Desenvolvimento de uma ferramenta para coleta e tratamento de dados do GitHub com aplicação de algoritmos de aprendizado de máquina para classificação

Max N. O. Lima[1], **Guilherme A. Avelino**[1]

[1]Departamento de Computação – Universidade Federal do Piauí (UFPI)
Teresina – PI – Brasil

max.lima2@gmail.com, gaa@ufpi.edu.br

Abstract. *Software has become increasingly present in the various activities of our daily lives. To meet this growing demand, it is necessary to constantly improve software development processes and tools. In this context, data on the development process, available in public software repositories, can be used as a basis for conducting research in software engineering. However, obtaining and processing this data is a challenge. Thus, this work aims to create a tool to assist in the collection and processing of data from GitHub. The tool presented here was used to build a research dataset composed of development data from 1,162 software repositories, selected from the most popular ones on GitHub.*

Resumo. *Software vem se tornando cada vez mais presente nas diversas atividades do nosso cotidiano. Para atender essa demanda crescente, faz-se necessário o aprimoramento constante dos processos e ferramentas de desenvolvimento de software. Nesse contexto, dados sobre o processo de desenvolvimento, disponíveis em repositórios públicos de software, podem ser utilizados como base para a realização de pesquisas em engenharia de software. Entretanto, obter e tratar esses dados é um desafio. Dessa forma, o presente trabalho tem como objetivo criar uma ferramenta para auxiliar na coleta e tratamento de dados oriundos do GitHub. A ferramenta aqui apresentada foi utilizada para construir um dataset de pesquisa composto de dados de desenvolvimento de 1.162 repositórios de software, selecionados dentre os mais populares no GitHub.*

1. Introdução

O desenvolvimento de software, ao longo do tempo se tornou cada vez mais uma atividade complexa requisitando assim o trabalho em conjunto de uma equipe de desenvolvimento [9], sendo os membros desta, em geral, dispersos fisicamente [7]. Diante disso, ferramentas para auxiliar o desenvolvimento em conjunto se fazem necessárias, servindo para comunicação e coordenação de atividades além de interação com o código-fonte do software. Essas ferramentas podem ser categorizadas em ferramentas de comunicação, como e-mails, controle de versão, tais como Git[1], CVS[2] e SVN[3], gerenciamento de atividades, como o *Trello*[4], e de desenvolvimento, como as IDEs utilizadas pelos desenvolvedores.

[1]https://git-scm.com/
[2]https://www.nongnu.org/cvs/
[3]https://subversion.apache.org/
[4]https://trello.com/

Todas estas ferramentas armazenam dados sobre o seu uso, os quais podem ser relevantes para entender o processo de desenvolvimento de software e facilitar a identificação de problemas nesse processo, podendo assim ajudar em tomadas de decisão pelos gerentes de projeto [6].

Durante muito tempo, um dos grande empecilhos para estudos nessa área se dá pelo âmbito competitivo deste setor, pois as empresas acabam por não dar visibilidade pública para o código-fonte de suas aplicações, por razões de competitividade, impedindo assim o acesso a dados de desenvolvimento e consequentemente a análise desses. Entretanto, com a evolução e informatização do mundo cada vez maior, o desenvolvimento de software passou a ter bastante relevância e por conseguinte, a demanda de ferramentas para automação e aprimoramento na realização de tarefas aumentou de forma proporcional. Isso promoveu um grande interesse por parte da comunidade de desenvolvedores de contribuir para o chamado *Open Source* e consequentemente para o aumento da quantidade de repositórios de software que podem ser acessados facilmente por qualquer usuário. Portanto, a análise e estudo na área de engenharia de software tornou-se mais fácil graças a essa nova grande fonte de dados.

Nesse contexto, estudos empíricos com foco em identificar problemas no processo de desenvolvimento de software e promover pontos de melhoria têm surgido, para assim auxiliar os gestores de projetos. É importante ainda mencionar que uma melhoria no desenvolvimento de software contribui também para os consumidores de aplicações visto que um desenvolvimento com melhor qualidade acarreta em um melhor resultado final, ou seja, aplicações mais bem trabalhadas. Porém, existem ainda problemas e desafios com relação à coleta e tratamento dos dados oriundos de repositórios de software como mostrados por Kalliamvakou et al. [8], que acabam por atrasar esses estudos, visto que esse processo, sendo a etapa inicial em uma pesquisa na área de engenharia de software, acaba por demandar bastante tempo desnecessariamente. Portanto, este trabalho visa criar uma ferramenta para auxiliar na construção de bases de dados, diminuindo o tempo para obtenção destes dados, que possa ser utilizada para pesquisas de forma a facilitar a mineração de dados em repositórios de software, permitindo que pesquisadores foquem a maior parte de seu tempo realmente na pesquisa ao invés da construção da base de dados, e com isso, contribuir positivamente para o processo desenvolvimento de software.

2. Referencial teórico

2.1. Repositório de software

A definição de repositórios de software poder ser dada como sendo um local para armazenamento de recursos que podem ser extraídos por usuários a qualquer instante de forma que tem como principal propósito fornecer acesso ao código de software em um âmbito colaborativo [14]. Atualmente, um dos maiores e mais famosos repositórios de software é o GitHub, que é a fonte de dados utilizada pelo presente trabalho e que atualmente conta com um total de mais de 56 milhões de desenvolvedores e mais de 100 milhões de repositórios [4].

Repositórios de software são geralmente compostos por um sistema de controle de versões, Git no caso do GitHub, além de ferramentas diversas de interação entre os usuários e controle de atividades. Essas informações são todas armazenadas e podem ser analisadas para promover um melhor entendimento sobre o desenvolvimento de software.

É importante enfatizar que repositórios de software, como dito por Hassan [6] são comumente usados na prática para armazenamento de dados e raramente são usados para apoiar processos de tomada de decisão.

