automatizar os passos citados e facilitar a visualização da regras. A solução é avaliada e os resultados encontrados demonstram que a extensão é capaz de auxiliar os gerentes de projetos e contribuidores de software na descoberta de padrões sobre o paradigma de colaboração baseado em Pull Requests.

Palavras-chave: Engenharia de software. Mineração de dados. Regras de associação. *Pull requests*. Weka.

ABSTRACT

The process of discovering knowledge in Pull Requests databases from repositories in GitHub requires the execution of pre-processing and mining of association rules, as shown by Soares (2017). There are no specific tools for automating these steps. This work presents the proposal of a complementary solution to the Weka tool, able to automate the mentioned steps and facilitate the visualization of the rules. The solution is evaluated and the results show that the extension is able to assist project managers and software contributors in the discovery of standards on the paradigm of collaboration based on Pull Requests.

Key-words: Software Engineering. Data Mining. Association rules. Pull requests. Weka.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No cenário atual de desenvolvimento de software colaborativo, os engenheiros de software podem fazer uso de várias ferramentas para auxiliar no processo de gerência de configuração de seus projetos. Entre essas ferramentas se encontram sistemas de controle de versão, como Git[1] e Mercurial[2], que utilizam repositórios de software onde são armazenados os itens de configuração (IC's) e que tem o papel de manter o histórico do projeto e controlar a entrada e saída de IC's (IEEE, 2012). O melhor exemplo da utilização de repositórios de software no cenário de desenvolvimento colaborativo é o serviço GitHub[3], que, segundo auto-definição, é a plataforma líder mundial para desenvolvimento de software desta natureza.

O GitHub é um serviço de repositórios de softwares online que utiliza o sistema de controle de versão Git. Bitbucket[4], GitLab[5] e Coding[6] são alguns exemplos de serviços semelhantes. O GitHub oferece todas as funcionalidades do Git e adiciona funções próprias para proporcionar um controle maior sobre os projetos. Uma dessas funções é o *pull request*, que permite que mudanças nos IC's do projeto sejam revisadas e discutidas antes de serem integradas. Os *pull requests* podem ser utilizados tanto pelos colaboradores internos de um projeto, para a

[1] https://git-scm.com/
[2] https://www.mercurial-scm.org/
[3] https://github.com/
[4] https://bitbucket.org/
[5] https://about.gitlab.com/
[6] https://coding.net/

validação de mudanças pelo time de desenvolvimento e/revisão por pares, como por colaboradores externos, que podem contribuir espontaneamente em projetos *open-source* (GOUSIOS; PINZGER; DEURSEN, 2014).

O trabalho realizado por Soares (2017) buscou encontrar quais são os fatores que influenciam na aceitação dos *pull requests*, assim como seu tempo de vida e na atribuição de seus revisores, refletindo um panorama sobre a natureza do paradigma de colaboração baseado em *pull requests*. Soares (2017) extraiu regras de associação para identificar se, fatores físicos, sociais e relacionados ao processo de revisão, influenciam nos três cenários descritos acima. O processo de mineração de dados que foi utilizado não é trivial, envolvendo a utilização de múltiplas ferramentas, sendo necessário bom domínio do processo de descoberta de conhecimento em bases de dados.

Este trabalho busca desenvolver uma ferramenta para automatizar partes do processo proposto por Soares (2017) para descobrir regras de associação sobre fatores que influenciam na aceitação, tempo de vida e atribuição de revisores de *pull requests* em repositórios no GitHub, utilizando dados extraídos com a ferramenta GHTorrent[7]. Como principal contribuição desta pesquisa, espera-se uma ferramenta capaz de auxiliar colaboradores de software e gerentes de projetos na obtenção de informações sobre os fatores de aceitação, tempo de vida e atribuição de revisores dos *pull requests* sobre seus repositórios no GitHub.

1.1 PROBLEMA DA PESQUISA

Soares (2017) conduziu uma pesquisa para explorar a natureza do *pull requests* quanto a fatores que influenciam em sua aceitação, tempo de vida e atribuição de revisores, utilizando dados sobre 132.660 *pull requests* de 88 projetos de software. Tal pesquisa descobriu que as características físicas dos *pull requests*, o perfil dos colaboradores, aspectos sociais do processo e a localização das contribuições são fatores que influenciam, em diferentes intensidades de força, na

[7] http://www.ghtorrent.org/

aceitação/rejeição, no tempo de vida e na atribuição de revisores. A identificação desses padrões pode apoiar colaboradores e gerentes de projeto na compreensão da natureza dos *pull requests* e guiá-los a práticas que permitam extrair ao máximo os benefícios desse paradigma de colaboração.

Para que tais fatores fossem identificados foi necessário seguir uma serie de passos (SOARES, 2017):

1. Extrair os dados dos repositórios utilizando a ferramenta GHTorrent;
2. Pré-processamento dos dados, que inclui:
    a) Limpeza dos dados e seleção dos atributos, para remover dados ausentes, incorretos ou inconsistentes;
    b) Discretização dos dados numéricos, utilizando um filtro não supervisionado da ferramenta Weka, onde foi considerando a distribuição de frequência dos dados;
    c) Os rótulos foram ajustados para uma melhor interpretação semântica dos padrões extraídos;
3. Extração de regras de associação usando o algoritmo Apriori implementado na ferramenta Weka, definindo os valores de suporte e confiança mínimos para garantir que as regras não fossem obtidas por acaso;
4. Interpretação e avaliação dos resultados, com o auxilio da ferramenta WekaPAR.

Os passos 2, 3 e, parcialmente, 4, são passos que seguem padrões bem definidos e são realizados com a junção de esforço humano e auxilio computacional. Porém, mesmo com o auxilio computacional, esses passos podem ser considerados altamente dispendiosos para os colaboradores de software e gerentes de projetos que precisam seguir prazos bem definidos. Além disso, para que o processo atinja bons resultados é necessário conhecimento específico de mineração de dados. Este

fato faz com que seja necessária a automação do processo em seus passos que ocorrem após a extração dos dados sobre o repositório.

Nesse contexto, questiona-se: Como automatizar parte do processo de descoberta dos fatores que influenciam na aceitação, tempo de vida e atribuição de revisores? Como fornecer informações válidas sobre *pull requests* para gerentes de projeto e colaboradores de software?

1.2 OBJETIVOS DA PESQUISA

Esta seção apresenta os objetivos que foram definidos para guiar o andamento do trabalho, tanto o objetivo geral, como os específicos.

1.2.1 Objetivo geral

Desenvolver uma ferramenta que automatize as etapas de pré-processamento e mineração de regras de associação do processo proposto por Soares (2017) para descobrir fatores que influenciam na aceitação, tempo de vida e atribuição de revisores dos *pull requests,* a partir de dados coletados com a ferramenta GHTorrent sobre repositórios no GitHub.

1.2.2 Objetivos específicos

Os seguintes objetivos específicos precisam ser alcançados previamente, para permitir que o objetivo geral seja alcançado:

a) Revisão da bibliografia;

b) Elaboração do documento de requisitos da ferramenta;

c) Analise de componentes para reúso;

d) Desenvolvimento e integração dos componentes;

e) Criação do pacote Weka da ferramenta;

f) Validar a ferramenta desenvolvida.

1.3 JUSTIFICATIVA DA PESQUISA

Em um cenário em que gerentes de projetos e colaboradores de software podem fazer o uso de diversas ferramentas para auxiliar na realização de suas atividades, fica claro que a automação de processos ajuda a melhorar o fluxo de trabalho, tornando-o mais eficiente e eficaz.

Verificando a situação atual do campo de descoberta de conhecimento sobre os fatores que influenciam os *pull requests*, é possível identificar que a área pode se aproveitar do desenvolvimento de ferramentas que automatizem passos necessários para o sucesso de seus objetivos.

A ferramenta desenvolvida neste trabalho busca auxiliar gerentes de software no processo de descoberta de conhecimento sobre padrões de colaboração existentes em seus repositórios, o que pode oportunizar a criação de boas práticas de colaboração, além de permitir a identificação de áreas criticas do projeto, criando oportunidades para implantação de outras ações de gerenciamento.

Colaboradores de software também podem descobrir conhecimento sobre os repositórios, auxiliados pela automação presente na ferramenta, para que possam definir estratégias e se adequarem aos fatores presentes nos seus repositórios de interesse, aumentando a aceitação de seus *pull requests*.

1.4 METODOLOGIA

Os procedimentos metodológicos que serão utilizados neste trabalho estão de acordo com as seguintes classificações definidas por Silva e Menezes (2005):

a) Do ponto de vista da sua natureza, o trabalho é uma pesquisa aplicada porque é uma aplicação prática dirigida à solução de um problema;

b) Do ponto de vista da abordagem do problema, o trabalho é uma pesquisa qualitativa porque é realizado uma interpretação dos fenômenos e atribuição de significados, sendo essa uma atividade básica de uma pesquisa qualitativa;

c) Do ponto de vista dos seus objetivos, o trabalho é uma pesquisa exploratória porque busca proporcionar melhor entendimento do problema, tornando-o mais explícito e sugerindo uma hipótese para solucionar o problema.

O trabalho foi dividido em quatro etapas distintas:

1. Na primeira etapa foi realizada a fundamentação teórica do trabalho, mostrando os conceitos sobre engenharia de software, abordando os temas de processo de software, desenvolvimento orientado a reúso, engenharia de requisitos e gerência de configuração de software (sistemas de controle de versões e *pull requests*), assim como sobre a mineração de dados, se aprofundando nos tópicos de processo de KDD, tarefas de mineração, regras de associação e, finalmente, a ferramenta Weka, que são necessários para o entendimento e a realização deste trabalho;

2. Na segunda etapa foi realizado o processo de engenharia de requisitos, com a criação de um documento de requisitos, para elucidar as funcionalidades que a ferramenta a ser desenvolvida deve possuir para se alcançar o objetivo deste trabalho;

3. O desenvolvimento da ferramenta utilizando linguagem de programação e a integração da solução à ferramenta Weka, utilizando as funcionalidades do gerenciador de pacotes presente nesta, é a terceira etapa do trabalho;

4. Na quarta e última etapa do trabalho, para validar a ferramenta desenvolvida, foram utilizados dados sobre *pull requests* de repositórios no GitHub. Os resultados do processamento foram analisados e são apresentados neste trabalho.

Durante o desenvolvimento foi utilizado o código fonte da ferramenta Weka, que é disponibilizado junto com a ferramenta, assim como sua API. A linguagem de programação utilizada foi o Java[8], com o auxilio do ambiente de desenvolvimento integrado (*Integrated Development Environment* - IDE) Eclipse[9].

Os dados utilizados na última fase foram coletados utilizando a ferramenta GHTorrent.

1.5 ORGANIZAÇÃO DO ESTUDO

Além desta introdução, responsável por contextualizar o problema e motivações da pesquisa, o trabalho contém mais 5 capítulos.

No Capítulo 2 é mostrada a fundamentação teórica para o conceito de engenharia de software. Além de definir engenharia de software, o capítulo também aborda os processos de software, o desenvolvimento orientado a reúso, a engenharia de requisitos e a gerência de configuração de software, neste último caso é aprofundado os conceitos de sistemas de controle de versão e *pull requests*.

[8] https://www.oracle.com/br/java/index.html
[9] https://eclipse.org/

O Capítulo 3 define os conceitos de mineração de dados, onde são explorados os conceitos sobre processo de KDD, tarefas de mineração, regras de associação e a ferramenta Weka.

No Capítulo 4 a solução proposta por este trabalho é apresentada, dando detalhes sobre o processo de seu desenvolvimento e integração com a ferramenta Weka.

Já o Capítulo 5 realiza uma avaliação sobre a solução proposta, demonstrando os resultados encontrados com a utilização da ferramenta que foi desenvolvida.

E finalmente, o Capítulo 6 é composto pelas considerações finais sobre o trabalho, onde é feito uma conclusão sobre o seu sucesso, atentando-se ao cumprimento dos objetivos que foram estabelecidos, e são dadas sugestões para futuros trabalhos.

2 ENGENHARIA DE SOFTWARE

Sommerville (2011) define a Engenharia de Software como uma disciplina de engenharia que se preocupa com todos os aspectos da produção de software, desde seus estágios iniciais de especificação do sistema, até a manutenção do sistema durante seu uso. Sendo assim, os engenheiros de software aplicam teorias, métodos e ferramentas onde for apropriado. Eles reconhecem que devem trabalhar com as restrições organizacionais e financeiras.

Já para Laplante (2007), Engenharia de Software é uma abordagem sistemática para análise, design, avaliação, implementação, teste, manutenção e reengenharia de software, ou seja, a aplicação de engenharia para software. Na engenharia de software, são definidos vários modelos para o ciclo de vida do software e muitas metodologias para a definição e avaliação das diferentes fases de um modelo de ciclo de vida. As atividades de engenharia de software também estão ligadas a produção dos artefatos relacionados à engenharia de software, como documentação e ferramentas. O Quadro 1 mostra uma visão geral das funções da Engenharia de Software.

Quadro 1. Funções da Engenharia de Software

Função	Descrição
Análise de requisitos	Determinar necessidades e restrições, analisando os requisitos do sistema alocados ao software
Design de software	Determinar formas de satisfazer requisitos e restrições, analisar possíveis soluções e selecionar a melhor

Função	Descrição
Planejamento de processo	Determinar tarefas de desenvolvimento de produtos, precedência e riscos potenciais para o projeto
Controle de processo	Determinar métodos para controlar o projeto e processar, medir o progresso e tomar medidas corretivas quando necessário
Verificação, validação e teste	Avaliar produto final e documentação

Fonte: Adaptado de Laplante (2007).

De acordo com Sommerville (2011), a Engenharia de Software não se preocupa apenas com os processos técnicos do desenvolvimento de software, mas também com atividades como gerenciamento de projetos de software e com o desenvolvimento de ferramentas, métodos e teorias para auxiliar a produção de software.

De maneira geral, a Engenharia de Software é responsável por garantir que o processo de desenvolvimento de software tenha uma alta qualidade e, desta forma, garantindo que o software desenvolvido também seja de alta qualidade.

2.1 PROCESSO DE SOFTWARE

Como definido por Sommerville (2011), um processo de software relaciona um conjunto de atividades que levam à produção de um produto de software. O desenvolvimento de software do zero utilizando uma linguagem de programação, pode ser uma dessas atividades. Porém, novos softwares podem ser desenvolvidos por meio de extensões ou modificações em sistemas já existentes.

Os processos de software devem incluir quatro atividades fundamentais para a engenharia de software (SOMMERVILLE, 2011):

1. **Especificação de software:** É onde ocorre a identificação e definição das funcionalidades e restrições do software quanto ao seu funcionamento.

2. **Projeto e implementação de software:** É a atividade onde o software é produzido, seguindo o que foi especificado anteriormente.

3. **Validação de software:** Nesta atividade o software passa por uma validação para verificar se ele atende às necessidades do cliente e ao que foi definido na especificação.

4. **Evolução de software:** Trata-se da evolução do software, para que ele atenda as necessidades de mudança dos clientes.

Para Pressman (2011), um processo, no contexto de engenharia de software, é uma abordagem adaptável que possibilita a realização do trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas para a entrega do software dentro do prazo e com qualidade. Assim uma metodologia de processo fornece um alicerce para a engenharia de software, pois identifica um pequeno número de atividades estruturais aplicáveis a todos os projetos de software, independente de seu tamanho ou complexidade. A metodologia de processo também é composta por um conjunto de atividades de apoio (*umbrella activities*), que podem ser utilizados por qualquer processo de software.

De acordo com Sommerville (2011), existem diferentes modelos de processo de software que podem ser definidos como uma representação simplificada de um processo de software e são uma perspectiva particular deste, fornecendo somente informações parciais. Ele considera que os modelos genéricos não são descrições definitivas, mas sim abstrações que podem ser usadas para explicar diferentes abordagens de desenvolvimento de software, assim como podem ser ampliados e adaptados em processos de engenharia mais específicos. Ele apresenta, inicialmente, três modelos de processo que podem ser utilizados:

1. **O modelo em cascata***.* Nesse modelo as atividades fundamentais do processo de desenvolvimento são representadas como fases distintas, especificação de requisitos, projeto de software, implementação e assim por diante. Neste modelo o resultado de cada estágio deve ser aprovado por um ou mais documentos e o estágio seguinte não deve ser iniciado até que a fase anterior seja concluída, por isso o nome de cascata.

