STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIX**, Number 2, 2024
DOI: 10.24193/subbi.2024.2.01

PYDSBUILDER – A DATASET BUILDER WRITTEN IN PYTHON DJANGO

LIVIU-MARIAN BERCIU

Abstract. Data mining and the analysis of open-source projects have become crucial in recent research, driven by the vast availability of data across multiple programming domains. This paper focuses on two main objectives: first, to present an experience report for designing a software quality data mining tool, and secondly, to provide an open-source solution, PyDs, that facilitates the creation of datasets specifically aimed at analyzing software quality attributes. PyDs, leveraging Python and the Django Framework, provides a comprehensive solution for researchers, encompassing data extraction from repositories, the application of software analysis tools, and the consolidation of results into a coherent format conducive to in-depth experimentation and analysis. This tool addresses the pressing need for effective data mining capabilities in evaluating software quality, allowing the research community to harness the full potential of the vast resources offered by open-source software projects.

1. Introduction

Open-source software development has seen a constant increase in popularity and adoption [17] in both industry and academia in the recent years. Open-source software (OSS) projects are software initiatives made freely accessible by their creators on various online platforms, such as GitHub and Bitbucket. These projects invite a broad audience to utilize the software, adhering solely to the terms of the associated open-source license. The widespread accessibility of these data, which span numerous programming languages, technologies, frameworks, and innovative solutions in various programming subdomains,

Received by the editors: 13 September 2024.
2010 *Mathematics Subject Classification.* 68N99.
1998 *CR Categories and Descriptors.* D2.0 [**Software Engineering**]: General – *Standards*; D2.9 [**Software Engineering**]: Management – *Software Quality Assurance*.
Key words and phrases. Data Mining, Software Quality Analysis Tools, Software Quality, Datasets, Dataset Builder, GitHub Mining.
© Studia UBB Informatica. Published by Babeş-Bolyai University
This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Licence.

has significantly propelled research efforts [13]. The integration of versioning systems plays a crucial role in these projects, as they not only facilitate collaborative development but also preserve a comprehensive history of the project's evolution. These historical data are invaluable, providing insight into development practices, trends, and the progression of software solutions over time, thereby enriching the research landscape with a deeper understanding of software development dynamics.

One topic that researchers have focused on when it comes to software projects is the assessment of software quality. Software quality is a multifaceted concept that refers to the degree to which a software product meets specified requirements, customer expectations, and user needs in a reliable, efficient, and maintainable manner. It encompasses various attributes such as maintainability, reliability, and security. The large number of OSS projects consisting of source code, issue tracking systems, and, more often than not, the competition brought about by projects tackling the same software domain has allowed researchers to review software quality in a transparent way [25].

Evaluation of software quality when it comes to open source data requires high volumes of information to be extracted, often due to the constant comparison between quality factors among a multitude of projects. Here, the concept of data mining comes into play [4], which implies tools and solutions that allow researchers to extract experiment data in a format that facilitates the execution of experiments and the drawing of conclusions about the problem studied. In the software engineering domain, examples of experiments include analyzing the technical debt of multiple Java projects [15] and studying maintainability when it comes to the long-term evolution of software projects [20]. When it comes to applying artificial intelligence to software engineering, more often than not, experiments require large amounts of data to be mined [26]. The mining of Github commit messages for natural language processing algorithms [12] and learning from the structure and history of the source code to automate the detection and fixing of bugs [8] are other applications where data mining is valuable.

Data mining often involves custom-created software solutions that facilitate data extraction from the internet. Some projects focus on the extraction and visualization of issue tracker data, such as [11] and [9], while others focus on offering a wider range of data extraction methods, focusing also on source code, commits and diffs, such as [24].

This paper outlines an experience report for the design and implementation of a data-mining tool specifically created for extracting datasets from software versioning systems, such as GitHub. The primary objective of this work is to present a detailed design proposal for a mining software repository tool

and to provide an open source artifact called PyDs Builder, drawing from our hands-on experience in developing a solution tailored to our research needs. In addition, we provide an experiment scenario to demonstrate the application and effectiveness of PyDs. We also provide suggestions on the application of the tool in empirical research studies. In developing this tool, we reference established methodologies for data extraction and analysis, ensuring a rigorous approach to our development process.

The insights gained from this project are diverse and offer valuable lessons on the challenges and strategies involved in designing efficient and scalable data mining tools. These lessons could serve as an important resource for researchers looking to develop or enhance their own tools, providing a practical foundation from which to approach similar projects.