2.2. Sistemas de controle de versão

Os sistemas de controle de versão, de forma intuitiva, remetem a ferramentas que possibilitam gerenciar as versões baseadas nas versões geradas do software. Dessa forma, o histórico de alterações em um projeto de software é armazenado, podendo ser consultado a qualquer momento possibilitando assim uma maior facilidade no desenvolvimento de software quando se tem uma equipe visto que as alterações são facilmente organizadas e atribuídas aos seus devidos autores. São exemplos de sistemas de controle de versão o CVS, SVN e Git, sendo este último utilizada pelo GitHub. Uma definição que pode auxiliar bastante no entendimento de sistemas de controle de versão é dada por Pavim [12]: "Um sistema de controle de versão é comparado a uma 'máquina do tempo' por alguns usuários, pois é sempre possível retornar ao passado de arquivos e diretórios armazenados.".

2.3. Mineração de repositórios de software

Tendo uma breve fundamentação sobre repositórios de software, a mineração de repositórios de software trata da extração de dados encontrados nos repositórios para que assim possam-se obter informações relevantes sobre projetos de software [6]. Em outras palavras, através da aplicação de técnicas de mineração de dados, objetiva-se transformar tais repositórios em fontes ativas de informação sobre o processo de desenvolvimento capazes de auxiliar na melhoria contínua desse processo. Esses dados, uma vez obtidos, servem como base para pesquisas de forma que métricas podem ser geradas e pontos importantes possam ser levantados sobre um determinado repositório ou um conjunto de repositórios. Podemos citar como exemplo de aplicação prática de mineração de repositórios de software o trabalho de Weiss et al. [15] que propõe uma forma de predição do tempo que um desenvolvedor pode levar para resolver uma *issue* através da similaridade na descrição da mesma com alguma outra *issue* que já se tem conhecimento do tempo de resolução.

2.4. Ferramentas

Além dos desafios existentes para realizar a implementação do trabalho, é necessário elencar mecanismos de obtenção dos dados de repositórios. Para isso, Mombach e Valente [10] em seu trabalho mostram 3 tipos de mecanismos que podem ser utilizados para coleta de dados do GitHub, são eles GitHub REST API[5], GHArquive[6] e GHTorrent[7]. No trabalho foram realizadas 3 buscas usando cada um dos sistemas de forma a fazer algum tipo de comparação. A partir desses testes foi possível identificar que o GHArchive, apesar de possuir os dados de forma estruturada, não consegue extrair todos os dados possíveis (como os tipos de linguagens presentes em um repositório). Além disso a complexidade para realizar as buscas é bastante elevada pelo fato de ter que relacionar várias tabelas para conseguir o resultado pretendido e ainda se faz necessário o uso de expressões regulares. A complexidade elevada também pode ser dita sobre o GHTorrent já que

[5]https://docs.github.com/en/free-pro-team@latest/rest
[6]https://www.gharchive.org/
[7]https://ghtorrent.org/

o mesmo apresenta os dados organizados também em tabelas e requisita o mesmo tipo de procedimento para obtenção dos dados no sentido de relacionar várias de suas tabelas. Já na REST API do GitHub, essa mesma complexidade não é constatada já que a sintaxe da requisição para executar as buscas é bem intuitiva e simples. Entretanto, é importante dizer que o uso da REST API apresenta a desvantagem de um limite de requisições em uma determinada faixa de tempo, o que não acontece com os outros dois mecanismos anteriores. Ainda assim, a API apresenta outra vantagem, que diz respeito à atualização dos dados e também completude dos dados, já que através da API é possível obter os dados atualizados em tempo real e desde o início do uso do GitHub. Já os outros dois apresentam dados apenas a partir de 2011 e ainda, não apresentam dados atualizados em tempo real, apenas de hora em hora, no caso do GHArchive e de mês em mês no caso do GHTorrent.

Adicionalmente, a API apresenta um pequeno processamento de dados se comparado com os outros dois sistemas, que, devido a quantidade de tabelas acaba por executar um alto processamento de dados. Diante disso, a API disponibilizada pelo GitHub, para o presente autor, mostra-se bem mais vantajosa em relação às duas outras técnicas citadas anteriormente, principalmente pelo nível de complexidade bem reduzido promovendo uma implementação bem mais aproveitável e satisfatória e tendo apenas o limite de requisições como obstáculo maior. Todavia, esse limite de requisições ainda é um problema e por isso, é possível que a utilização da versão GraphQL[8] ao invés da REST possa contornar esse obstáculo, pois de acordo com a documentação oficial da API do GitHub, "[...] com o GraphQL, uma chamada pode substituir várias chamadas REST. [...]" [3]. Isso se dá pois com o GraphQL várias requisições podem ser concatenadas em uma só, ou seja, além de diminuir o número de requisições, os dados que serão retornados serão bem mais específicos e filtrados.

No trabalho de Brito, Mombach e Valente [2], são mostradas duas tabelas que exibem resultados de diferentes casos utilizando REST e GraphQL. Com GraphQL, o número de requisições foi igual ou inferior em relação à REST e, além disso, o número de dados retornados usando GraphQL era bem inferior aos casos utilizando REST, mostrando assim, que a utilização do GraphQL pode trazer uma razoável solução para o problema do limite de requisições do GitHub além de trazer uma menor complexidade no tratamento de dados com a recuperação de dados de forma bem mais limpa e precisa.

Também existe uma ferramenta, denominada Perceval [9], desenvolvido pela GrimoireLab[10], que foi utilizada e facilitou em muito uma parte da implementação que trata da recuperação de dados de *commits*, visto que o propósito da ferramenta diz respeito à recuperação de dados de repositórios e esta utiliza-se da abordagem de analise de logs do arquivo local do Git, e por sua vez, acaba por ter uma melhor performance em relação a fazer requisições ao servidor, pois, geralmente, a quantidade de *commits* em relação aos outros dados é bem mais abundante, podendo provocar atrasos desnecessários e facilmente evitáveis.