2. **Desenvolvimento incremental**. No desenvolvimento incremental as atividades de especificação, desenvolvimento e validação são

intercaladas. É baseado na ideia de *feedback* rápido entre as atividades, onde uma implementação inicial é exposta ao cliente para coleta de comentários que vão direcionar as próximas versões, até que um sistema adequado seja desenvolvido. É uma parte fundamental das abordagens ágeis e se mostra como uma melhor opção ao modelo em cascatas para a maioria dos sistemas de negócios, *e-commerce* e sistemas pessoais.

3. **Engenharia de software orientada a reúso**. É um modelo que se baseia na existência de um número significativo de componentes reusáveis. O foco deste modelo está voltado para a integração desses componentes em um sistema já existente em vez de desenvolver sistema por completo.

Sommerville (2011) também explora outros modelos de processos, como a prototipação, a entrega incremental, o modelo espiral e o *Rational Unified Process* (RUP), que não serão explorados aqui.

A engenharia de software orientada a reúso é o modelo que contempla o escopo do desenvolvimento da ferramenta em questão, nesse sentido, tal metodologia é exposta na seção 2.2.

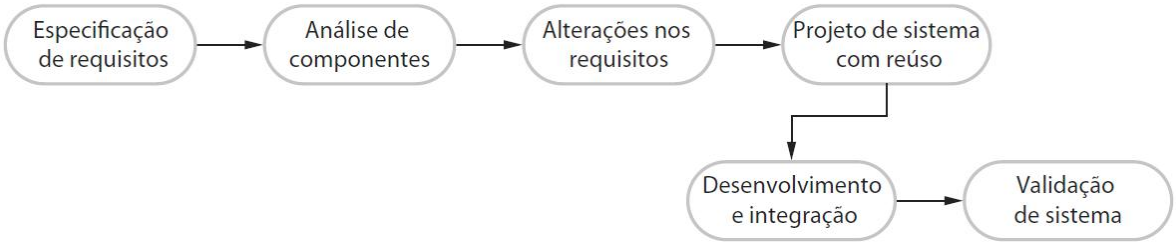
2.2 DESENVOLVIMENTO ORIENTADO A REÚSO

Como dito por Sommerville (2011), o reúso de software é utilizado na maioria dos projetos, muitas vezes de maneira informal, quando os desenvolvedores conhecem algum projeto ou código semelhante ao que é requisitado e isso é independente do processo de software utilizado. Porém, a partir do século XXI os processos de desenvolvimento de software orientado a reúso se tornaram amplamente usados.

Para Pressman (2011), o modelo de desenvolvimento orientado a reúso desenvolve aplicações a partir de componentes de software pré-empacotados. O processo de reúso é evolucionário em sua natureza, utilizando uma abordagem iterativa para a criação do software. Neste modelo a modelagem e construção

iniciam com a identificação de possíveis componentes a serem utilizados, sendo então considerados os itens de integração de componentes. A partir disso é projetado uma arquitetura de software para receber os componentes, que são integrados na arquitetura. Após isso é realizado testes completos que validam a funcionalidades do sistema.

Figura 1. Engenharia de software orientada a reúso



Fonte: Sommerville (2011, p. 23).

Sommerville também descreveu o funcionamento de um processo geral de desenvolvimento baseado em reúso, que é ilustrado na Figura 1. Mesmo que os estágios de especificação dos requisitos e de validação sejam equivalentes aos de outros processos, os estágios intermediários são específicos para o processo orientado a reúso, sendo eles:

1. **Análise de componentes.** Com os requisitos especificados, é realizado uma busca por componentes que possam ser utilizados para a implementação dos requisitos. Geralmente não existem componentes que resolvem todos os problemas, apenas alguma funcionalidade especifica.

2. **Modificação de requisitos.** Os requisitos são analisados para que sejam comparados com as informações sobre os componentes que foram descobertos. Então os requisitos são modificados de forma que estejam de acordo com os componentes disponíveis. Caso não seja possível a modificação de um requisito, é necessário realizar outra análise de componentes.

3. **Projeto do sistema com reúso.** Nesse estágio é projetado o framework do sistema ou um já existente é utilizado. Com os componentes conhecidos, os projetistas podem organizar o framework para reúso.

Pode ser necessário a utilização de novos softwares caso não haja componentes reutilizáveis disponíveis.

4. **Desenvolvimento e integração.** Durante esse estágio ocorre o desenvolvimento de softwares que não puderam ser adquiridos externamente, e os componentes escolhidos são integrados no novo sistema. Nesse modelo a integração de sistemas pode ocorrer durante o desenvolvimento, em vez de ser uma atividade separada.

A Figura 1 ilustra o processo de engenharia de software orientada a reúso.

De acordo com Sametinger (1997), existem muitos produtos que podem ser reutilizados, como: algoritmos, bibliotecas de funções, biblioteca de classes, arquitetura e design de software, classes de *framework*, padrões de design, aplicações e documentação.

Entre os benefícios gerados pela utilização do modelo de desenvolvimento orientado a reúso, pode se citar a redução no tempo do ciclo de desenvolvimento e a redução no custo do projeto caso a reutilização de componentes se torne parte de sua cultura (PRESSMAN, 2011). Inevitáveis compromissos com os requisitos, que podem levar a um sistema que não atende as reais necessidades do cliente, e uma possível perda de controle sobre a evolução do sistema estão entre as desvantagens da utilização deste modelo (SOMMERVILLE, 2011). Outras vantagens e desvantagens podem ser vistas no Quadro 2.

Quadro 2. Vantagens e desvantagens do reúso de software

Vantagens	Desvantagens
● Softwares reutilizados já foram testados sob vários aspectos e provavelmente são mais confiáveis; ● Risco de processo reduzido, na medida em que o custo do software existente já é conhecido, reduzindo a margem de erro de estimativa de custos; ● Evita repetir o mesmo trabalho, pois muitas funcionalidades já estão	● Possibilidade de maiores custos de manutenção, pois os componentes reutilizáveis podem se tonar incompatíveis com futuras modificações; ● Algumas ferramentas de software não suportam o desenvolvimento com reúso; ● Alguns profissionais acreditam que

Vantagens	Desvantagens
encapsuladas nos softwares reutilizáveis; ● Conformidade com padrões, sobretudo de interface gráfica, pois a utilização de componentes reutilizáveis permite ao usuário reconhecer o mesmo padrão em todas as aplicações; ● Redução do tempo de desenvolvimento e validação.	escrever seu próprio código é mais confiável; ● A criação, manutenção e uso de uma biblioteca de componente reutilizáveis pode demandar que processos de desenvolvimento sejam adaptados de forma a garantir que estes componentes sejam realmente utilizados; ● O emprego de reúso envolve encontrar, compreender e, muitas vezes, adaptar os componentes reutilizáveis.

Fonte: Adaptado de Silva (2016).

Levando em consideração as desvantagens do desenvolvimento orientado ao reúso apresentados no Quadro 2, a ferramenta a ser desenvolvida apresenta apenas a necessidade de adaptar os componentes, pois as outras desvantagens não estarão presentes durante o seu desenvolvimento. Desta forma, os benefícios que o modelo pode trazer se sobrepõem às suas desvantagens. Assim, o desenvolvimento orientado a reúso foi escolhido para o desenvolvimento da ferramenta proposta.

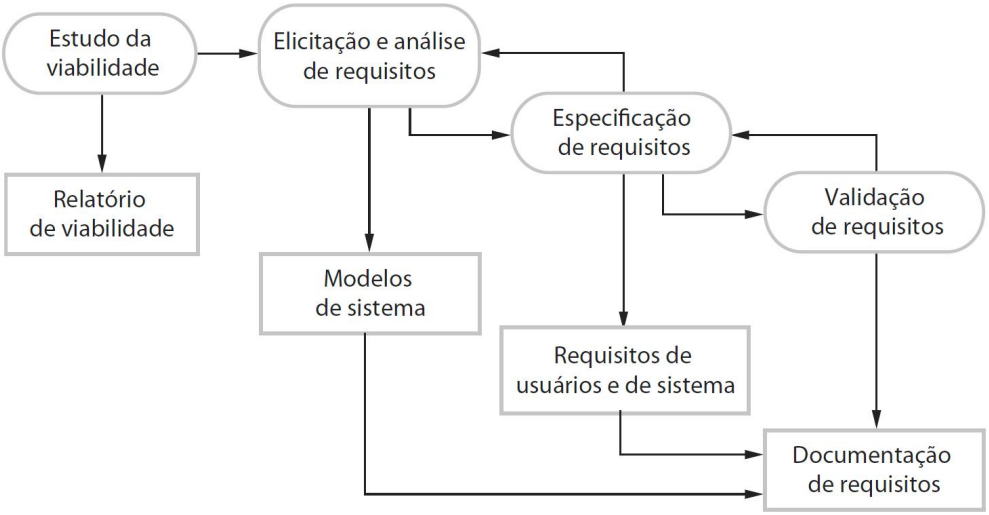
2.3 ENGENHARIA DE REQUISITOS

De acordo com Sommerville (2011), a Engenharia de Requisitos é o processo de compreensão e definição dos serviços requisitados do sistema e identificação de restrições relativas à operação e ao desenvolvimento do sistema. Assim sendo, a Engenharia de Requisitos possui um papel crítico no processo de desenvolvimento de softwares, pois erros nesta fase podem comprometer todo o andamento do projeto.

Já para Pressman (2011), a engenharia de requisitos é o amplo espectro de tarefas e técnicas que levam a um entendimento dos requisitos do sistema, estabelecendo uma base sólida para o projeto e para a construção. Sem ela, o software resultante tem grande probabilidade de não atender às necessidades do cliente. A engenharia de requisitos é um procedimento que se inicia durante a atividade de comunicação e continua durante a modelagem do sistema, se adaptando às necessidades do processo e do projeto.

Como Sommerville (2011) aponta, o processo de engenharia de requisitos tem como objetivo final a criação de um documento de requisitos, que especifica um sistema que satisfaz as necessidades dos *stakeholders*. A Figura 2 ilustra o processo de engenharia de requisitos.

Figura 2. Processo de engenharia de requisitos



Fonte: Sommerville (2011, p. 24).

Neste processo existem quatro atividades principais (SOMMERVILLE, 2011):

1. **Estudo de viabilidade.** Aqui é realizada uma estimativa sobre a possibilidade de as necessidades do cliente serem satisfeitas usando as tecnologias de hardware e software atuais. Também é verificado a rentabilidade do sistema de um ponto de vista de negócio e se ele pode ser desenvolvido com as restrições orçamentais impostas. Essa atividade deve ser rápida e barata, e seu resultado deve informar se o projeto irá avançar ou não.

2. **Elicitação e análise de requisitos.** Neste processo ocorre a derivação dos requisitos do sistema através da observação dos sistemas já existentes, assim como discussões com potenciais usuários e compradores, análise de tarefas e outras etapas. Podem ser desenvolvidos um ou mais modelos de sistemas e protótipos para auxiliar no entendimento do sistema.

3. **Especificação de requisitos.** As informações coletadas durante a análise são traduzidas em documento que possui um conjunto de requisitos. Os requisitos podem ser de dois tipos: requisitos de usuário, que são declarações abstratas dos requisitos do sistema para o cliente e usuário final do sistema, e requisitos de sistema, que são uma descrição com mais detalhes da funcionalidade que deve ser desenvolvida.

4. **A validação de requisitos.** É a atividade que verifica se os requisitos quanto ao seu realismo, consistência e completude. É nessa etapa que erros presentes no documento de requisitos são identificados. Após isso o documento é modificado com as correções necessárias.

Sommerville também aponta que as atividades não são realizadas em uma única sequência, sendo que a análise de requisitos continua durante a definição e especificação, e novos requisitos surgem durante o processo, fazendo com que estas atividades sejam intercaladas.

Como Pressman (2011) cita, conforme os requisitos são identificados e o modelo de análise é criado, ocorrem negociações entre a equipe de software e os interessados no projeto sobre a prioridade, disponibilidade e custo de cada requisito. Isso é feito para que o plano do projeto seja realista. Assim como cada requisito é validado para que esteja realmente de acordo com as necessidades do cliente.

A engenharia de requisitos é um passo necessário para o desenvolvimento de softwares de qualidade que realmente atendem as necessidades de seus clientes. Por tal motivo, suas técnicas foram amplamente utilizadas durante a fase de especificação da ferramenta proposta por este trabalho.

2.4 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

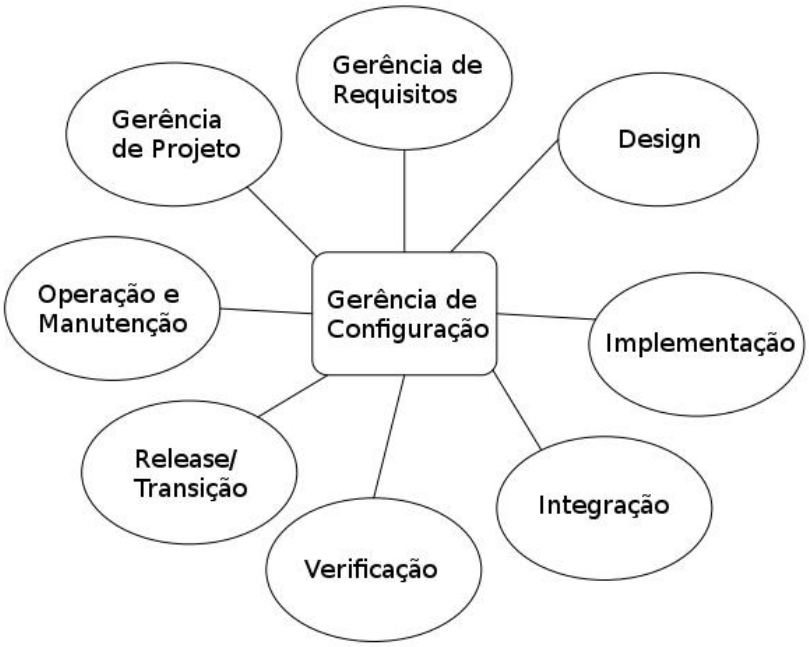
A Gerência de Configuração de Software (GCS), segundo Estublier (2000), é “a disciplina que nos permite manter a evolução dos produtos de software sob controle e, assim, contribui para a satisfação de restrições de qualidade e atraso”. Para Dart (1991), a GCS é a disciplina para controlar a evolução dos softwares. A GCS auxilia os gerentes de projetos a manterem o ambiente de desenvolvimento sob controle.

Ainda segundo Estublier, um típico sistema GCS tenta prover serviços nas seguintes áreas: gerenciar um repositório de componentes, ajudar engenheiros em suas atividades usuais e controle de processos e suporte.

Para Pressman (2011), a gerência de configuração de software é uma atividade do tipo “guarda-chuva” que identifica, organiza e controla modificações no software em desenvolvimento pela equipe de programação. Seu objetivo é maximizar a produtividade e minimizar os erros. Como as mudanças podem ocorrer a qualquer momento, suas atividades tem como objetivo identificar a alteração, controlar a alteração, garantir que alteração seja implementada corretamente e relatar as alterações para os outros envolvidos.

Dentro da norma IEEE 828-2012 a GCS é definida como uma disciplina que aplica direção técnica e administrativa, assim como vigilância, para: identificar e documentar as características funcionais e físicas de um Item de Configuração (IC), controlar mudanças nessas características, registrar e reportar o processo de mudança e o estado da implementação, e também verificar a conformidade com os requisitos especificados. Na mesma norma, um IC é definido como a agregação de produtos de trabalho que são designados para gerenciamento de configuração e tratado como uma entidade única no processo de gerenciamento de configuração. A Figura 3 mostra os processos suportados pela GCS.