The overview of the tool on the side of the article is divided into two main parts. The first one implies extracting issue tracking data and overall data pertaining to a software project. Currently, the tool only supports Github as a versioning system. The second one implies running software quality tools such as SonarQube[23] and SZZ [22] on the extracted data, in order to further refine the dataset and offer insights into the software quality attributes of a specific project.

The paper is structured as follows: in Section 2 we will outline an overview of a subset of the current data mining tools in academia. Section 3 discusses the conceptual design of the solution. In sections 4, 5 and 6, we will go through the architecture, database optimization, and usage of the application, reflecting along the way the design choices matching the concept. Section 7 offers a final overview of the challenges and experiences received while implementing the solution. Sections 8 and 9 will underline future work, possible extensions of the application, and concluding remarks.

2. Related Work

There has been several contributions addressing tools to mine software repository and we intend to present those that we believe are of interest in our approach.

PyDriller [24] is a software engineering tool created to help developers mine Git repositories. Its main features include extracting the repository's source code, differences, commits, and modifications. It is a framework capable of manipulating data and exporting them in the right format. The authors also focused on creating a performant tool, allowing for fast-onboarding and easy usage by developers. In opposition to our tool, PyDriller does not provide a capability of saving the parsed data in a database for persistence, nor does it allow extracting issue tracking data from a specific versioning system such as

GitHub. It does, however, work with any git repository, as it utilizes the git diff feature in order to parse repository data, making it, as stated beforehand, a fast program.

GrumPy [11] is a Python and Django Framework [5] developed web-tool with the purpose of mining issue data from issue trackers. With a focus on GitHub as its main issue tracker of choice, the tool offers database management capabilities without actually having database knowledge, it allows researchers to download repositories issues data in parallel, using multiple task queues and also provides access to data visualization features and statistical analysis of the mined data. There are some similarities and differences between GrumPy and our tool. On the one hand, both tools are implemented using Python and Django. They use custom databases, different from the SQLite default that Django comes with (MongoDB for GrumPy, PostgreSQL for our tool), and the same is done for the queue management technology (Redis, as opposed to RabbitMQ). Both tools also offer issue data mining using the same Github API technology. On the other hand, GrumPy offers a visual overview on the mined issues through a web platform, also targeting people without prior knowledge of programming, while our tool is more technical, offering just the Django admin panel for database visualization. In contrast, we mine more Github information about repositories, such as Commits and Issue timelines, while also allowing for custom tools execution in order to properly build a dataset.

GHTorrent is another project that retrieves data from Github repositories [10]. With this tool, the authors aim to provide persistent data and event streams to the research community, as a service [10]. Data retrieval from Github is done using a specifically implemented crawler, which queries for raw data using the Github API. The extracted data are then sent to a set of RabbitMQ queues, which further refine the data. It is important to note here that this mechanism allows replication on multiple hosts, circumventing the API limits by using different API tokens from multiple research teams. Data persistence is done using MongoDB, due to the database technology's capability of scaling and handling of large amounts of data. While the GhTorrent solution allows for high amounts of data to be processed and put to the community service, it does not process Github issues and also covers a wide view of Github repositories, making it hard to target a specific niche. Instead, our tool provides researchers with a solution in building their own dataset for their specific needs, using a similar host distribution approach in order to allow higher API limit thresholds and also supports tool execution, in order to further enhance and interpret the data extracted.

Lastly, Perceval [7] is a command line tool that supports a multitude of data sources to retrieve data from, such as mailing lists, version control systems, ticketing tools, and Q/A solutions. It comes as either a Python library or as a command line tool, allowing for flexible usage. It is composed of multiple back-end implementations targeting different data sources, with the possibility of extension to support new entries, abiding by the user's needs. While both our tool and Perceval strive to offer data extraction and easy development extension to researchers, there are two main differences to take into consideration. Perceval [7] offers JSON-format data dumps, leaving the user to carry the data persistence responsibility. We, on the other hand, use PostgreSQL in order to save data directly for later use. The second difference is that there is no analysis tools support on the raw data, leaving users to implement/use their own data analysis pipelines. In our case, we support three out-of-the-box tools for data analysis, allowing researchers to easily automate flows and use the solution for an end-to-end dataset generation flow.

More often than not, tools created for data mining depend largely on research purposes, and in many cases, they are created to address the specific needs of a scientific experiment. For this reason, existing solutions are not always enough, as they often have to aggregate information from different tools, making most of the tools rigid and hard to extend for research purposes other than the ones they were built for.