[8] https://docs.github.com/en/free-pro-team@latest/graphql

[9] https://github.com/chaoss/grimoirelab-perceval

[10] https://chaoss.community/

3. Trabalhos relacionados

3.1. Desafios

Primeiramente é importante destacar os desafios que existem no âmbito de mineração de repositórios de software. Nesse contexto, um dos trabalhos mais importantes é o desenvolvido por Kalliamvakou et al. [8] que elenca desafios enfrentados ao minerar dados do GitHub. Dentre os desafios mostrados em seu trabalho, apenas os mais importantes para a execução deste serão elencados. Tratam-se dos problemas numerados como IV e V que dizem respeito à repositórios que não são necessariamente repositórios de software e que dois terços dos projetos em repositórios de software tratam-se de projetos pessoais, respectivamente. Esses pontos podem interferir bastante com relação aos repositórios escolhidos para realizar estudos, portanto, faz-se necessária uma estratégia para que esses projetos sejam identificados. Analisar esses repositórios e classificá-los não é uma atividade trivial. Entretanto, os autores nos dão sugestões que auxiliam a realizar essa atividade. Eles sugerem investigar dados tais como o número de *commits* (um dado que diz respeito a um registro de alterações nos arquivos do repositório), número de usuários que realizam *commits* e cadastram *issues* (dado que serve geralmente para que usuários registrem problemas em um código no repositório para ser resolvido), utilizando tais números como indício de que o repositório é ou não de uso pessoal.

Ainda com relação aos desafios, na área de mineração de dados, Justen [16], em uma apresentação feita na conferencia PGConf Brasil[11], mostra alguns problemas que podem ser analisados e verificados como relevantes para a aplicação deste trabalho e que serão mencionados a seguir. Os desafios citados por ele foram: formato, dispersão, quantidade e domínio da área. No que diz respeito ao formato dos dados, os dados que serão extraídos dos repositórios serão unicamente retirados no formato de texto padrão e assim estruturados para melhor organização. Dados como *commits*, *issues*, comentários etc. Já com relação a dispersão, é relevante a preocupação já que dependendo da fonte dos dados de onde os mesmos serão extraídos, os mesmos podem estar estruturados de formas distintas ocasionando dificuldades na implementação das ferramentas para retirada desses dados de forma igualitária, independente da fonte. Quanto ao desafio da quantidade de dados, o autor mostra que muitas vezes o usuário não consegue acessar os dados por utilizar ferramentas limitadas para fazê-lo. Um exemplo dado na própria apresentação diz respeito à tentar acessar um arquivo csv muito grande com um software como Excel, Planilhas do Google etc. O que não aconteceria já que estes softwares não conseguem abrir arquivos tão grandes. Com relação ao desafio do domínio, ele acontece por que o trabalho de Justen tem o objetivo de disponibilizar os dados que ele se propõe a trabalhar, para qualquer tipo de usuário, sendo este leigo ou não. Tal desafio, não se aplica para este trabalho já que o público alvo são usuários capacitados que querem trabalhar com pesquisas na área de ciência da computação e que por isso possuem conhecimento sobre os dados. Justen, além de mostrar desafios no que diz respeito a mineração de dados também levanta a questão da utilização do sistema gerenciador de banco de dados objeto relacional PostgreSQL que, segundo ele, possui uma boa integração com a linguagem Python, que será utilizada nas implementações deste trabalho.

[11] https://www.pgconf.com.br/2019/

4. Metodologia

Para execução do trabalho, foi seguinda uma metodologia conforme represetada na Figura 1.

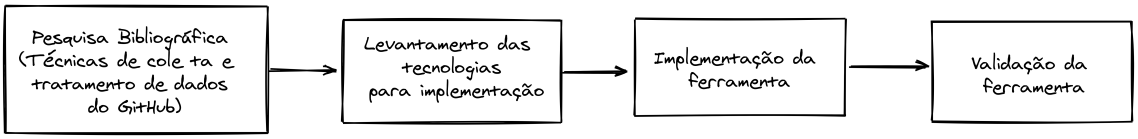


Figura 1. Diagrama de metodologia

Inicialmente foi realizada uma busca na literatura por conteúdos relacionados ao tema, para que pudessem ser revisadas as diversas técnicas de coleta e tratamento de dados oriundos de repositórios de desenvolvimento de software já utilizadas por outros pesquisadores. Essa etapa inicial teve como objetivo compreender melhor o estado da arte, identificando as técnicas mais promissoras e suas limitações de forma a dar subsídios para o desenvolvimento de uma nova solução. Tendo levantando essas informações, foram escolhidas ferramentas e tecnologias que possibilitariam o desenvolvimento do trabalho resultando assim no início do desenvolvimento da ferramenta propriamente dita, que é responsável por capturar dados de repositórios e tratá-los para então salvá-los em uma base da dados.

Adicionalmente à coleta, tratamento e persistência dos dados, foram desenvolvidas funcionalidades que lidam diretamente com os problemas elencados na Seção 3.1. Tais funcionalidades lidam com a classificação de repositórios e o mapeamento de usuários com identificadores diferentes que podem vir a estarem relacionados ao mesmo desenvolvedor.

Por fim, a última etapa se deu pela validação da ferramenta desenvolvida, bem como das funcionalidades adicionais de tratamento dos dados. Nas seções a seguir são descritas as tecnologias escolhidas para criação da ferramenta bem como o funcionamento da mesma.

4.1. Tecnologias utilizadas

No presente trabalho, para implementação da ferramenta como um todo, foi utilizado a linguagem Python que, além de ser uma linguagem amigável e de fácil entendimento. Além disso é uma linguagem famosa para análise de dados utilizada por cientistas e é utilizada para isso devido aos seus vários pacotes que auxiliam em implementações desse tipo [13]. Além disso foi utilizado Django[12], um *framework* Python de rápido e fácil desenvolvimento, que possibilitou a disponibilização para uso da ferramenta no formato de API.