Figura 3. Exemplos de processos suportados pela GCS



Fonte: Adaptado de IEEE 828-2012.

Sommerville (2011) diz que o gerenciamento de configuração envolve quatro atividades:

1. **Gerenciamento de mudanças.** Responsável por acompanhar as solicitações dos clientes e desenvolvedores por mudanças no software, definindo os custos e o impacto de se realizar tais mudanças, para que seja tomada a decisão de implementar ou não as mudanças.

2. **Gerenciamento de versões.** Acompanha as várias versões dos componentes do sistema, assegurando que as mudanças nos componentes, realizadas por diferentes desenvolvedores, não apresentem incompatibilidade.

3. **Construção do sistema.** Nesse processo é onde ocorre a junção dos componentes de programa, dados e bibliotecas, para que sejam compilados e ligados.

4. **Gerenciamento de *releases*.** Relacionado a preparação de software para o *release* externo e com o acompanhamento das versões que estão sendo utilizadas pelo cliente.

2.4.1 Sistemas de controle de versão

Para Sommerville (2011), os sistemas de controle de versões são uma ferramenta que oferece suporte ao gerenciamento de versões. Eles identificam, armazenam e controlam o acesso a diferentes versões de componentes, e devem sempre ser utilizados. Murta (2006) por sua vez, considera que os sistemas de controle de versões “permitem que os IC’s sejam identificados, segundo estabelecido pela função de identificação da configuração, e que eles evoluam de forma distribuída e concorrente, porém disciplinada”.

Sommerville (2011) descreve que os sistemas de controle de versões fornecem uma variedade de recursos:

1. **Identificação de versão e *release*.** Ao serem submetidas ao sistema, as versões gerenciadas recebem identificadores que geralmente se baseiam no nome to item de configuração seguido por números que identificam sua versão. Uma identificação consistente é importante, pois simplifica a definição de configuração.

2. **Gerenciamento de armazenamento.** Como um projeto de software apresenta muitos itens de configuração e estes itens acabam apresentando muitas versões que se diferem apenas ligeiramente, os sistemas de gerenciamento de versões fornecem recursos de armazenamento para reduzir o espaço necessário. Não são mantidas verões completas de cada versão, sendo armazenadas apenas uma lista de diferenças (deltas) entre cada versão. Unindo os deltas com uma versão-fonte (usualmente a versão mais recente), é possível recriar uma versão específica do item de configuração.

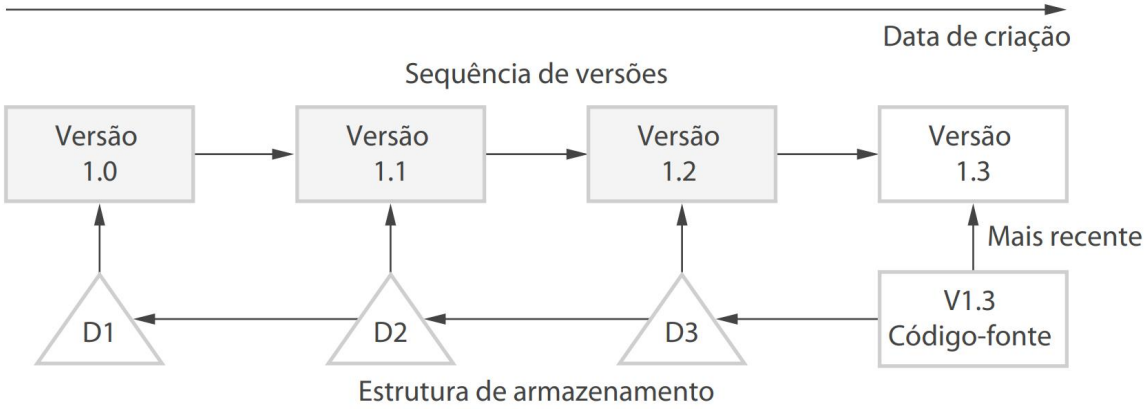
3. **Registro de histórico de mudanças.** São registradas e listadas todas as modificações feitas no código de um componente ou sistema. Esses registros podem ser utilizados para escolher uma versão específica do sistema. Para isso é necessário utilizar palavras-chaves que descrevam os mudanças feitas para os componentes.

4. **Desenvolvimento independente.** Permite que mais de um desenvolvedor trabalhe com o mesmo item de configuração ao mesmo tempo. O sistema de gerenciamento de versões é responsável por garantir que as modificações feitas por diferentes desenvolvedores não causem conflitos.

5. **Suporte a projetos.** É capaz de apoiar o processo de desenvolvimento de vários projetos, que compartilham componentes.

A Figura 4 ilustra o funcionamento da identificação de versão e do gerenciamento de armazenamento através do exemplo de um IC e sua evolução durante o desenvolvimento.

Figura 4. Versões e armazenamento



Fonte: Sommerville (2011, p. 483).

O IC da Figura 4, em sua primeira submissão ao sistema recebeu como versão o número 1.0. Em cada submissão posterior do mesmo IC, foi atribuído um novo valor de versão, de maneira crescente (1.1, 1.2, 1.3), para indicar que se trata de uma versão mais atual, no caso ilustrado a versão mais atual é a 1.3. A cada nova submissão do IC foi feito a identificação das diferenças entre a versão mais atual presente no repositório e a versão submetida. Esta diferença foram armazenadas como deltas, D1, D2 e D3, que são utilizadas para se reconstruir a versão desejada. Apenas a versão mais atual é armazenada de maneira integral.

Existem sistemas de controle de versões centralizados e distribuídos. No primeiro caso há apenas um repositório central que guarda todos os IC's e pode ser acessado via LAN ou WAN para a recuperação de um arquivo ou mais (*chekout*).

Após os arquivos serem modificados eles podem ser enviados novamente para o repositório (*commit*). Na topologia distribuída, o usuário do sistema baixa todo o repositório para sua máquina, criando assim um repositório local (*clone*). A partir disso o usuário pode modificar os IC's localmente em um processo idêntico ao sistema de controle de versões centralizado, porém modificando apenas seu repositório local. Para as mudanças sejam compartilhadas com os outros usuários é necessário juntar as modificações do repositório local com o repositório remoto (*push*) (OTTE, 2009).

Otte (2009) descreve 3 exemplos de sistemas de controle de versão:

1. *Concurrent Versions System* (CVS): um dos primeiros a utilizar o modelo centralizado. Criado por Dick Grune na Universidade Livre de Amsterdã, no ano de 1984. Foi usado pelo SourceForge[10] para hospedar mais de cem mil projetos. Limitações: utiliza arquivos RCS, não é capaz de criar deltas de arquivos binários, não suporta *commits* atómicos, não há como determinar se o repositório está em um estado inconsistente, não suporta que arquivos sejam movidos e renomeados de sem perder o histórico dos arquivos, entre outros;

2. *Subversion* (SVN): foi planejado como um sucessor gratuito do CVS. É basicamente o CVS sem suas falhas. Foi iniciado pela CollabNet em 2000 e lançado em Agosto de 2001. Diferenças em relação ao CVS: não utiliza RCS, *commits* são atómicos, suporta arquivos binários, arquivos podem ser renomeados e movidos, entre outros;

3. Git: iniciado em 2005, quando o processo de desenvolvimento do kernel Linux perdeu seu sistema de controle de versão, no fim da licença livre do serviço BitKeeper. Após isso, Linus Torvalds desejava um sistema de controle de versão que: fosse confiável, tivesse um bom desempenho, fosse distribuído e que não fosse como o CVS. Sem opções para satisfazer suas necessidades, ele criou o seu próprio sistema de controle de versões, o Git. Algumas de suas propriedades

[10] SourceForge é um repositório de código fonte baseado em Web. Ele atua como um centro para desenvolvedores gerenciarem projetos livres e de código aberto colaborativamente.

são: não utiliza deltas para armazenar os arquivos, armazena *snapshots* de todos os arquivos em uma estrutura de árvore, rastreia conteúdo e não arquivos, pode importar repositórios completos de outros sistemas de controle de versão, possui poucos comandos que precisam de conexão com a Internet: git clone, para iniciar um repositório de um servidor; git fetch, para obter todas as revisões que o usuário ainda não possui de um servidor; git pull, é um git fetch seguido de um git merge, combinando as mudanças baixadas no ramo atual; git push, inclui as mudanças feitas pelo usuário no repositório do servidor.

2.4.2 Pull request

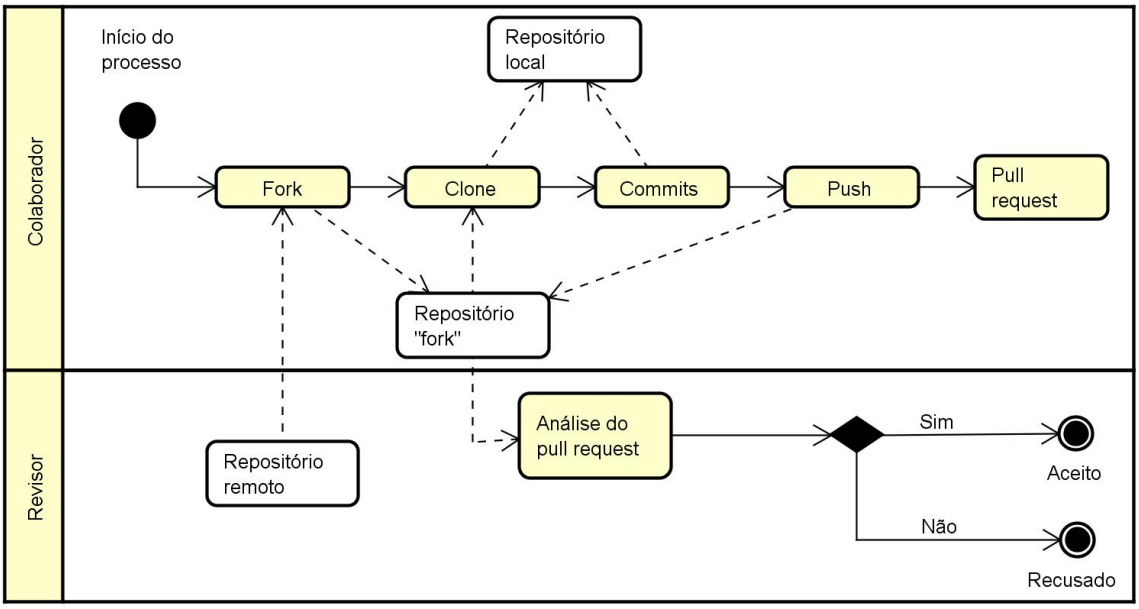
A descrição dada pelo GitHub[11] sobre *pull requests* é a seguinte:

> Os *pull requests* permitem que você conte aos outros sobre as mudanças que você enviou para um repositório no GitHub. Uma vez que um *pull request* é aberto, você pode discutir e analisar as possíveis mudanças com os colaboradores e adicionar *commits* de acompanhamento antes que as alterações sejam incorporadas no repositório.

Como descrito por Gousios *et al.* (2014), o repositório central dos projetos em serviços como GitHub e Bitbucket não são compartilhados com os desenvolvedores, sendo necessário a realização do *fork* do repositório para que seja feitas modificações nos IC's do projeto. As modificações são realizadas no repositório local do desenvolvedor e quando estivem prontas para serem submetidas é necessário a criação de um *pull request* solicitando a integração dos *commits* locais no repositório central. Com a criação do *pull request*, os desenvolvedores responsáveis pelo repositório central são notificados da necessidade de revisar o código que foi submetido e realizar a integração com o repositório central. Caso as modificações sejam consideradas insatisfatórias, novas modificações podem ser requisitadas e o desenvolvedor pode adicionar mais *commits* ao *pull request*. A Figura 5 ilustra o processo do *pull request.*

[11] https://help.github.com/articles/about-pull-requests/

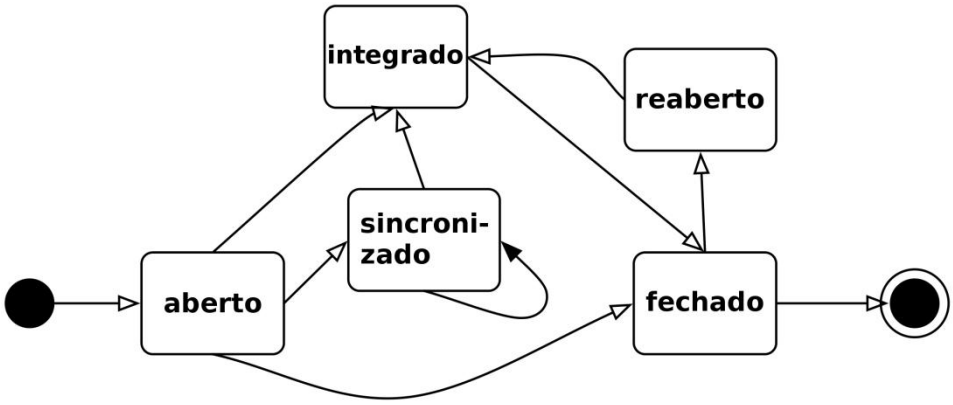
Figura 5. O processo do *pull request*



Fonte: Adaptado de Soares (2017).

Como abordado por Gousios *et al.* (2014), o GitHub oferece um tratamento especial aos *pull requests*. Seu site apresenta funcionalidades que permitem que usuários externos realizem facilmente o *fork* de projetos, automatizam a geração de *pull requests* através de comparações automáticas de ramificações de projetos e atualizam o *pull request* caso novas mudanças sejam realizadas no repositório *fork*. Cada *pull request* apresenta um estado que é atualizado automaticamente pelo GitHub conforme os usuários manipulam o *pull request*. A Figura 6 mostra os possíveis estados de um *pull request*.

Figura 6. Diferentes estados de um *pull request* no GitHub



Fonte: Adaptado de Gousios (2014).

Gousios *et al.* (2014) também mostrou que existem diferentes maneiras para os desenvolvedores que são parte do time principal do projeto aceitarem um *pull request*:

1. **Através das funcionalidades do GitHub.** O GitHub é capaz de identificar se o *pull request* pode ser integrado sem conflitos com o repositório. Dessa forma o GitHub pode aplicar os *commits* automaticamente, preservando os informações sobre o autor e histórico.

2. **Utilizando a função *merge* do Git.** Quando não for possível a integração automática do *pull request*, ou tal ação não seja permitida pelas políticas do projeto, pode-se utilizar as seguintes funcionalidades do sistema Git:

    a. Integração de ramificações. A ramificação remota que contém os *commits* do *pull request* é integrada em uma ramificação local, que é então enviada para o repositório central ou publicada para que seja adicionado mais modificações feitas por outros desenvolvedores. A informações sobre autoria e histórico são mantidas, mas o GitHub não detecta e não registra o evento de integração.

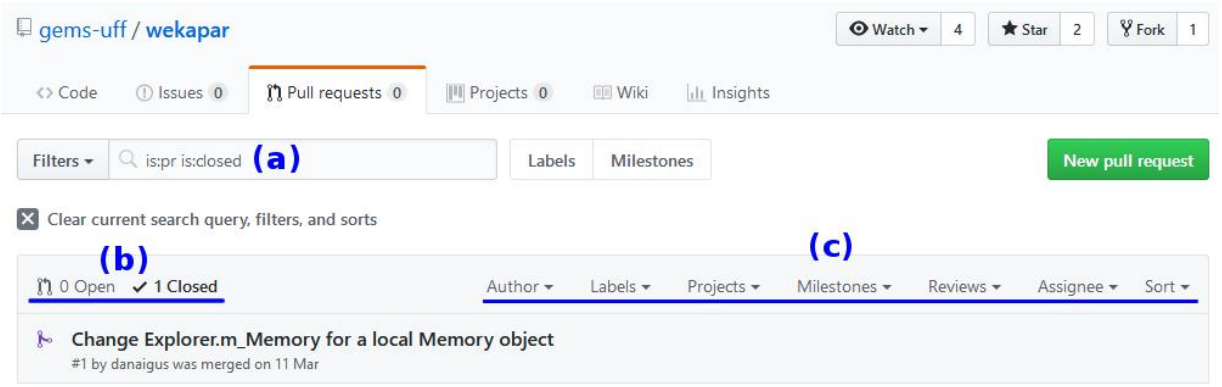
    b. Cherry-picking. Neste método, o revisor escolhe *commits* específicos do *pull request* para serem integrados no repositório. A informações de histórico acaba sendo perdida, porém a informações de autoria são mantidas.