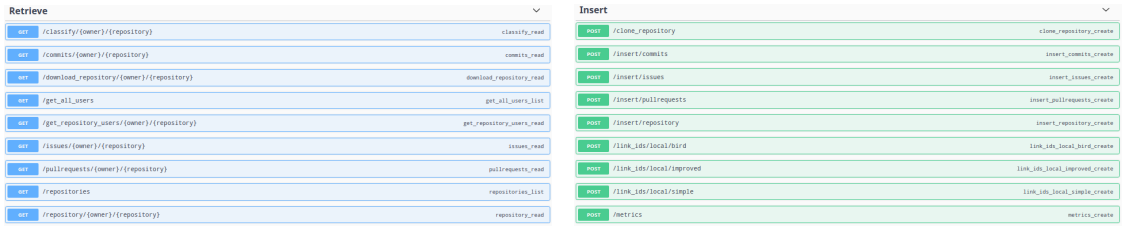
Para captura dos dados dos repositórios, utiliza-se uma mescla de requisições à API do GitHub na sua versão em GraphQL e da ferramenta Perceval, para captura de *commits* como já citado anteriormente na Seção 2.4, e ainda auxiliar na definição de quais dados, em relação à cada categoria de dados (*issues*, *commits* e *pull requests*, uma requisição para enviar uma sugestão de melhoria para o repositório), seriam capturados

[12]https://www.djangoproject.com/

e armazenados no *database* já que a ferramenta possui isso bem definido. Por fim, foi utilizado o PostgreSQL para armazenar todos os dados obtidos, tratados e gerados.

5. Ferramenta

Nesta Seção apresentamos a ferramenta implementada. Esta foi feita no formato de API como já citado na Seção 4.1 e todos os *endpoints* que são disponibilizados pela ferramenta encontram-se na própria página onde pode ser diretamente testado, sem a necessidade de uma aplicação cliente. Na Figura 2(a) podemos ter uma visão de todos os endpoints para obtenção dos dados armazenados pela API e na Figura 2(b) todos os que se referem a inserção de dados na ferramenta.



(a) Obtenção de dados (b) Inserção de dados

Figura 2. Endpoints da API

5.1. Implementação

5.1.1. Arquitetura

Para a implementação da ferramenta foi utilizado o padrão REST provido pela própria *framework* Django possuindo 4 componentes principais, tais como *Model*, responsável por abstrair os dados e relacionamentos do banco de dados em classes, *View*, que faz a comunicação entre o *Model* e *template*, ou seja, é nesta onde são definidas o que cada rota da API pode receber como parâmetro, quais a regras de lógica serão executadas para aquele comando e qual o retorno para o usuário, *Serializer* que permite que dados complexos sejam convertidos em tipos de dados nativos do *python* e *URL*, responsável por armazenar as informações de roteamento da API.

5.1.2. Utilização

Para utilização da ferramenta, visto que esta é implementada no formato de API, se faz necessária apenas fazer requisições para os determinados *endpoints*, dependendo da necessidade do usuário. Para inserção dos dados é necessário, obrigatoriamente, seguir uma ordem sendo ela, a clonagem do repositório, recuperação dos dados de repositório e só então as recuperações dos outros dados. Além disso, para que seja possível a recuperação de métricas, que será detalhada posteriormente, também se faz necessário que a clonagem e a recuperação dos outros dados seja feita. Isto também pode ser dito para a questão do mapeamento de usuários, visto que os dados de identificação só são obtidos após recuperar os dados de commits. Por fim é importante destacar que, para a maioria dessas requisições, a referência para o repositório (nome do dono e o nome do repositório) se faz obrigatória, independente do método da requisição (*GET* ou *POST*).

5.2. Funcionalidade

Como pode ser observada na Figura 3, a ferramenta é composta por dois módulos principais, módulo de Coleta de Dados e módulo de Tratamento de dados, os quais são detalhados, respectivamente, nas Seções 5.2.1 e 5.2.2.

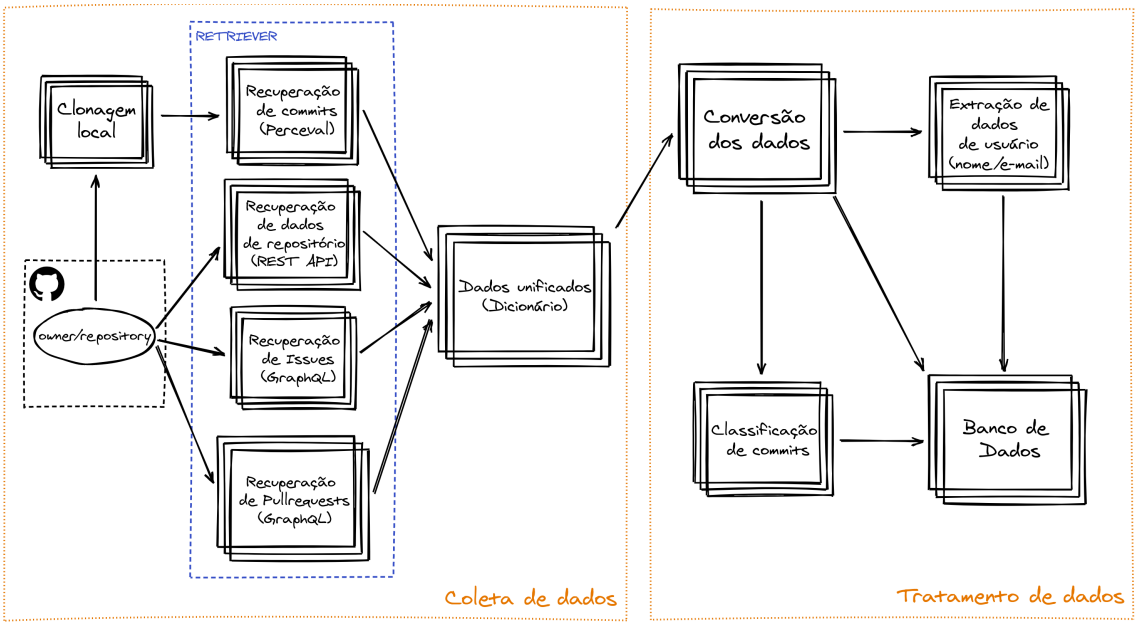


Figura 3. Diagrama de coleta e tratamento de dados

5.2.1. Coleta de dados

Após a fase de revisão de literatura, foi desenvolvida uma ferramenta para coletar os dados oriundos do GitHub. Os dados a serem coletados foram selecionados baseando-se em discussões entre membros desse projeto de forma a escolher os dados que são mais relevantes para pesquisas em Engenharia de Software. Essa ferramenta captura 4 categorias de dados, sendo elas: Repositórios, *Commits*, *Issues* e *Pullrequests*. A coleta de dados segue o fluxo definido na Figura 3 e , em níveis de implementação, toda a coleta de dados é gerenciada por uma classe chamada *Retriever*, que termina abruptamente o processo de coleta ao capturar algum erro em alguma das etapas etapas, retornando o erro para o usuário, ou envia os dados para serem tratados caso tudo ocorra com sucesso.

Passo 1: Preparação para coleta de dados. Primeiro, indica-se o repositório com nome do autor e nome do repositório propriamente dito. Com a referência para o repositório alvo válido, ou seja, dentro do conjunto de repositórios do GitHub, o primeiro passo é a clonagem do repositório no servidor que hospeda a ferramenta. Desta forma, os arquivos resultantes da clonagem poderão ser usados para a extração de *commits* e de métricas para classificação de repositórios, como será descrito na Seção 5.2.3.1.

Passo 2: Coleta de dados de repositório. Depois da clonagem, as informações detalhadas do repositório serão obtidos através da API versão REST do GitHub, pois alguns dos dados necessários, e que não poderiam ser descartados, só poderiam ser obtidos através do recurso em questão. Além disto, como os dados de repositórios são de pequena quan-

tidade, não existe problema em utilizar este meio para o fazer. Este passo é importante pois se algum problema ocorrer ao capturar os dados de repositórios o processo de coleta será abortado e o respectivo erro será retornado para o usuário.

Passo 3: Coleta de dados de *commits*. Com a coleta de dados de repositórios bem como sua clonagem feitas, é o momento da recuperação de *commits* através da ferramenta Perceval, que utiliza os registros de alterações salvos em arquivos de *log* pelo Git. Esses arquivos estão presentes no repositório que foi clonado no primeiro passo e por isso se torna imprescindível que essa clonagem seja realizada com antecedência.

Passo 4: Coleta de dados de *issues* e *pullrequests*. Agora, para *issues* e *pullrequests* temos o mesmo passo com a única diferença de quais os dados que são recuperados de cada uma e, para isso, são geradas *strings* de busca para a API do GitHub na sua versão GraphQL.

Passo 5: Unificação de dados. Com todas essas informações recuperadas, o *Retriever* monta um dicionário de forma a organizar todos esses dados para que, na fase de tratamento, esses dados possam ser facilmente diferenciados.

5.2.2. Tratamento de dados

Após a coleta dos dados, é feito o tratamento que tornará possível a inserção desses dados no banco de dados. Detalhes do esquema dessa base de dados são representados na Figura 4.