3. **Realizando o *commit* das modificações.** O revisor cria uma diferença textual entre o conteúdo do *pull request* e o repositório central, que é integrada ao repositório. Todas as informações sobre autoria e histórico são perdidas.

Na Figura 7 é mostrado a interface do GitHub para a listagem de *pull requests* do projeto WekaPAR. É possível filtrar os resultados em *issues* e *pull requests*

(Figura 7 (a)), ver quantos *pull requests* estão abertos e fechados (Figura 7 (b)) e classificar os resultados utilizando diferentes parâmetros (Figura 7 (c)).

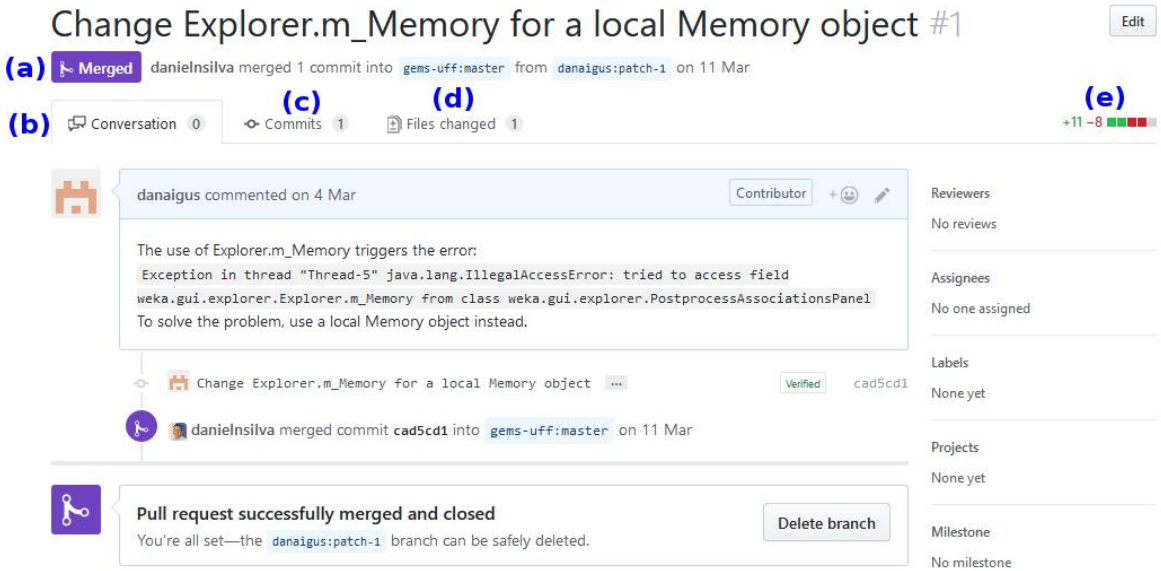
Figura 7. *Pull requests* do projeto WekaPAR no GitHub



Fonte: Autoria própria.

A Figura 8 mostra a discussão sobre o *pull request* enviado para o projeto WekaPAR. O *pull reques*t mostrado possui o estado de "integrado" (Figura 8 (a)), não houve discussão sobre o seu conteúdo (Figura 8 (b)), possui apenas 1 *commit* (Figura 8 (c)), foi modificado apenas 1 arquivo (Figura 8 (d)) e 11 linhas foram adicionadas e 8 removidas (Figura 8 (e)).

Figura 8. Tela de discussão do *pull request* no GitHub



Fonte: Autoria própria.

Os *pull requests,* como um modelo de desenvolvimento distribuído, formaram um novo método para a colaboração no desenvolvimento de softwares. O seu diferencial está na dissociação do esforço de desenvolvimento da decisão de

incorporar os resultados das modificações no código-fonte, permitindo uma melhor distribuição do trabalho realizado (GOUSIOS *et al.*, 2014).

2.5 CONCLUSÃO

A engenharia de software aborda um amplo espectro do desenvolvimento de software. Como um processo que lida com todo o clico de vida dos softwares, é essencial que seus métodos, atividades e processos sejam conhecidos e seguidos corretamente para que o resultado do desenvolvimento de software tenha sucesso. Este capítulo abordou os temas de engenharia de software que são essenciais para o entendimento e desenvolvimento deste trabalho.

No próximo capitulo é abordado o campo de mineração de dados, explorando o processo de KDD, se aprofundando na tarefa de regras de associação e a ferramenta Weka, que é fundamental para a ferramenta proposta pelo trabalho.

3 MINERAÇÃO DE DADOS

Atualmente é comum afirmar que “vivemos na era da informação”. Porém uma afirmação mais correta seria “vivemos na era dos dados”. Todos os dias, uma quantidade massiva de dados é despejada na rede mundial de computadores, World Wide Web, e em vários dispositivos de armazenamento de dados, que vão desde dados sobre negócios, sociedade, ciência e engenharia, medicina, assim como quase todos os aspectos da vida diária. Essa grande disponibilidade de dados gerou uma necessidade por ferramentas poderosas e versáteis para a descoberta automática de informações valiosas dentro da enorme quantidade de dados disponíveis e para transformar esses dados em conhecimento organizado. Esta necessidade levou ao surgimento da mineração de dados (HAN; KAMBER;PEI, 2011).

A mineração de dados (*Data mining*) pode ser vista como o resultado da evolução natural da tecnologia da informação e ser definida como a “extração de padrões interessantes (não-trivial, implícito, previamente desconhecido e potencialmente útil) ou de conhecimento a partir de grandes quantidades de dados” (HAN; KAMBER; PEI, 2011, tradução nossa), assim como a “exploração e análise de grandes quantidades de dados a fim de descobrir padrões e regras significativas” (BERRY; LINOFF, 2004, p. 7, tradução nossa).

Han, Kamber e Pei (2011) descrevem que a mineração de dados, como um domínio direcionado a um grande leque de aplicações, incorporou muitas técnicas

de outros domínios. Eles também citaram algumas disciplinas que tem grande influência no desenvolvimento de métodos de mineração de dados, como pode ser visto na Figura 9. Dentro das disciplinas citadas, 4 se destacam significativamente:

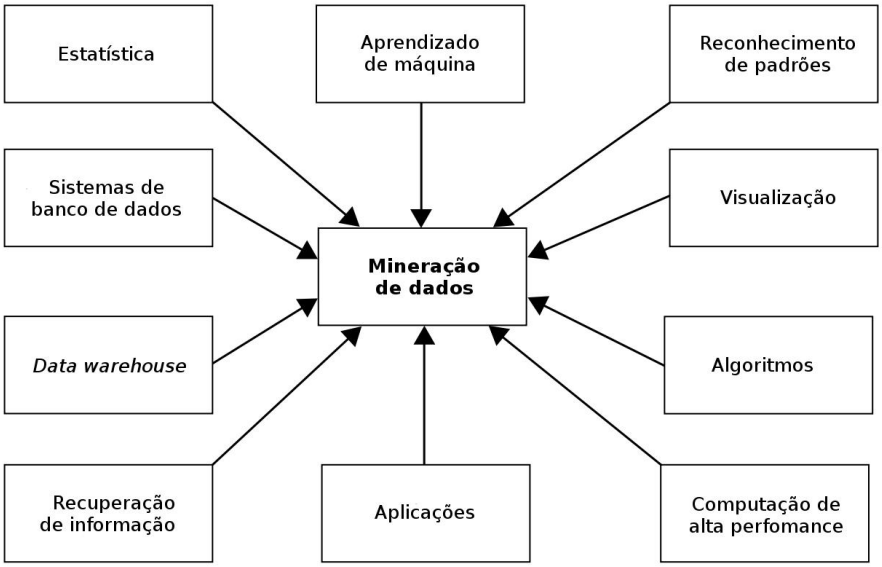
1. **Estatística.** Como uma disciplina que estuda a coleta, análise, interpretação e apresentação de dados, é inerente que exista uma conexão profunda entre estatística e mineração de dados. Os métodos estatísticos podem ser usados para resumir ou descrever uma coleção de dados. Eles são úteis para mineração de padrões de dados, bem como para a sua compreensão.

2. **Aprendizado de máquina.** Investiga como os computadores podem aprender, ou melhorar seu desempenho, com base em dados. Sua principal área de pesquisa é fazer com que programas de computador aprendam automaticamente a reconhecer padrões complexos e tomar decisões inteligentes com base em dados. Alguns de seus problemas estão altamente ligado a mineração de dados, como os campos de aprendizagem supervisionada, não supervisionada e ativa.

3. **Sistemas de banco de dados e data warehouses.** As pesquisas nessa área estabeleceram princípios em modelos de dados, linguagens de consulta, métodos de indexação e acesso, entre outros. Sistemas de bancos de dados apresentam alta escalabilidade no processamento de conjuntos de dados muito grandes e relativamente estruturados, o que auxilia as tarefas de mineração que precisam lidar com grandes números de dados ou até mesmo dados em tempo real.

4. **Recuperação de informação.** Trata-se de pesquisar por documentos ou por informações em documentos, que podem ser textos ou multimídia e estar hospedados na Web. A recuperação de informação pressupõe que os dados sob pesquisa são desestruturados e as consultas são formadas principalmente por palavras-chave. Ao integrar esta disciplina às técnicas de mineração de dados, podemos encontrar os principais tópicos em uma coleção de documentos e, para cada documento na coleção, os principais tópicos envolvidos.

Figura 9. Domínios que auxiliam a mineração de dados



Fonte: Adaptado de Han, Kamber e Pei (2011, p. 23).

A mineração de dados pode ser vista tanto como uma etapa do processo de descobrimento de conhecimento em bases de dados (*Knowledge Discovery in Databases* - KDD), assim como um sinônimo deste. O processo de KDD será aprofundado na próxima seção.

3.1 *KNOWLEDGE DISCOVERY IN DATABASES* - KDD

De acordo com Fayyad, Piatetsky-Shpiro e Smyth (1996), KDD se refere ao processo geral de descoberta de conhecimento útil a partir de dados, e mineração de dados se refere à um passo em particular neste processo. Os passos adicionais no processo de KDD, em relação ao passo de mineração de dados, são essencialmente para assegurar que conhecimento útil será retirado dos dados. Em um nível abstrato, o campo de KDD se preocupa com o desenvolvimento de métodos e técnicas para retirar sentido de dados.

O processo de KDD é interativo e iterativo, envolvendo vários passos com muitas decisões feitas pelo usuário. Fayyad, Piatetsky-Shpiro e Smyth (1996) descrevem algumas de suas etapas básicas:

1. **Desenvolver um domínio.** É necessária uma compreensão do domínio da aplicação e que seja identificado o objetivo do processo de KDD para o ponto de vista do cliente.

2. **Selecionar um conjunto de dados.** Deve-se selecionar uma base de dados ou um subconjunto de variáveis e amostras de dados que serão utilizadas para a descoberta de conhecimento.

3. **Limpeza e pré-processamento dos dados.** Suas operações básicas envolvem a remoção de ruídos, se necessário, a coleta de informações necessárias para modelar ou representar o ruído, a decisão de estratégias para lidar com campos de dados em branco, e por fim a contabilização de informações de sequência de tempo e mudanças conhecidas.

4. **Redução e projeção de dados.** É encontrar recursos úteis para representar os dados de acordo com o objetivo da tarefa. Através de métodos de redução de dimensões ou de transformação, pode-se reduzir o número efetivo de variáveis que serão consideradas. Também podem ser encontradas representações invariantes para os dados.

5. **Ajustar os objetivos.** Os objetivos do processo de KDD devem ser alinhados à uma tarefa de mineração, como classificação, agrupamento e regras de associação.

6. **Análise exploratória e seleção de modelo e hipótese.** É a escolha do algoritmo de mineração de dados e a seleção do método que será utilizado para a procura de padrões de dados. Esse processo inclui a decisão de quais modelos e parâmetros podem ser apropriados e combinação de um método particular de mineração de dados com os critérios gerais do processo KDD.

7. **Mineração de dados.** É a busca por padrões de interesse em uma determinada forma de representação ou um conjunto de representações, incluindo regras de classificação, classificação e

agrupamento. O usuário pode direcionar significativamente o método de mineração de dados realizando as etapas anteriores.

8. **Interpretar os padrões minerados.** Esta etapa envolve a visualização dos padrões e modelos extraídos ou a visualização dos dados dados os modelos extraídos. Também é possível retornar para um dos passos anteriores para uma iteração adicional.

9. **Agir com o conhecimento descoberto.** Neste ponto o conhecimento já foi extraído e deve ser utilizado para decisões futuras em outros sistemas ou simplesmente ser documentado e reportado para as partes interessadas. Também pode ser necessário a verificação e resolução de conflitos com conhecimentos que anteriormente eram dados como verdadeiros.

3.2 TAREFAS DE MINERAÇÃO

De acordo com Han, Kamber e Pei (2011), de maneira geral, as tarefas de mineração de dados podem ser divididas em duas categorias: descritivas e preditivas. As tarefas descritivas de mineração caraterizam propriedades desconhecidas dos dados em uma base de dados. Já as tarefas preditivas de mineração executam induções sobre os dados atuais, para descobrir o comportamento de dados futuros.

Dentre as tarefas de mineração que podem ser realizadas, mostradas por Berry e Linoff (2004), podem ser destacadas a classificação, o agrupamento e as regras de associação. A primeira tarefa é categorizada como preditiva, e as duas ultimas como descritivas, respectivamente.

A tarefa de classificação, para Berry e Linoff (2004), consiste em examinar as características de um conjunto de dados recém-apresentado e atribuí-lo a um conjunto pré-definido de classes. Os dados a serem classificados são geralmente representados por registros em uma tabela de banco de dados ou um arquivo, e a

tarefa de classificação consiste em adicionar uma nova coluna com algum tipo de dado que identifique a classe. Para Han, Kamber e Pei (2011), classificação é o processo de encontrar um modelo (ou função) que distingue classes ou conceitos de dados. Modelo este que é gerado durante a etapa de treino, onde um algoritmo de classificação constrói um modelo classificador com base na análise de um conjunto de dados de treinamento (dados para os quais os rótulos de classe são conhecidos). Após o treino é necessário estimar a precisão preditiva do classificador, sendo esta a etapa de teste. Para que estimativa seja precisa, deve-se utilizar um conjunto de dados de teste independente do conjunto de dados usado para treino. A acurácia do classificador em um conjunto de teste é a porcentagem de dados corretamente classificados pelo modelo. O modelo gerado é utilizado para prever o rótulo da classe dos objetos para os quais o rótulo da classe é desconhecido.

Segundo Han, Kamber e Pei (2011), a atividade de agrupamento realiza a analise de dados sem consultar suas classes. Em muitos casos os dados podem não conter rótulos que informem suas classes. O agrupamento pode ser usado para gerar rótulos de classe para um grupo de dados. Os dados são agrupados com base no princípio de maximizar a similaridade intraclasse e minimizar a similaridade interclasse. Portanto conjuntos de dados são formados para que os dados dentro de um grupo tenham uma alta semelhança entre si, mas que sejam bastante diferentes dos dados em outros grupos.

Para Han, Kamber e Pei (2011), as regras de associação são padrões que ocorrem frequentemente em uma base de dados. Existem muitos tipos de padrões frequentes, incluindo conjuntos de itens frequentes. Um conjunto de itens frequentes geralmente se refere a um conjunto de itens que constantemente aparecem juntos em um conjunto de dados transacionais. Já Berry e Linoff (2004) consideram que as regras de associação representam padrões nos dados sem um alvo especificado. Desta forma, elas são um exemplo de mineração de dados não direcionada. Se os padrões fazem sentido é deixado para a interpretação humana.

3.3 REGRAS DE ASSOCIAÇÃO

A mineração de regras de associação de bases de dados "pretende extrair correlações interessantes, padrões frequentes, associações ou estruturas causais entre conjuntos de itens nos bancos de dados de transações ou outros repositórios de dados" (KOTSIANTIS; KANELLOPOULOS, 2006, tradução nossa).

Supondo que $I = \{i_1, i_2, ..., i_n\}$ seja um conjunto de dados e que D seja um conjunto de transações em um banco de dados em que cada transação é um conjunto de itens contidos em I. Uma regra de associação é uma implicação na forma $A \Rightarrow B$, onde $A \subset I$, $B \subset I$ e $A \cap B \neq \varnothing$. O que está do lado esquerdo da regra (A) é o antecedente e o que está do lado direito (B) é o consequente. A regra $A \Rightarrow B$ possui um *suporte* s, que é a porcentagem de transações em D que contém tanto os itens de A, como os de B. Tal regra também possui uma *confiança* c, sendo esta a porcentagem de transações em D que contem A, que também contem B (HAN; KAMBER; PEI, 2011;SILVA, 2016). Isso nos leva a:

$$Suporte(A \rightarrow B) = P(A \cup B) \tag{1}$$

$$Confiança(A \rightarrow B) = P(A \mid B) = \frac{Suporte(A \cup B)}{Suporte(A)} \tag{2}$$

Han, Kamber e Pei (2011) definem que regras que satisfaçam tanto um limite de suporte mínimo quanto um limite de confiança mínimo são chamados de **fortes**. A frequência de ocorrência de um conjunto de itens é o número de transações que contêm o conjunto de itens. Se o suporte relativo de um conjunto de itens I satisfaz um limite mínimo de suporte especificado, então I é um conjunto de itens **frequente**. De maneira geral. A mineração de regras de associação pode ser vista como um processo de dois passos:

1. **Encontrar todos os conjuntos de itens frequentes:** por definição, cada um dos conjuntos de dados terá frequência maior do que o suporte mínimo determinado.

2. **Gerar regras de associação fortes a partir dos conjuntos de itens frequentes:** essas regras devem satisfazer o suporte e confiança mínimos.

Como o segundo passo é muito menos complexo do que o primeiro, o desempenho geral da mineração de regras de associação é determinado pelo primeiro passo.

Geralmente um número muito grande de regras é gerado, o que faz necessário saber quais regras são realmente interessantes. Como as medidas de suporte e confiança são insuficientes para filtrar regras de associação desinteressantes, pois não medem a força real (ou falta de força) da correlação e implicação entre A e B, deve-se utiliza as chamadas medidas de interesse (HAN; KAMBER; PEI, 2011;SILVA, 2016).

Lift é um exemplo de medida de interesse de correlação simples que é dada da seguinte forma: a ocorrência do conjunto de itens A é **independente** da ocorrência do conjunto de itens B se $P(A \cup B) = P(A)P(B)$; caso contrário, os conjuntos de itens A e B são **dependentes** e **correlacionados** como um evento. O *lift* entre a ocorrência de A e B pode ser medido pela equação:

$$lift(A, B) = \frac{P(A \cup B)}{P(A)P(B)} \tag{3}$$

Se o *lift* for menor do que 1, então a ocorrência de A está negativamente correlacionada com a ocorrência de B, o que significa que a ocorrência de uma provavelmente leva à ausência do outra. Se o *lift* resultante for superior a 1, então A e B estão positivamente correlacionados, o que significa que a ocorrência de um implica na ocorrência do outro. Se o *lift* resultante for igual a 1, então A e B são independentes e não há correlação entre eles.

3.4 FERRAMENTA WEKA

Como Silva (2016) descreve, a mineração de dados é composta por técnicas de alta complexidade, e que requerem a utilização de ferramentas auxiliares como uma parte imprescindível do processo de descoberta de conhecimento em bases de dados. Sendo assim, existem diversas ferramentas disponíveis para serem utilizadas na aplicação das técnicas de mineração de dados, podendo elas serem tanto software de uso comercial, de uso livre e até mesmo de código aberto.

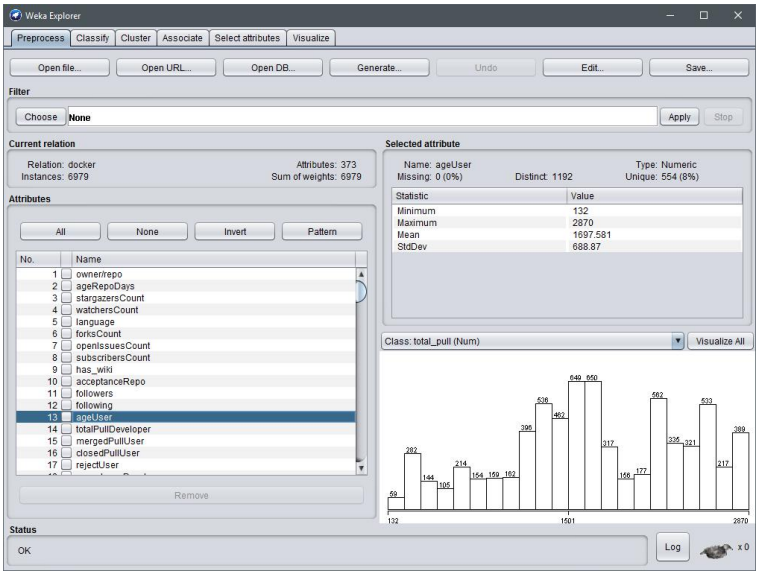
Uma das opções disponíveis é a ferramenta Weka. Segundo Frank, Hall e Witten (2016), a ferramenta Weka é uma coleção de algoritmos de aprendizagem de máquina e ferramentas de pré-processamento de dados para tarefas de mineração de dados. Weka foi desenvolvida na Universidade de Waikato, na Nova Zelândia. Seu nome significa Ambiente Waikato para Análise de Conhecimento (*Waikato Environment for Knowledge Analysis*). É escrita na linguagem de programação Java e distribuída sobre a licença GNU GPL.

Ela inclui uma variedade de ferramentas para transformar conjuntos de dados, como os algoritmos para discretização e amostragem, métodos para os principais problemas de mineração de dados: regressão, classificação, agrupamento, mineração de regras de associação e seleção de atributos. Também são fornecidas ferramentas de visualização de dados (FRANK;HALL;WITTEN, 2016).

De acordo com Frank, Hall e Witten (2016), a ferramenta Weka pode ser utilizada através de cinco diferentes interfaces de usuário: *Explorer*, que dá acesso a todas as ferramentas usando menus e formulários; *Knowledge Flow*, que permite a criação de configurações para processamento de dados em fluxo; *Experimenter*, auxilia a responder uma pergunta prática ao aplicar técnicas de classificação e regressão; *Workbench*, uma interface gráfica unificada que combina as outras três; Interface de Linha de Comandos, tem acesso a todos os recursos através comandos textuais.

A Figura 10 mostra a interface *Explorer* para o processo de pré-processamento dos dados.

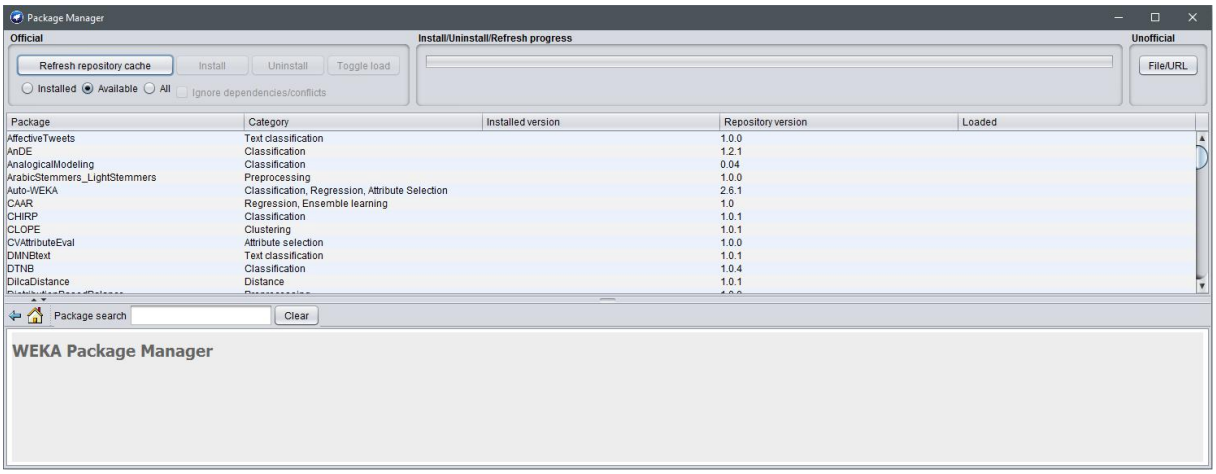
Figura 10. Interface Explorer para pré-processamento



Fonte: Autoria própria.

Uma das funcionalidades da ferramenta Weka é a utilização de um sistema de gerenciamento de pacotes que fornecem novas funcionalidades para o usuário. Tal sistema permite ao usuário navegar e instalar seletivamente os pacotes de seu interesse. Esse sistema também facilita, para desenvolvedores interessados, o processo de contribuição para a ferramenta Weka. Um contribuidor de um pacote/extensão é responsável por manter seu código e hospedar o arquivo instalável, cabendo à Weka apenas o acompanhamento dos metadados do pacote (FRANK;HALL;WITTEN, 2016). Na Figura 11 é mostrada a interface gráfica do gerenciador de pacotes Weka.

Figura 11. Interface do gerenciador de pacotes Weka



Fonte: Autoria própria.

3.5 CONCLUSÃO

A mineração de dados é uma área responsável pela geração de conhecimento através do processamento de dados. É uma etapa chave do processo de KDD e de grande importância no cenário atual. Para realizar as tarefas de mineração é necessário a utilização de ferramentas como a Weka, que auxiliam os pesquisadores através da utilização de poder computacional. Este capítulo conduziu uma revisão bibliografia sobre os conceitos necessários para o entendimento e o sucesso durante a realização deste trabalho.

No próximo capítulo é abordado o desenvolvimento da solução proposta. É mostrado como foi dada sua definição, quais técnicas de desenvolvimento foram utilizadas e sua normalização como um pacote para a ferramenta Weka.

4 PR-DISCOVER

Para a solução desenvolvida neste trabalho, duas opções foram consideradas inicialmente: um programa independente, que utilizaria bibliotecas para a realização dos passos de mineração de dados ou uma extensão integrada à ferramenta Weka. Durante a fase de estudo de viabilidade, definiu-se que o desenvolvimento de um programa independente não oferecia vantagens sobre a criação de uma extensão integrada com a Weka, pois a Weka oferece recursos que facilitam a construção e distribuição de pacotes, como arquivos de configuração e construção, permitindo a aplicação sistemática do desenvolvimento do reúso.

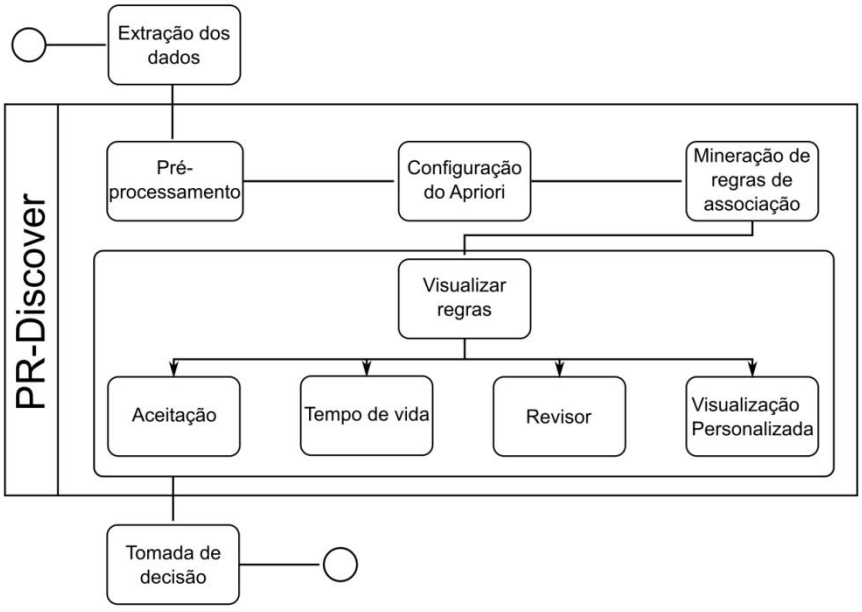
Outro fator que viabilizou o desenvolvimento de uma extensão foi a existência da extensão WekaPAR, que possui funcionalidades para visualização de regras de associação geradas com a ferramenta Weka, tornando possível reaproveitar o código já existente para as funcionalidades de visualização que são requisitos da solução proposta por este trabalho (SILVA, 2016).

A extensão proposta possui funcionalidades para automatizar algumas atividades da mineração de dados, a saber: o pré-processamento da bases de dados, onde é necessário discretizar os atributos numéricos, tornado-os atributos nominais, pois o algoritmo Apriori[12] não é capaz de processar dados numéricos. Tal discretização deve seguir parâmetros pré-estabelecidos, que foram definidos por Soares (2017).

12 Apriori é um algoritmo para mineração de conjuntos de itens frequentes e descoberta de regras de associação em bancos de dados transacionais.

Na Figura 12 é mostrado o fluxo do processo de descoberta de regras de associação com a utilização da extensão sugerida neste trabalho. A extração dos dados sobre o repositório que será utilizado na mineração é fornecido pelo usuário, utilizando a ferramenta GHTorrent. Após isso a extensão é utilizada para realizar automaticamente o pré-processamento dos dados e a configuração dos parâmetros de mineração do algoritmo Apriori, para que seja gerado o maior número de regras de associação possíveis. Com o resultado da execução do algoritmo, uma visualização de todas as regras é gerada pela extensão. A partir disso é possível escolher entre visualizações que buscam mostrar fatores que influenciam na aceitação, no tempo de vida e na atribuição de revisores dos *pull requests*, assim como uma visualização personalizada das regras de associação. Com tais funcionalidades, os gerentes de projeto podem utilizar as informações geradas em sua tomada de decisão, finalizando assim o processo de descoberta de conhecimento. A Figura 12 mostra o processo de mineração sobre *pull requests* utilizando a solução proposta por este trabalho.

Figura 12. Processo de mineração sobre *pull requests*



Fonte: Autoria própria.

As funcionalidades da extensão são voltadas para facilitar as etapas de mineração que ocorrem após a extração dos dados. Sendo assim foi utilizado os conhecimentos sobre engenharia de requisitos para se especificar o funcionamento

da extensão através de um documento de requisitos. No Quadro 3 é mostrado os requisitos funcionais que foram definidos para a extensão proposta.

Quadro 3. Requisitos Funcionais

ID	Funcionalidade	Necessidades	Prioridade
RF01	Carregar base de dados	A extensão deve carregar a base de dados que foi coletada	Alta
RF02	Pré-processamento automático	A extensão deve realizar o tratamento dos dados de maneira automática (remoção de atributos não interessantes e discretização dos dados numéricos)	Alta
RF03	Extração automática de regras de associação	A extensão deve fazer a mineração das regras automaticamente	Alta
RF04	Visualizar regras	A extensão deve mostrar as regras para o usuário	Alta
RF05	Diferentes visualizações	A extensão deve mostrar as regras sobre diferentes perspectivas	Média
RF06	Perspectiva “aceitação”	A extensão deve mostrar regras específicas sobre a aceitação dos *pull requests*	Alta
RF07	Perspectiva “revisor”	A extensão deve mostrar regras específicas sobre a atribuição de revisores dos *pull requests*	Média
RF08	Perspectiva “lifetime”	A extensão deve mostrar regras específicas sobre o tempo de vida dos *pull requests*	Média
RF09	Instalação pela Weka	A extensão deve ser instalado pelo package manager da Weka	Alta
RF10	Exportar resultados	A extensão deve exportar os resultados da mineração	Baixa
RF11	Salvar resultados	A extensão deve salvar os resultados da mineração para que possam ser abertos novamente	Baixa
RF12	Abrir resultados salvos	A extensão deve abrir os resultados salvos de minerações anteriores	Baixa
RF13	Salvar dados pré-processados	A extensão deve permitir que os dados que foram pré-processados sejam salvos em um arquivo	Média

ID	Funcionalidade	Necessidades	Prioridade
RF14	Configurar visualização por atributos das regras	A extensão deve permitir configurar uma visualização personalizada com base em atributos das regras	Baixa
RF15	Configurar visualização por métricas das regras	A extensão deve permitir configurar uma visualização personalizada com base em métricas das regras	Baixa

Fonte: Autoria própria.