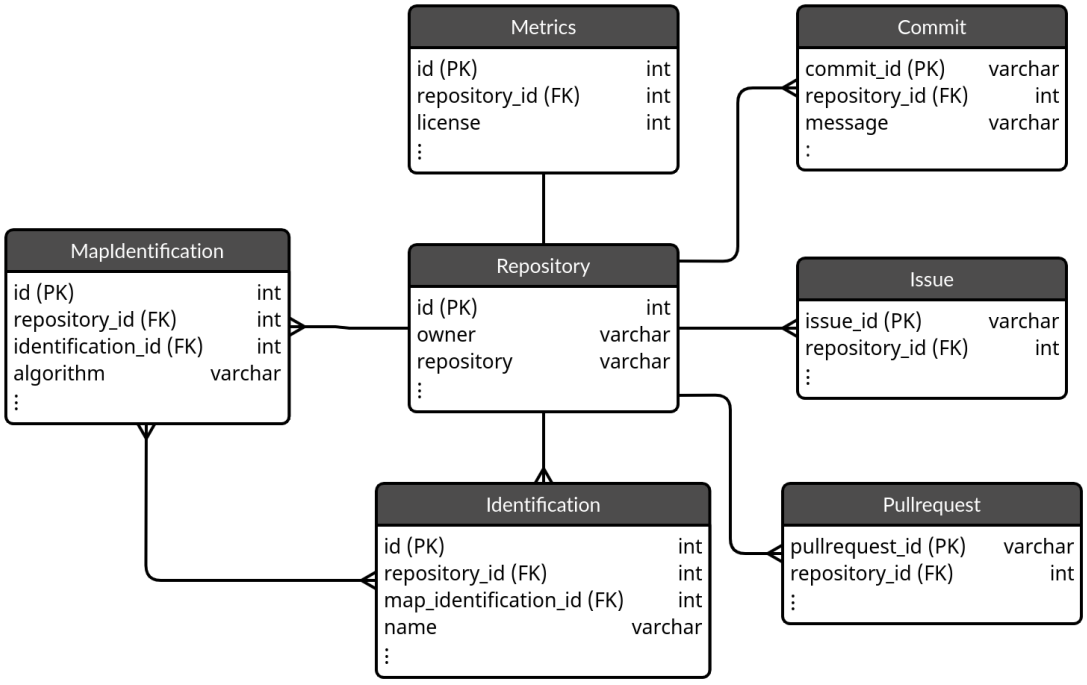


Figura 4. Esquema do banco de dados

A Figura 4 apresenta um esquema simplificado da base de dados, objetivando

apresentar as tabelas e relacionamento entre elas, ou seja, dados sobre cada categorias de dados estão sendo omitidos, logo, como exemplo temos para *commits*, informações do autor do mesmo, arquivos que este modifica etc. O mesmo pode ser dito sobre as outras classes de dados (*issues*, *pullrequests* etc). Em linhas gerais a entidade central é a tabela de repositórios (*Repository*) que armazenam os dados que caracterizam um repositório de software, tais como linguagem de programação utilizada, proprietário do repositório, descrição, dentre outros. Todas as demais entidades se relacionam com a tabela *Repository*. Além disso, temos as tabelas que armazenam as principais informações, sendo elas a de *commits*, *issues* e *pullrequests* e ainda, mais três tabelas que serão descritas a seguir.

A tabela *Identification* é utilizada para armazenar informações de nome e e-mail dos usuários de um repositório e a tabela *MapIdentification* é responsável por armazenar informação de mapeamento dos usuários já armazenados do repositório. A tabela *Metrics* é responsável por armazenar as métricas calculadas de um repositório para posteriormente ser feita a classificação do mesmo, classificação está que será explicada posteriormente como uma funcionalidade adicional na Seção 5.2.3.1.

Adicionalmente à inserção dos dados no banco existem passos extra. Para os *commits*, há um processo de classificação destes com relação ao seu objetivo utilizando uma abordagem proposta por Amit e Feitelson [1], que analisa a mensagem dos *commits*, classificando-os em *corrective*, *refactor*, *perfective* e *adaptative*. A classificação é feita utilizando expressões regulares para analisar as mensagens de *commits* e realizar as devidas relações com cada categoria. Esses dados de classificação são disponibilizados para cada *commit*, individualmente, no banco de dados, de forma a permitir que o usuário possa usar essa informação em seus estudos. Amit e Feitelson propõe que a comparação entre as proporções de *bugFixes* antes e depois de *commits* de refatoração podem indicar a importância de se fazerem refatorações em projetos de software. Este é um exemplo prático do uso da classificação automática do propósito dos *commits* em pesquisas de Engenharia de Software.

Além disso também existe a etapa de extração de dados de usuário para o mapeamento desses usuários, uma vez que, geralmente, quando um usuário faz um *commit*, este possuirá dados de usuário referentes ao que foi configurado localmente e este identificador pode ser independente do identificador da conta no servidor remoto e por isso, um problema acaba por ser gerado, visto que usuários com identificadores (nome/e-mail) diferentes podem remeter à mesma pessoa, o que acaba por interferir negativamente no cálculo de métricas em repositórios em que isso é recorrente. Na próxima seção, mais detalhes dessa funcionalidade serão mostrados.

5.2.3. Funcionalidades adicionais

5.2.3.1 Classificação de repositórios

Kalliamvakou et al. [8], como já citado anteriormente, apresentou alguns perigos relacionados a minerar dados extraídos do GitHub. Dentre esses perigos, é mencionada a existência de muitos repositórios que não são efetivamente projetos de desenvolvimento de software. Dessa forma, para lidar com esse perigo, é importante a classificação dos repositórios de forma que seja possível identificar repositórios aptos a servir como fonte de

estudo, ou seja, repositórios que contém um projeto de desenvolvimento software. Como solução para isto, foi implementado um classificador de repositórios seguindo a abordagem proposta por Muniah et al. [11]. Para realizar a classificação dos repositórios são necessários dois passos: a captura de métricas dos repositórios e a efetiva classificação, a partir das métricas capturadas, utilizando um classificador. Essas métricas, umas vez capturadas, são persistidas no banco de dados evitando o custo de computações futuras. Muniah et al., em seu trabalho, utiliza duas estratégias diferentes para fazer essa classificação: *Random Forest*, que consiste em um classificador que utiliza árvores de decisão e é de fácil utilização, e *Score Based Algorithm*, algoritmo criado pelos autores do trabalho citado e que consiste em classificar um repositório como válido a partir de uma função que compara se uma métrica dada é maior ou não que o seu limiar, retornando 1 em caso afirmativo ou 0 caso contrário, e multiplicando esse resultado pelo seu respectivo peso. Esses limiares foram obtidos por eles através de treinamentos com repositórios previamente rotulados. No trabalho é mostrado como pode-se obter esses valores a fim de conseguir classificar os repositórios definidos. Para cálculo das métricas temos as seguintes dimensões:

- Comunidade: obtido através do cálculo de contribuidores responsáveis por pelo menos 80% do código fonte
- Integração contínua (CI), como evidência de qualidade: obtido através da localização de arquivos de configuração de alguma das ferramentas para integração contínua.
- Documentação, como evidência de mantenabilidade: obtido através do cálculo do número de comentários em relação ao número de linhas de código.
- Histórico, como evidência de evolução sustentada: obtido através do cálculo da média do número de *commits* por mês.
- Issues, como evidência de gerenciamento de projeto: obtido através do cálculo da média do número de *issues* por mês.
- Licença, como evidência de responsabilidade: checando se o projeto possui ou não licenciamento.
- Testes unitários, também como evidência de qualidade: obtido através do cálculo do número de linhas de teste em relação ao número de linhas de código fonte.

Existe ainda a dimensão relacionada à arquitetura, entretanto esta foi descartada visto que existe uma grande variedade de arquiteturas de software tornando assim difícil a relação entre vários repositórios diferentes que podem conter todas ou a maioria dessa variedade.

No trabalho de Muniah et al. existe a separação entre repositórios de software organizacionais, que, em síntese, são repositórios que contém projetos semelhantes aos projetos mantidos por grandes organizações como *facebook*, *microsoft* dentre outros e os utilitários, que são mais voltados para auxiliar os próprios desenvolvedores ou para outros propósitos gerais, como *plugins* para navegadores ou *packages* para alguma linguagem de programação etc. Para este trabalho, foi considerado que, se um repositório for classificado como válido para qualquer um dos tipos de repositórios - organizacional ou utilitário - então este é válido como um repositório que contém um projeto de software. Além disso, como temos dois algoritmos, ambos são executados e se ao menos um dos resultados forem positivos então o resultado final será que aquele repositório em análise é válido como um repositório que contém um projeto de desenvolvimento de software.

5.2.3.2 Mapeamento de identificadores

Com relação ao mapeamento de identificadores de usuários, Kalliamvakou et al. [8] nos mostra que uma das promessas se trata de que a interligação entre os desenvolvedores participantes no projeto de desenvolvimento pode nos dar uma ampla visão das atividades de desenvolvimento de software. Essa promessa se refere principalmente a entender o fluxo de atividades e interações que acontece no âmbito do processo de desenvolvimento e, para isso, identificar os usuários que remetem à mesma pessoa se torna crucial. Portanto, além dos motivos mencionados na seção 3, o mapeamento de identificadores se torna de grande relevância para este trabalho.

Goeminne e Mens [5] solucionam esse problema utilizando 3 algoritmos diferentes. Todos estes analisam os dados de e-mail e nome a fim de encontrar semelhanças. Com a classificação realizada, esse mapeamento também é armazenado no *database* para poupar tempo em outras utilizações da funcionalidade. Sobre os algoritmos mencionados, em síntese, estes funcionam de forma a analisar os nomes e e-mails estruturalmente a fim de buscar proximidades entre palavras usando, por exemplo, o algoritmo de *Levenshtein* e ainda, fazer várias combinações com caracteres especiais como +, -, . etc, a fim de confirmar ou não que dois identificadores diferentes podem remeter ao mesmo usuário.

É importante destacar que realizar esse mapeamento de identificadores de usuários acaba por ser um desafio até mesmo se feito manualmente por seres humanos, ou seja, atualmente, é praticamente impossível que esses resultados tenham ou estejam muito próximos do máximo grau de precisão, logo, essa implementação busca apenas auxiliar os pesquisadores dando um possível direcionamento em relação a este problema. Além disso, apenas os usuários de um repositório, individualmente, são agrupados em razão de os algoritmos requisitarem grande quantidade de tempo para serem executados quando se utilizam grandes quantidades de usuários, mostrando portanto uma limitação na implementação.

6. Resultados

Após a implementação da ferramenta, de forma a avaliar seu uso prático, ela foi utilizada para construir um grande *dataset* de pesquisa. Esse *dataset* é composto por 1.162 repositórios, selecionados entre os mais populares (números de estrela) em seis linguagens de programação diferentes. As linguagens escolhidas foram *JavaScript*, *C++*, *C#*, *Java*, *PHP* e *Python*, as quais são as mais utilizadas no GitHub[13]. Foi possível coletar e persistir uma grande quantidade de informações, que pesquisadores podem utilizar em suas pesquisas. A ferramenta apresentou bom desempenho, sendo capaz de extrair todos os dados de um repositório pequeno como *d3/d3* em pouco menos de 2 minutos e gastando pouco mais de 30 minutos para repositórios grandes como *facebook/react* além de recuperar informações de identificação de um total de 295.506 desenvolvedores.

Com relação às funcionalidades adicionais, apresentamos os resultados para cada uma delas nas próximas seções. Entretanto, antes de apresentar esses resultados, é importante apresentar como interpretar os valores que serão mostrados para melhor entendimento dos resultados.

[13]The six most popular programming languages in 2019 https://octoverse.github.com/#top-languages

- *Precision*: Pode ser caracterizado como a capacidade que a classificação teve em evitar falsos positivos, ou seja, dados que não eram válidos para o contexto da classificação, mas que foram classificados como válidos assim mesmo.
- *Accuracy*: Indica a taxa de classificações corretas em relação à todo o conjunto, isto é, a razão entre a soma de verdadeiros positivos e verdadeiros negativos em relação a soma de todos os valores.
- *Recall*: Este valor indica o valor com que os dados que eram válidos foram classificados como realmente válidos, logo, o ideal é que essa taxa seja 100% ou o mais próximo possível desse valor.
- *F-Meansure*: Este por sua vez, demonstra uma relação entre *precision* e *recall* de forma que quanto maior o valor, maior a confiabilidade que podemos ter em relação a acurácia, já que reflete diretamente nos valores de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos.

6.1. Classificação de repositórios

Para a classificação de repositórios, foram selecionados um total de 60 repositórios aleatoriamente dentre os 1.162 repositórios persistidos na ferramenta, de forma a facilitar o processo de validação da ferramenta. Ao adquirir esse subconjunto, todos esses repositórios foram manualmente rotulados pelo autor deste trabalho e por mais um aluno de computação. Após uma rodada de rotulação realizada separadamente, foi feita uma reunião para discutir as divergências e construção da rotulação final. Por fim, os dois algoritmos implementados nesse trabalho foram avaliados utilizando essa rotulação. Os resultados da execução são mostrados na Tabela 6.1 e demonstram ótimos resultados.

Tabela 1. Resultados da classificação de repositórios

Precision	**Accuracy**	**Recall**	**F-Meansure**
86.44%	86.67%	100%	92.70%

6.2. Mapeamento de usuários

Para validação do mapeamento de usuários foi pensado inicialmente em extrair os dados de usuários que foram extraídos nas inserções de repositório da ferramenta proposta nesse trabalho, entretanto seria necessária uma rotulação manual que poderia acarretar em um esforço a mais além de ser mais propício a erros, já que fazer esse mapeamento se torna difícil até mesmo para o ser humano.