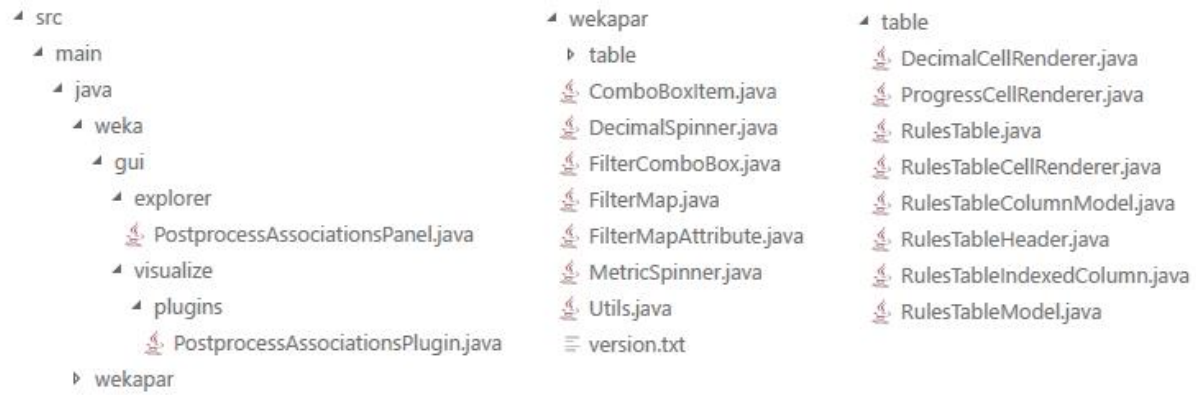
O documento de requisitos completo pode ser encontrado no Apêndice A deste trabalho.

4.1 DESENVOLVIMENTO

Como ficou definido que a solução proposta seria uma extensão para a ferramenta Weka, a linguagem de programação utilizada para sua implementação deve ser a linguagem Java, pois é esta a linguagem utilizada pela ferramenta Weka.

Foi realizada uma análise no código fonte da ferramenta e da extensão WekaPAR, para que fosse identificado os possíveis componentes a serem utilizados durante o desenvolvimento orientado ao reúso. A Figura 13 ilustra a estrutura do código fonte da extensão WekaPAR.

Figura 13. Estrutura do código fonte da extensão WekaPAR



Fonte: Autoria própria.

Nesta fase, decidiu-se pela reutilização da extensão WekaPAR, (classe *PostprocessAssociationsPanel)*, pois ela oferece recursos avançados para a

visualização de regras de associação, porém foi necessário modificações, com acréscimo de novas funções para atender a missão do software proposto por esta pesquisa. Assim como todos os componentes do diretório *wekapar* foram reutilizados, sem a necessidade de modificações, porque seus componentes são necessários para o funcionamento da classe *PostprocessAssociationsPanel*.

A partir do código fonte da ferramenta Weka, foram utilizadas algumas funções específicas da classe *PrepocessPanel*, que fornecem funcionalidades de carregamento de arquivos e persistência para a extensão em desenvolvimento. A API Weka foi utilizada como uma biblioteca para o acesso aos métodos de pré-processamento e mineração que estão presentes em seu código, necessários para que as tarefas de pré-processamento dos dados e mineração das regras de associação sejam automatizadas.

Na fase de integração dos componentes, foi necessário realizar modificações na classe *PostprocessAssociationsPanel:*

1. Foi acrescentada uma nova função, *prepareData*, que utiliza métodos fornecidos pela API Weka, para a discretização automática dos dados;

2. Criado nova função, *findAssociationRules*, para a mineração de regras de associação utilizando o o algoritmo *Apriori*, fornecido pela API Weka;

3. As funções para carregamento de arquivos e persistência da classe *PrepocessPanel* foram integradas ao código;

4. Foi renomeada para *PullRequestMiningPanel*.

4.2 CRIAÇÃO DO PACOTE WEKA

Um pacote Weka é um arquivo comprimido (tipo “zip”) contendo vários recursos, como código compilado, código fonte, javadocs (documentação), arquivos de descrição do pacote (metadados), bibliotecas de terceiros e arquivos propriedades de configuração. A ferramenta deixa claro que o conceito de pacote

utilizado por ela é diferente do conceito de pacotes Java, que simplesmente definem como as classes devem ser organizadas hierarquicamente.

Os pacotes Weka tem uma estrutura definida que deve ser seguida para que sejam válidos. Tal estrutura é definida no site oficial da ferramenta[13]. Ao aplicar a estrutura necessária na extensão que foi desenvolvida, foi obtido a estrutura vista na Figura 14.

O diretório *doc* contém a documentação (javadoc) das classes que compõem a extensão. O diretório *lib* contém as bibliotecas utilizadas pela extensão. No diretório *src* estão os arquivos de código fonte da extensão, onde a classe desenvolvida precisa seguir a hierarquia de pacotes Java do código fonte da ferramenta Weka. O presença do diretório src é opcional, sendo indicado apenas para pacotes de código aberto. O que é realmente utilizado pela ferramenta Weka é o arquivo JAR que contém as classes do pacote a ser integrado.

Figura 14. Estrutura do pacote da extensão PR-Discover

Fonte: Autoria própria.

Todo pacote Weka deve possuir um arquivo *Description.props,* que fornece os metadados do pacote. Esses metadados incluem o nome do pacote, data, título, categoria, autor, responsável, licença, descrição, link de download, pacotes

13 http://weka.wikispaces.com/How+are+packages+structured+for+the+package+management+system%3F

relacionados e dependências. A Figura 15 ilustra um trecho do arquivo *Description.props* da extensão desenvolvida.

Figura 15. Trecho do arquivo *Description.props*

```
# Package name (required)
PackageName=PR-Discover

# Version (required)
Version=0.1.0

#Date (year-month-day)
Date=2018-03-04

# Title (required)
Title=PR-Discover

# Category (recommended)
Category=Associations

# Author (required)
Author=José Marcos <josemarcosld@gmail.com>
```

Fonte: Autoria própria.

Certos tipos de pacotes podem exigir que arquivos de configuração adicionais estejam presentes como parte do pacote. No caso da extensão PR-Discover, que adiciona uma nova aba na interface de usuário *Explorer* da Weka, é necessário a utilização de um arquivo *Explorer.props*, para que a Weka instancie e mostre o novo painel. Tal arquivo é mostrado na Figura 16.

Figura 16. Arquivo Explorer.props

```
# Explorer.props file.
# Adds the PullRequestMiningPanel to the Tabs key.
# The standalone option makes the tab available without
# requiring the preprocess panel to load a dataset first.

Tabs=weka.gui.explorer.PullRequestMiningPanel:standalone
```

Fonte: Autoria própria.

Em seu site[14], a ferramenta Weka disponibiliza o modelo que um arquivo *build_package.xml*, para ser modificado de acordo com as necessidades do pacote. Através da utilização deste arquivo, em conjunto com a ferramenta Apache Ant[15], é

[14] http://weka.wikispaces.com/How%20are%20packages%20structured%20for%20the%20package%20management%20system%3F

[15] Apache Ant é uma ferramenta para automatizar processos de compilação de software. http://ant.apache.org/

possível automatizar vários passos do processo de montagem dos pacotes Weka, como a criação da documentação do código do pacote, do arquivo JAR e do arquivo zip do pacote. Tal arquivo é ilustrado na Figura 17.

Figura 17. Trecho do arquivo *build_package.xml*

```
<!--
  ================================================================================
  Release making stuff
  ================================================================================
-->

  <target name="init_dist" depends="init_all">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}"/>
  </target>

  <!-- Put everything in ${build}/classes into the ${package}.jar file -->
  <target name="exejar" depends="compile, docs, init_dist" description="Create a binary jar file in ./dist">
    <jar jarfile="${dist}/${package}.jar" basedir="${build}/classes">
    </jar>
  </target>
```

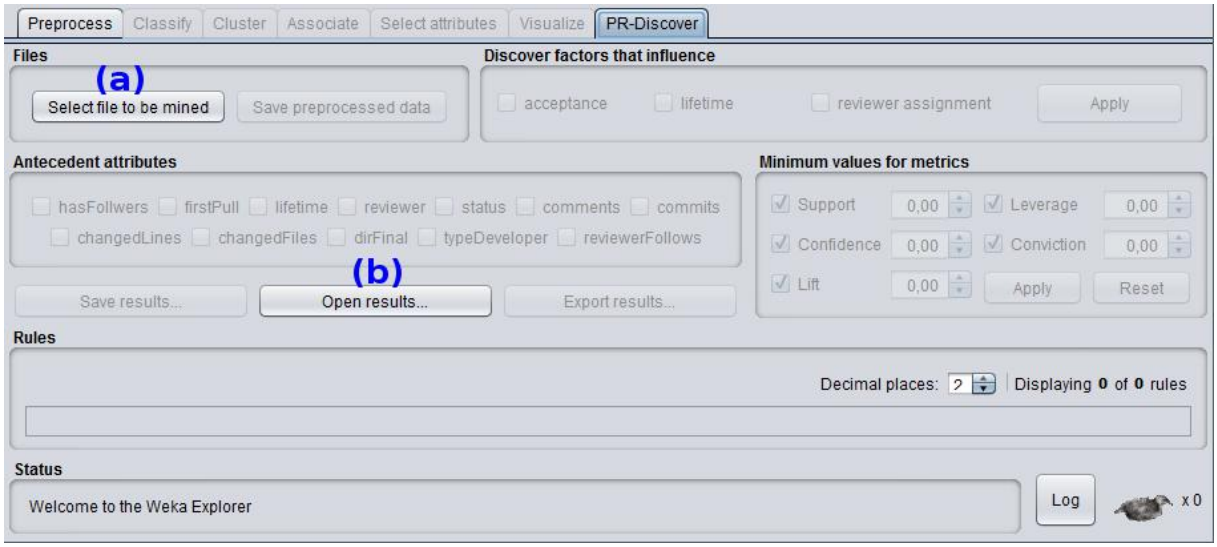
Fonte: Autoria própria.

O pacote zip da extensão PR-Discover foi feito com o auxilio da ferramenta Ant, utilizando o arquivo *build_package.xml* com algumas modificações que se fizeram necessárias.

4.3 VISÃO GERAL

Como a solução proposta é uma ferramenta que automatiza os passos de pré-processamento e mineração de dados, o procedimento para se realizar a mineração das regras de associação através da ferramenta Weka é bastante modificado. Todos os recursos disponibilizados pela solução proposta estão acessíveis através do painel “*Pull request mining*” na interface de usuário *Explorer* da ferramenta Weka. A visualização inicial da interface da extensão desenvolvida pode ser visto na Figura 18.

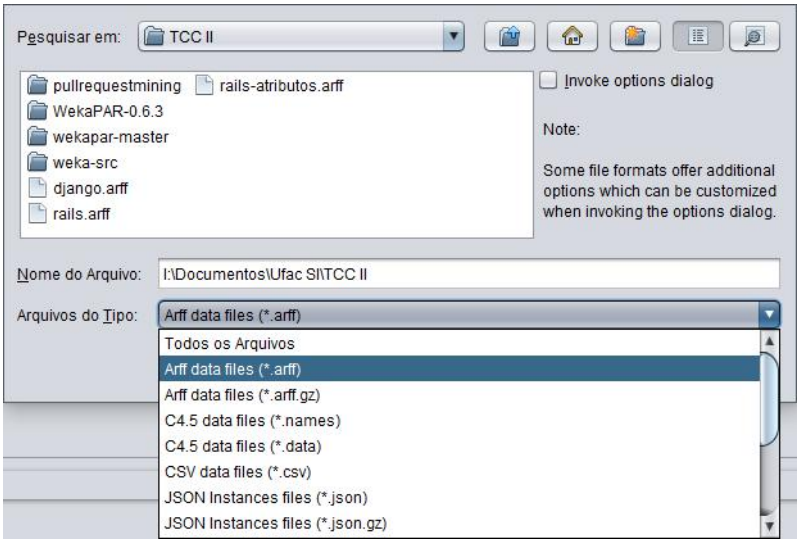
Figura 18. Visualização inicial da extensão PR-Discover



Fonte: Autoria própria.

Através da visualização inicial é possível escolher um arquivo com os dados que serão utilizados para o processo de mineração de regras de associação utilizando o botão "*Select file to be mined*" (Figura 18 (a)), que quando pressionado, abre uma tela com um navegador de arquivos gráfico, como pode ser visto na Figura 19, que permite que se escolha um arquivo. Este arquivo pode estar em diversos formatos, como arff, csv e json, porém existe uma limitação para o funcionamento da solução: os dados sobre o repositório no GitHub devem ter sido extraídos através da ferramenta GHTorrent. Também é possível abrir o resultado de processamentos anteriores através do botão "Open results..." (Figura 18 (b)).

Figura 19. Tela de seleção de arquivo



Fonte: Autoria própria.

Para a realização do pré-processamento da base de dados, a extensão segue as técnicas e métodos utilizados por Soares (2017), que descobriu os atributos relevantes para a descoberta de fatores que influenciam os *pull requests*. Os atributos e seus valores seguem o padrões definidos no Quadro 4.

Quadro 4. Atributos utilizados pela extensão e seus valores

Atributo	Descrição
hasFollowers	Se o contribuidor possui seguidores no GitHub: “verdadeiro” ou “falso”.
firstPull	Se o *pull request* é o primeiro feito pelo solicitante: “verdadeiro” ou “falso”.
lifetime	Quantidade de tempo entre a submissão e o fechamento do *pull request*: “muito curto” = 1 dia; “curto” = 1 a 3 dias; “médio” = 3 a 7 dias; “longo” = 7 a 10 dias; “muito longo” = mais de 10 dias.
reviewerPull	O nome do revisor responsável por revisar o *pull request*.
status	O status final do *pull request*: “aceito” ou “rejeitado”.
comments	Quantos comentários o *pull request* contém: “sem comentários”; “1 comentário”; “alguns comentários” = 2 a 10 comentários; “muitos comentários” = mais que 10 comentário.
commitsPull	O número de commits no *pull request*: “1-commit”; “alguns commits” = 2 a 4 commits; e “”muitos commits” = mais do que 4 commits.
changedLines	O número de linha modificadas: “1 linha”; “algumas linhas” = 2 a 20 linhas; “muitas linhas” = mais do que 20 linhas.
changedFiles	O número de arquivos editados: “1 arquivo”; “alguns arquivos” = 2 a 4 arquivos; “muitos arquivos” = mais do que 4 arquivos.
dirFinal	Diretórios modificados no *pull request*, separado por “|”.
fileNames	Arquivos modificados no *pull request*, separados por “|”.
typeDeveloper	Se o contribuidor é “interno” ou “externo”.
ReviewerFollows ExternalContributor	Se o revisor segue o contribuidor no GitHub: “verdadeiro” ou “falso”.

Fonte: Autoria própria.