Diante disso, foi extraído um conjunto de usuários diretamente do GitHub, pois ele através do mapeamento de usuários mantido por esse pode-se obter uma relação entre os usuários e os identificadores (nome/e-mail) que um determinado usuário cadastrou até o momento da extração desses dados. Essa estratégia visa garantir uma melhor taxa de confiabilidade dos dados para uso na validação das técnicas implementadas. Porém, devido aos tempos de execução dos algoritmos de mapeamento, conjuntos com muitos dados torna-se inviável para a execução, então foi extraída uma amostra de 2.000 ids que possuem mais de uma ocorrência de identificadores diferentes, aleatoriamente selecionados do conjunto de cerca de 400.000 usuários obtidos do GitHub. Desses 2.000 ids, todas

as suas ocorrências foram obtidas totalizando ao fim, 6.414 identificadores. Na Tabela 2 tem-se os resultados obtidos do mapeamento.

Tabela 2. Resultados dos algoritmos de mapeamento de usuários

Algorithm	Precision	Accuracy	Recall	F-Meansure
Simple	94.99%	91.26%	88.04%	91.39%
Bird	94.88%	91.36%	88.34%	91.49%
Improved	98.22%	85.75%	76.19%	85.82%

Podemos verificar que os resultados foram muito satisfatórios pelas altas taxas de *Recall* e *F-Meansure*. Ainda é importante elencar como foram obtidos os valores que permitiram calcular esses resultados, pois para esse mapeamento foi necessário um esforço maior para identificar como obter valores relacionados aos verdadeiros positivos (VP), verdadeiros negativos (VN), falsos positivos (FP) e falsos negativos (FN).

- Verdadeiros Positivos (VP): usuários que estão no grupo correspondente do primeiro usuário que foi inserido no grupo correspondente
- Verdadeiros Negativos (VN): usuários que não estão no grupo de outros usuários com o mesmo rótulo
- Falsos Positivos (FP): usuários que estão em grupos que possuem usuários com rótulos divergentes
- Falsos Negativos (FN): usuários que deveriam estar em um grupo com rótulo que já contém pelo menos um usuário, mas estão em outros grupos.

7. Conclusão

Ao fim deste trabalho pudemos contribuir com a construção de uma ferramenta para coleta e tratamento de dados do Github de forma automática, além de disponibilizar todas as funcionalidades descritas que se utilizam de algoritmos para classificações e, ainda, permitindo que tudo isso seja de fácil utilização para o usuário. Esta foi devidamente avaliada com resultados bem satisfatórios e também foi utilizada para construir um *dataset* relevante com 1.162 repositórios que está sendo utilizado para o desenvolvimento de pesquisas na área de engenharia de software, realizadas pelo grupo de pesquisa dos autores deste trabalho. Entretanto, é importante destacar que a ferramenta pode receber melhorias com relação a quais atributos de cada categoria de dados (*commits*, *issues* e *pullrequests*) são recuperados de um repositório. Além disso, pode-se fazer um aprimoramento dos relacionamentos entre as tabelas, verificando quais dados podem ser relacionados diretamente diante das diferentes tabelas já existentes na implementação. Por fim, como já foi citado, os algoritmos para mapeamento de usuários apresentam desempenho muito baixo e por isso só possibilitam a execução utilizando os usuários de um repositório individualmente, logo, cabe a trabalhos futuros uma forma para solucionar esse problema.

Referências

[1] I. Amit and D. G. Feitelson. Which refactoring reduces bug rate? 2019.

[2] G. Brito, T. Mombach, and M. T. Valente. Migrating to GraphQL: A Practical Assessment. 2019. URL https://homepages.dcc.ufmg.br/~mtov/pub/2019-saner-graphql.pdf.

[3] GitHub. Graphql API v4. https://developer.github.com/v4/guides/resource-limitations/, . URL https://developer.github.com/v4/guides/resource-limitations/. Acesso em 16 de Outubro de 2019.

[4] GitHub. About, . URL https://github.com/about. Acesso em 15 de Janeiro de 2021.

[5] M. Goeminne and T. Mens. A comparison of identity merge algorithms for software repositories. 2013.

[6] A. E. Hassan. The road ahead for mining software repositories. 2008.

[7] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. 2007.

[8] E. Kalliamvakou, L. Singer, G. Gousios, D. M. German, K. Blincoe, and D. Damian. The promises and perils of mining GitHub. 2014.

[9] I. Mistrík, J. Grundy, A. van der Hoek, Whitehead, and J. (Eds.). *Collaborative Software Engineering*. 2010.

[10] T. Mombach and M. T. Valente. GitHub REST API vs GHTorrent vs GitHub Archive: A Comparative Study. 2018. URL https://homepages.dcc.ufmg.br/~mtov/pub/2018-vem-thais.pdf.

[11] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. 2017.

[12] A. X. Pavim. Controle de versão com subversion, 2011. URL http://s2i.das.ufsc.br/seminarios/apresentacoes/controle-versao.pdf. Acesso em 29 de Dezembro de 2020.

[13] T. Siddiqui, M. Alkadri, and N. A. Khan. Review of Programming Languages and Tools for Big Data Analytics. 2017. URL https://pdfs.semanticscholar.org/1b91/eea121b7ea3051f8577a1d8373cf661164ec.pdf.

[14] Techopedia. Software repository. URL https://www.techopedia.com/definition/32890/software-repository. Acesso em 15 de Janeiro de 2021.

[15] C. Weiss, T. Zimmermann, A. Zeller, and R. Premraj. How long will it take to fix this bug? 2007.

[16] Álvaro Justen. Dados abertos, Python e PostgreSQL: a combinação perfeita, 2019. URL https://www.infoq.com/br/presentations/dados-abertos-python-e-postgresql-a-combinacao-perfeita/. Acesso em 31 de Outubro de 2019.