Como as etapas de pré-processamento e mineração das regras de associação são automáticas, o usuário apenas aguarda que o processo seja completo para que o resultado seja apresentado. Para que o usuário saiba que o

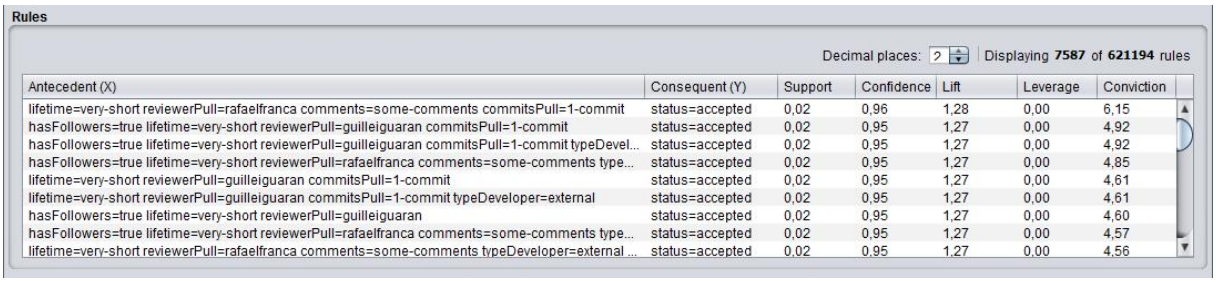
processo está em andamento, ou sobre possíveis erros na execução, a extensão gera mensagens de *feedback* na barra de *status* da ferramenta Weka, como visto na Figura 20.

Figura 20. Exemplo de mensagem de *feedback*

Fonte: Autoria própria.

Após a mineração estar completa os resultados são apresentados graficamente em uma lista no campo "*Rules*", ilustrado na Figura 21.

Figura 21. Regras encontradas na mineração



Antecedent (X)	Consequent (Y)	Support	Confidence	Lift	Leverage	Conviction
lifetime=very-short reviewerPull=rafaelfranca comments=some-comments commitsPull=1-commit	status=accepted	0,02	0,96	1,28	0,00	6,15
hasFollowers=true lifetime=very-short reviewerPull=guilleiguaran commitsPull=1-commit	status=accepted	0,02	0,95	1,27	0,00	4,92
hasFollowers=true lifetime=very-short reviewerPull=guilleiguaran commitsPull=1-commit typeDevel...	status=accepted	0,02	0,95	1,27	0,00	4,92
hasFollowers=true lifetime=very-short reviewerPull=rafaelfranca comments=some-comments type...	status=accepted	0,02	0,95	1,27	0,00	4,85
lifetime=very-short reviewerPull=guilleiguaran commitsPull=1-commit	status=accepted	0,02	0,95	1,27	0,00	4,61
lifetime=very-short reviewerPull=guilleiguaran commitsPull=1-commit typeDeveloper=external	status=accepted	0,02	0,95	1,27	0,00	4,61
hasFollowers=true lifetime=very-short reviewerPull=guilleiguaran	status=accepted	0,02	0,95	1,27	0,00	4,60
hasFollowers=true lifetime=very-short reviewerPull=rafaelfranca comments=some-comments type...	status=accepted	0,02	0,95	1,27	0,00	4,57
lifetime=very-short reviewerPull=rafaelfranca comments=some-comments typeDeveloper=external ...	status=accepted	0,02	0,95	1,27	0,00	4,56

Fonte: Autoria própria.

A partir disso é possível escolher entre visualizações direcionadas para a descoberta de fatores que influenciam os *pull requests* em:

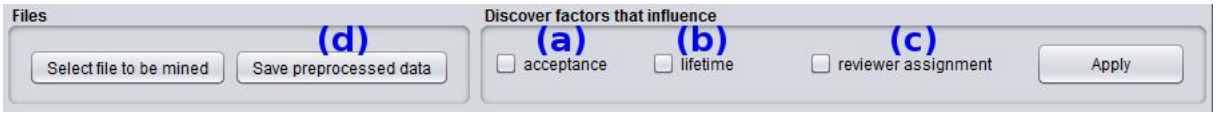
1. Sua aceitação (Figura 22 (a)): mostra regras que possuem apenas *status* como consequente;

2. Seu tempo de vida (Figura 22 (b)): mostra regras que possuem apenas *lifetime* como consequente;

3. Sua atribuição de revisor (Figura 22 (c)): exibe regras que tem apenas *reviewer* como consequente.

Nesse ponto, também é possível salvar um arquivo contendo o resultado do pré-processamento da base de dados que foi utilizada, através do botão "*Save preprocessed data*" (Figura 22 (d)). A Figura 22 mostra a interface para o acesso à funções de visualização e persistência dos dados pré-processados.

Figura 22. Botões para salvar dados e aplicar visualização



Fonte: Autoria própria.

É possível configurar uma visualização personalizada das regras de associação, definindo quais atributos devem estar presentes nas regras como antecedente, através das caixas de seleção presentes no campo "*Antecedent attributes*" (Figura 23 (a)). As métricas de suporte, confiança, *lift* e *leverage* e convicção são outros fatores que podem ser configurados para filtrar as regras que são apresentadas pela extensão, utilizando as opções presentes no campo "*Minimum values for metric*" (Figura 23 (b)). Os resultados da mineração podem ser salvos (Figura 23 (c)) ou exportados (Figura 23 (d)). As funcionalidades de definição de métricas mínimas e persistência dos resultados são uma herança direta da extensão WekaPAR. A Figura 23 mostra a interface que permite o utilização da funções citadas neste parágrafo.

Figura 23. Opções de atributos e de métricas

Fonte: Autoria própria.

4.4 CONCLUSÃO

Este capítulo apresentou de forma geral a extensão PR-Discover. Foi mostrado os requisitos funcionais que nortearam o seu desenvolvimento, assim como se deu o desenvolvimento baseado em reúso e os passos necessários para a finalização da extensão. Se fez necessário que a extensão seguisse o modelo de pacotes que é exigido pela ferramenta Weka, que é explicitado durante o capítulo. A extensão também teve seu funcionamento e principais recursos demonstrados.

O próximo capítulo se trata de uma avaliação realizada com o intuito de validar as funcionalidades e usabilidade da extensão desenvolvida no trabalho.

5 AVALIAÇÃO DA SOLUÇÃO PROPOSTA

O processo de avaliação da solução proposta neste trabalho consistiu em verificar se as funcionalidades da extensão PR-Discover estavam de acordo com o que foi definido no documento de requisitos, e se tais funcionalidades são realmente capazes de auxiliar no processo de descoberta de conhecimento sobre *pull requests*, atendendo as necessidades do usuário.

5.1 VALIDAÇÃO DAS FUNCIONALIDADES IMPLEMENTADAS

Para a validação das funcionalidades implementadas na extensão são utilizados diferentes tipos de experimentos. Estes experimentos buscam mostrar o comportamento da extensão dentro dos cenários de descoberta de fatores de aceitação, tempo de vida e atribuição de revisor dos *pull requests*. Sob a perspectiva de verificação e validação foi utilizada uma abordagem dinâmica, ou seja, foram executados testes funcionais na ferramenta para analisar o comportamento da mesma em relação à entradas e saídas.

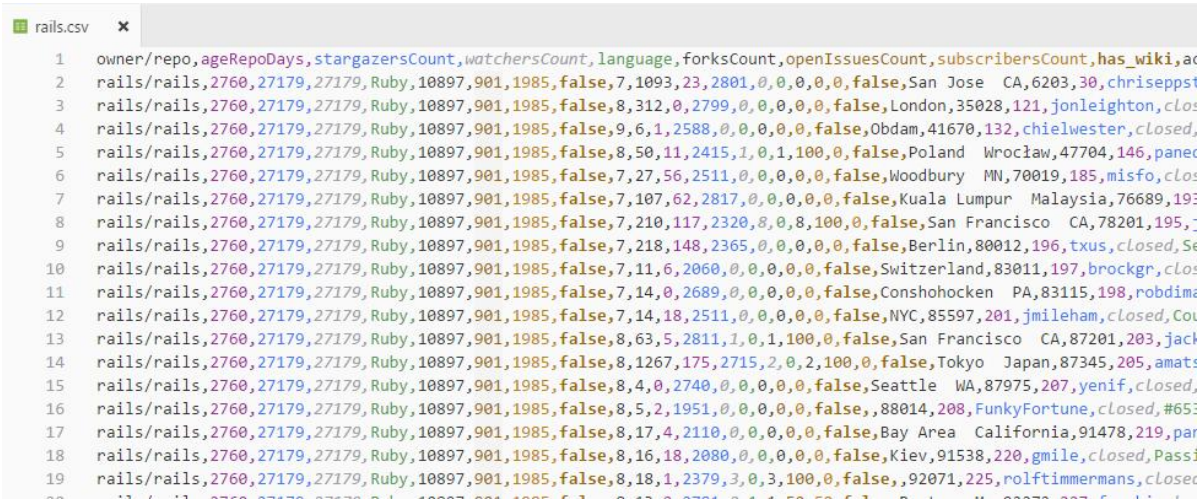
Buscando mostrar a usabilidade real da extensão foi utilizada uma base de dados coletada de um repositório com grande popularidade no serviço GitHub.

5.1.1 Base de dados

Foi escolhida uma base da dados sobre os *pull requests* do repositório GitHub do projeto Rails[16]. Os dados foram extraídos com a utilização da ferramenta GHTorrent, como é requerido para o funcionamento da extensão.

A base de dados do projeto Rails é um arquivo csv que apresenta 345 atributos e 11.728 instâncias. A porcentagem de aceitação dos *pull requests* enviados para o projeto no período analisado é de 74,99%. Os *pull requests* foram feitos por 2.934 contribuidores únicos. Dentre o total de *pull requests*, 570 foram feitos por desenvolvedores internos do projeto e 11.158 foram feitos por colaboradores externos. Uma breve visualização desta base de dados pode ser vista na Figura 24.

Figura 24. Base de dados rails.csv



Fonte: Autoria própria.

5.1.2 Pré-processamento e mineração automática

Nesta etapa, a base de dados escolhida foi carregada na extensão, ainda com os dados “brutos”, e o pré-processamento foi então realizado. A extensão

[16] https://github.com/rails/rails

realizou o tratamento dos dados, onde foram escolhidos apenas os atributos que apresentam influência sobre os *pull requests,* de acordo com o que foi definido por Soares (2017).

Para garantir que os dados foram realmente pré-processados, foi utilizada a função de salvar os dados pré-processados, visto na Figura 22. O resultado encontrado no arquivo arff, próprio da ferramenta Weka, foi uma base de dados totalmente discretizada de acordo com o que foi definido pelo Quadro 4, como pode ser visto na Figura 25.

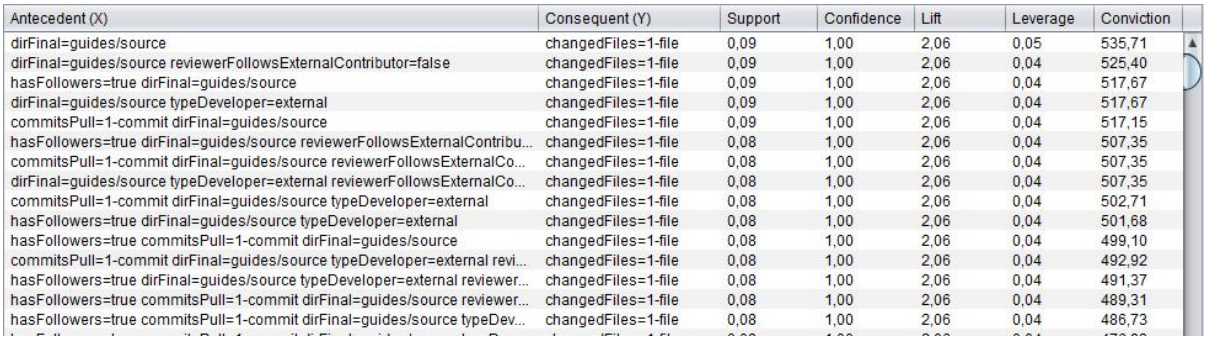
Figura 25. Base de dados pré-processada

Fonte: Autoria própria.

Quanto ao processo de mineração, esta foi automaticamente realizada em sequência ao processo de pré-processamento, sem a necessidade de intervir no funcionamento da extensão. Os resultados da execução do algoritmo Apriori foram apresentados na visualização integrada a extensão.

A extensão define, por padrão, as medidas de suporte e confiança mínimos em 2% e 25%, respectivamente. Com estas medidas, foram encontradas 621.194 regras de associação, com métricas de suporte entre 2% e 95%, confiança entre 25% e 100% e *lift* de 0,44 até 3,8. Um breve visualização das regras pode ser vista na Figura 26.

Figura 26. Regras de associação descobertas



Antecedent (X)	Consequent (Y)	Support	Confidence	Lift	Leverage	Conviction
dirFinal=guides/source	changedFiles=1-file	0,09	1,00	2,06	0,05	535,71
dirFinal=guides/source reviewerFollowsExternalContributor=false	changedFiles=1-file	0,09	1,00	2,06	0,04	525,40
hasFollowers=true dirFinal=guides/source	changedFiles=1-file	0,09	1,00	2,06	0,04	517,67
dirFinal=guides/source typeDeveloper=external	changedFiles=1-file	0,09	1,00	2,06	0,04	517,67
commitsPull=1-commit dirFinal=guides/source	changedFiles=1-file	0,09	1,00	2,06	0,04	517,15
hasFollowers=true dirFinal=guides/source reviewerFollowsExternalContribu...	changedFiles=1-file	0,08	1,00	2,06	0,04	507,35
commitsPull=1-commit dirFinal=guides/source reviewerFollowsExternalCo...	changedFiles=1-file	0,08	1,00	2,06	0,04	507,35
dirFinal=guides/source typeDeveloper=external reviewerFollowsExternalCo...	changedFiles=1-file	0,08	1,00	2,06	0,04	507,35
commitsPull=1-commit dirFinal=guides/source typeDeveloper=external	changedFiles=1-file	0,08	1,00	2,06	0,04	502,71
hasFollowers=true dirFinal=guides/source typeDeveloper=external	changedFiles=1-file	0,08	1,00	2,06	0,04	501,68
hasFollowers=true commitsPull=1-commit dirFinal=guides/source	changedFiles=1-file	0,08	1,00	2,06	0,04	499,10
commitsPull=1-commit dirFinal=guides/source typeDeveloper=external revi...	changedFiles=1-file	0,08	1,00	2,06	0,04	492,92
hasFollowers=true dirFinal=guides/source typeDeveloper=external reviewer...	changedFiles=1-file	0,08	1,00	2,06	0,04	491,37
hasFollowers=true commitsPull=1-commit dirFinal=guides/source reviewer...	changedFiles=1-file	0,08	1,00	2,06	0,04	489,31
hasFollowers=true commitsPull=1-commit dirFinal=guides/source typeDev...	changedFiles=1-file	0,08	1,00	2,06	0,04	486,73

Fonte: Autoria própria.

5.1.3 Fatores de aceitação

No cenário de descoberta de fatores que influenciam a aceitação dos *pull requests*, a extensão mostrou 7.587 regras. Utilizando as opções de atributos antecedentes, foi possível refinar as regras para a descoberta de fatores específicos de cada atributo da base de dados. A Tabela 1 mostra algumas regras onde foi especificado os atributos que deviam estar presentes no antecedente das regras.

Tabela 1. Regras para aceitação

#	Regras	Sup	Conf	Lift
1	hasFollowers=true -> status=accepted	0,74	0,76	1,01
2	firstPull=true -> status=rejected	0,09	0,42	1,69
3	lifetime=very-lenghty -> status=rejected	0,08	0,51	2,05
4	reviewerPull=spastorino -> status=accepted	0,05	0,92	1,22
5	comments=some-somments -> status=accepted	0,13	0,79	1,06
6	commitsPull=some-commits -> status=rejected	0,04	0,35	1,41
7	changedLines=many-lines -> status=accepted	0,10	0,30	1,20
8	changedFiles=many-files -> status=rejected	0,03	0,30	1,20
9	typeDeveloper=external -> status=accepted	0,72	0,76	1,01
10	reviewerFollowsExternalContributor=true -> status=accepted	0,02	0,92	1,23

Fonte: Autoria própria.

Analisando as regras da Tabela 1, é possível identificar, através das regras com maior *lift*, que os fatores que mais influenciam na aceitação dos *pull requests* do projeto Rails são: o *pull request* ser o primeiro enviado pelo colaborador (regra 2) e o

tempo de vida do *pull request* ser muito longo (regra 3), ambos levando o *pull request* a ser rejeitado.

5.1.4 Fatores do tempo de vida

No cenário de descoberta de fatores que influenciam o tempo de vida dos *pull requests*, a extensão mostrou 7.043 regras. Com a utilização da opção de atributos antecedentes, as regras foram refinadas para descobrir fatores específicos. Na Tabela 2 é mostrado algumas regras que foram encontradas ao definir o atributo antecedente.

Tabela 2. Regras para tempo de vida

#	Regras	Sup	Conf	Lift
1	hasFollowers=true -> lifetime=very-short	0,68	0,69	1,00
2	firstPull=false -> lifetime=very-short	0,56	0,72	1,04
3	reviewerPull=josevalim -> lifetime=very-short	0,10	0,84	1,22
4	status=rejected -> lifetime=very-lengthy	0,08	0,32	2,05
5	comments=some-somments -> lifetime=very-lengthy	0,04	0,28	1,79
6	commitsPull=1-commits -> lifetime=very-short	0,62	0,72	1,04
7	changedLines=many-lines -> lifetime=very-lengthy	0,09	0,26	1,69
8	changedFiles=many-files -> lifetime=very-lengthy	0,03	0,27	1,75
9	typeDeveloper=external -> lifetime=very-short	0,67	0,71	1,02
10	reviewerFollowsExternalContributor=true -> lifetime=very-short	0,02	0,77	1,11

Fonte: Autoria própria.

Analisando as regras da Tabela 2, é possível identificar, através das regras com maior *lift*, que os fatores que possuem maior influencia no tempo de vida dos *pull requests* do projeto Rails são:

1. O *pull request* ser rejeitado (regra 4);
2. O *pull request* possuir muitos comentários (regra 5);
3. O *pull request* modificar muitas linhas (regra 7);
4. O *pull request* modificar muitos arquivos (regra 8).

Todos estes fatores apresentam a tendência de fazer com que os *pull requests* sejam rejeitados.

5.1.5 Fatores do revisor

No cenário de descoberta de fatores que influenciam a atribuição de revisores dos *pull requests*, a extensão mostrou 2.789 regras. Todos os resultados encontrados apresentaram “rafaelfranca” como consequente, o que pode indicar uma insuficiência de dados para produzir bons resultados. Algumas regras encontradas são mostradas na Tabela 3.

Tabela 3. Regras para atribuição de revisor

#	Regras	Sup	Conf	Lift
1	hasFollowers=true -> reviewerPull=rafaelfranca	0,28	0,29	0,99
2	firstPull=false -> reviewerPull=rafaelfranca	0,23	0,33	1,12
3	lifetime=short -> reviewerPull=rafaelfranca	0,08	0,51	2,05
4	status=accepted -> reviewerPull=rafaelfranca	0,22	0,29	1,01
5	comments=some-somments -> reviewerPull=rafaelfranca	0,06	0,37	1,28
6	commitsPull=1-commits -> reviewerPull=rafaelfranca	0,25	0,29	1,01
7	changedLines=many-lines -> reviewerPull=rafaelfranca	0,12	0,34	1,16
8	changedFiles=many-files ->reviewerPull=rafaelfranca	0,04	0,35	1,21
9	typeDeveloper=external ->reviewerPull=rafaelfranca	0,28	0,29	1,00
10	reviewerFollowsExternalContributor=false -> reviewerPull=rafaelfranca	0,29	0,29	1,01

Fonte: Autoria própria.

Dentre as regras da Tabela 3, a regra com o *lift* mais significativo mostra que os *pull requests* do projeto Rails com tempo de vida curto tendem a ser revisados pelo desenvolvedor “rafaelfranca”. As outra regras apresentam *lifts* consideravelmente inferiores, sendo então regras de menor interesse para o pesquisador.

5.2 CONCLUSÃO

Este capítulo mostrou como foi o processo de avaliação da extensão desenvolvida. Foi abordado a base de dados utilizada e como foi o procedimento para a realização dos experimentos de validação das funcionalidades da extensão.

O capítulo a seguir abordará as considerações finais sobre o trabalho e extensão. Também é apresentado as limitações da ferramenta e as recomendações para trabalhos futuros.

6 CONSIDERAÇÕES FINAIS E RECOMENDAÇÕES

Neste capítulo final são apresentadas as considerações finais do trabalho no as recomendações para trabalhos futuros.

6.1 CONSIDERAÇÕES FINAIS

Os esforços que devem ser realizados durante o processo de descoberta de conhecimento sobre *pull requests* são um dos problemas que gerentes de projeto e colaboradores de software devem lidar. Problema esse que pode ser amenizado através da automação de alguns passos deste processo.

Este trabalho buscou solucionar este problema através do desenvolvimento de uma extensão da ferramenta Weka, permitindo que gerentes de projeto e colaboradores de software possam automatizar os processos de pré-processamento e mineração de regras de associação, a partir de bases de dados sobre *pull requests* em repositórios no GitHub, que são extraídas com a ferramenta GHTorrent.

Realizou-se uma revisão da literatura sobre engenharia de software, *pull requests* e mineração de dados. Foi elaborado um documento de requisitos para a extensão proposta e seu desenvolvimento se deu com a aplicação do desenvolvimento orientado a reúso, que incluiu as atividades de analise de

componentes para reúso, adaptação e integração dos componentes. A extensão foi adequada à estrutura de pacotes definida pela ferramenta Weka, para que então fosse validada.

Conclui-se que os objetivos do trabalho foram cumpridos com sucesso. O problema apresentado foi solucionado, uma vez que a ferramenta desenvolvida é capaz de automatizar parte do processo de descoberta dos fatores que influenciam na aceitação, tempo de vida e atribuição de revisores dos *pull requests,* de acordo com o modelo proposto por Soares (2017).

A principal contribuição deste trabalho é auxiliar os gerentes de projeto e contribuidores de software no processo de descoberta de conhecimento. Automatizando parte do processo, através de técnicas elaboradas e validadas por Soares (2017), é possível garantir que os engenheiros e contribuidores descubram conhecimentos válidos e relevantes.

6.2 LIMITAÇÕES

A extensão desenvolvida apresenta limitações na discretização dos atributos *file* e *dirFinal*, que tem como valor uma lista de arquivos e diretórios, respectivamente, separados pelo caractere “|”. A extensão não é capaz de separar estes dados, para que sejam melhores aproveitados, devido a existência de um comportamento estranho no código Java de uma das classes Weka necessárias para a discretização. Não foi possível resolver ou contornar o problema dentro do prazo de desenvolvimento da extensão, o que gerou tais limitações.

Tal limitação não é um grande problema, pois é possível armazenar os dados pré-processados e realizar o tratamento desses atributos através da interface gráfica da ferramenta Weka, onde não ocorre o problema citado.

6.3 RECOMENDAÇÕES

Como recomendações para trabalhos futuros pode-se sugerir que:

a) O processo de extração dos dados sobre os *pull request*, que é realizado com a ferramenta GHTorrent, seja incorporado em uma nova versão desta ferramenta, para facilitar ainda mais o trabalho de descoberta de conhecimento por gerentes de projeto e colaboradores de software;

b) As funcionalidades da extensão sejam portadas para outras ferramentas de mineração de dados, assim como a exploração de novos algoritmos para descoberta de regras de associação;

c) Seja feito uma refatoração do código da extensão, assim como o acréscimo de possíveis melhorias para a interação, de acordo com os aspectos relacionados à interação humano-computador;

d) Acréscimo da capacidade de realizar o tratamento dos atributos *file* e *dirFinal* na extensão PR-Discover.

REFERÊNCIAS

BERRY, Michael J. A.; LINOFF, Gordon S. **Data Mining Techniques**: For Marketing, Sales, and Customer Relationship Management. 2. ed. USA: Wiley, 2004.

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

ESTUBLIER, Jacky. **Software Configuration Management:** A Roadmap. Em: Proceedings of the Conference on The Future of Software Engineering. New York, USA: ACM, p. 279-289, 2000.

FAYYAD, Usama; PIATETSKY-SHAPIRO, Gregory; SMYTH, Padhraic. **From data mining to knowledge discovery in databases.** AI magazine, v. 17, n. 3, p. 37, 1996.

FRANK, Eibe; HALL, Mark A.; WITTEN, Ian H. **The WEKA Workbench.** Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

GOUSIOS, Georgios; PINZGER, Martin; DEURSEN, Arie van. **An Exploratory Study of the Pull-based Software Development Model**. Em: ICSE 2014 Proceedings of the 36th International Conference on Software Engineering, p. 345-355, 2014.

HAN, Jiawei; KAMBER, Micheline; PEI, Jian. **Data Mining:** Concepts and Techniques. 3rd. ed. Rio de Janeiro, Brasil: Elsevier, 2011.

Institute of Electrical and Electronics Engineers. Std 828 - IEEE Standard for Configuration Management in Systems and Software Engineering. New York, 2012.

KOTSIANTIS, Sotiris; KANELLOPOULOS, Dimitris. **Association Rules Mining:** A Recent Overview. Em: GESTS International Transactions on Computer Science and Engineering, v. 32, n. 1, p. 71-82, 2006.

LAPLANTE, Philip A. **What Every Engineer Should Know about Software Engineering.** CRC Press, 2007.

MURTA, Leonardo G. P. **Gerência de Configuração no Desenvolvimento Baseado em Componentes**. Tese (Doutorado) - Universidade Federal do Rio de Janeiro. Rio de Janeiro, 2006.

OTTE, Stefan. **Version control systems.** Computer Systems and Telematics, Institute of Computer Science, Freie Universität, Berlin, Germany, 2009.

PRESSMAN, R. S. **Engenharia de software:** uma abordagem profissional. 7. ed. Porto Alegre: McGraw-Hill, 2011.

RAHMAN, Mohammad Masudur; ROY, Chanchal K. **An insight into the pull requests of github**. Em: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014. p. 364-367.

SAMETINGER, Johannes. **Software engineering with reusable components**. Springer Science & Business Media, 1997.

SILVA, Daniel A. N. da. **WEKAPAR:** Uma Extensão da Ferramenta WEKA para Auxiliar o Pós-Processamento de Regras de Associação. TCC - Universidade Federal do Acre. Rio Branco, 2016.

SILVA, Edna L. da; MENEZES, Estera M. **Metodologia da Pesquisa e Elaboração de Dissertação**. 4. ed. Florianópolis: UFSC, 2005.

SOARES, Daricélio M. **On the Nature of Pull Requests:** A Study About this Collaboration Paradigm over Open-Source Projects Using Association Rules. Tese (Doutorado) - Universidade Federal Fluminense. Niterói, 2017.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

TSAY, Jason; DABBISH, Laura; HERBSLEB, James. **Influence of social and technical factors for evaluating contribution in GitHub**. Em: Proceedings of the 36th international conference on Software engineering. ACM, 2014. p. 356-366.

APÊNDICE

APÊNDICE A – DOCUMENTO DE REQUISITOS

Documento de Requisitos de Software

EXTENSÃO WEKA - PR-DISCOVER

Versão 1.0

Desenvolvedores/Analistas

José Marcos Lopes Damasceno

Rio Branco – AC

2018

Histórico de Alterações

Data	Versão	Descrição	Autor
28/02/2018	0.1	Versão inicial do documento.	José Marcos
17/03/2018	1.0	Versão final do documento.	José Marcos

1. Análise do Problema

O processo de descoberta de conhecimento sobre *pull requests* em bases de dados de um repositório específico no GitHub envolve muitos passos que exigem que o pesquisador possua noções sobre *data mining* e também sobre a ferramenta Weka, que é utilizada durante o processo. A extensão a ser desenvolvida busca tornar o processo mais simples e acessível para gerentes que não tenham afinidade com o processo de mineração ou desejem automatizar parte do processo.

2. Necessidades Básicas do Cliente

Simplicidade no pré-processamento dos dados que serão utilizados, visualização das regras descobertas e instalação através do *package manager* Weka.

3. Estudo de Viabilidade

Foi realizado um estudo para se verificar a viabilidade do desenvolvimento da extensão em questão.

3.1. Viabilidade Técnica

A extensão é tecnicamente viável, pois o desenvolvedor já possui experiência com a linguagem Java, que é utilizada pelo software Weka.

3.2. Viabilidade Econômica

É economicamente viável porque todos os softwares utilizados durante o desenvolvimento são livres para o uso.

3.3. Viabilidade Legal

A extensão é legalmente viável pois não viola nenhum contrato ou lei vigente.

4. Missão do Software

Facilitar o processo de descoberta de conhecimento em um repositório no GitHub, para que gerentes de desenvolvimento e pessoas interessadas tenham acesso a informações úteis sobre o a utilização de *pull requests* em seus repositórios.

5. Limites do Sistema

ID	Funcionalidade	Justificativa
L1	Coleta de dados	A extensão não será capaz de realizar a coleta dos dados, sendo o usuário o responsável por fornecer os dados sobre o repositório.

6. Benefícios Gerais

ID	Benefício
B1	Pré-processamento automático.
B2	Descoberta das regras de associação automaticamente.
B3	Não há necessidade de conhecimento profundo sobre mineração de dados.
B4	
B5	

7. Restrições

ID	Restrição	Descrição
R1	Weka package manager	A extensão deve ser compatível com o package manager.
R2		

8. Atores

ID	Atores	Descrição
A1	Usuário	Instala a extensão e faz a utilização de suas funcionalidades.

9. Requisitos Funcionais

ID	Funcionalidade	Necessidades	Prioridade
RF01	Carregar base de dados	A extensão deve carregar a base de dados que foi coletada.	Alta
RF02	Pré-proc. automático	A extensão deve realizar o tratamento dos dados de maneira automática	Alta
RF03	Extração automática de	A extensão deve fazer a mineração	Alta

ID	Funcionalidade	Necessidades	Prioridade
	regras de associação	das regras automaticamente	
RF04	Visualizar regras	A extensão deve mostrar as regras para o usuário	Alta
RF05	Diferentes visualizações	A extensão deve mostrar as regras sobre diferentes perspectivas	Média
RF06	Perspectiva “aceitação”	A extensão deve mostrar regras específicas sobre a aceitação dos *pull requests*	Alta
RF07	Perspectiva “revisor”	A extensão deve mostrar regras específicas sobre a atribuição de revisores dos *pull requests*	Média
RF08	Perspectiva “lifetime”	A extensão deve mostrar regras específicas sobre o tempo de vida dos *pull requests*	Média
RF09	Instalação pela Weka	A extensão deve ser instalado pelo package manager da Weka	Alta
RF10	Exportar resultados	A extensão deve exportar os resultados da mineração	Baixa
RF11	Salvar resultados	A extensão deve salvar os resultados da mineração para que possam ser abertos novamente	Baixa
RF12	Abrir resultados salvos	A extensão deve abrir abrir os resultados salvos de minerações anteriores	Baixa
RF13	Salvar dados pré-processados	A extensão deve permitir que os dados que foram pré-processados sejam salvos em um arquivo	Média
RF14	Configurar visualização por atributos das regras	A extensão deve permitir configurar uma visualização personalizada com base em atributos no antecedente das regras	Baixa
RF15	Configurar visualização por métricas das regras	A extensão deve permitir configurar uma visualização personalizada com base em métricas das regras	Baixa

10. Requisitos Não-Funcionais

ID	Requisitos	Categoria
NRF1	Facilidade de uso	Usabilidade
NRF2	Instalação simples	Usabilidade
NRF3		

11. Requisitos de Hardware

11.1. Configuração Mínima

- Máquina com Java 8 instalado;
- Weka 3.8.0.

12. Ferramentas de Desenvolvimento e Licença de Uso

a. Eclipse Oxygen.2 (4.7.2) - Eclipse Public License - v2.0;
b. Weka 3 (3.8.0) - GNU General Public License - v3.0.