3. Tool design

The requirement of extracting data from different sources in a consistent and organized manner, which can be utilized for various experiments and easily expanded, often leads to the need to create a software solution that encompasses these aspects. When working with large amounts of data, the need to structure and normalize the data is paramount and often involves employing different algorithms that scan, extract and aggregate the needed information in a form that facilitates processing. In order to do this, typically different software tools are used and then their results aggregated in one form or another, necessitating more effort from the researchers. The purpose of this article is outlining the experience and actually creating a solution that can aggregate data from different streams, under different formats, in one, uniform, and general format that can be used for creative experiments and extended as the researchers see fit.

An important aspect in creating a solution like this is **to decide what data formats are supported**. Usually, data are extracted using API requests that are provided by the data sources, or downloaded directly under the form of files. Data thus often appears in JSON, CSV, YAMl or SQL formats, among

many others, and are then processed into a single, uniform standard. For our case, we see the need of supporting data conversion from the different formats enumerated beforehand into a standardized format, such as SQL, that enables researchers to conduct complex queries and analyses, and can also scale as the amount of data increases.

Another important aspect is offering a way in running data extraction for large periods of time without the constant supervision of the person using the solution. This is done by implementing automation, under the form of task queues, that allows cloud deployment and a clear set of instructions on how to ensure data is processed continuously. Furthermore, the solution should be implemented in a programming language that is popular in the programming community, has a low learning curve and offers many out-of-the-box features that developers can use, ensuring quick adoption and extension. Thus, another important issue to follow is **automation of data extraction**.

Obtaining the data in the desired format should allow further processing by feeding them to a data pipe of custom tools, each with its own purpose and end results, suiting the specific needs of the researcher. The data obtained from the execution of the tools will then be saved in the same uniform structure decided beforehand, leaving the decision of further processing or concluding the experiment in the hands of the user. In conclusion, if data from several tools are needed, **the decision about flow is important**.

4. PYDS BUILDER SOLUTION

In the following sections, we introduce the specifics and implementation of a data mining tool with a focus on the three important guidelines underlined above: **automation of data, format specification** and **flow decision**.

PyDs Builder is a web, API-based solution that aims to offer a way for researchers to create experimental datasets. It is built using Python and Django Framework, leveraging the capabilities offered by both technologies, such as fast development, scalability, excellent documentation, and an ORM system allowing intuitive database manipulation.

Its main focus is extracting repository data from versioning systems, with the incipient implementation offering support for Github. Data is processed into the desired form and inserted into a custom database, following a **pre-established SQL schema**. Data can be processed further by custom tools in order to complete an experiment's data acquisition goals. Afterwards, the data can be used as the researchers see fit, either by publishing a totally new dataset or feeding the data into an artificial intelligence solution, drawing new conclusions and desired results.

An overall overview of the solution features is enumerated as follows:

- Allow tool interaction through an API interface
- Extract Github repository data such as issues, issue timelines, commits details and source code
- Execute software quality tools such as SonarQube [23], SZZ [22] and PyRef [3] and ensure data persistence
- Provide automation for fast data processing
- Allow contribution and code extension through project modularity and intuitive programming interface.

4.1. **ARCHITECTURE.** From an architectural point of view, a modular monolith approach was used in order to build the application. This approach was taken due to a few considerations. First, Django is a Python framework that is designed as a monolith by default. It consists in a single code base, a shared database and a single deployment. Second, Django has a native application support, meaning it can be designed into modules such that a single module can hold a single responsibility. From those two points came the third, which implied that, by using modularization, we managed to create separate code units for each tool that we support, setting boundaries so that the shared code is held in common modules and tools don't have to interfere with each other. This further enriches the mission of allowing developers to contribute to the PyDs solution by simply using the common modules already defined to create a new module for their own specific needs.

Next, to enable automation, RabbitMQ [21] and Celery were used in order to setup task queues. The task queue functionality provided by Celery allows the application to perform asynchronous work in the background, while RabbitMQ is the message broker that Celery uses in order to exchange messages and run tasks. In this way, the application can be started, for example, on a virtual machine in the cloud and perform work without constant supervision from the user. Furthermore, it bypasses the HTTP request limit, allowing a task to take from a few minutes to a few hours, depending on the needs, without the risk of timeouts. The same rationale can be replicated on multiple instances of virtual machines, allowing parallel execution and data extraction from multiple sources at the same time. With this, we have covered the point about **setting up an automation mechanism for data extraction**.

Data persistence was managed using PostgreSQL as the database engine of choice. The reason this database technology was used, as opposed to using the default SQLite Django database, was the following: it is highly scalable, handling large volumes of data and concurrent users efficiently, it can be hosted in the cloud, it supports JSON data types, and it has a robust security. In contrast, SQLite is a self-contained system that has no server setup, has limited

scalability and concurrency, and lacks advanced features found in more complex RDBMS (Relational Database Management System/s), such as ACID transactions, complex query support and concurrency control.

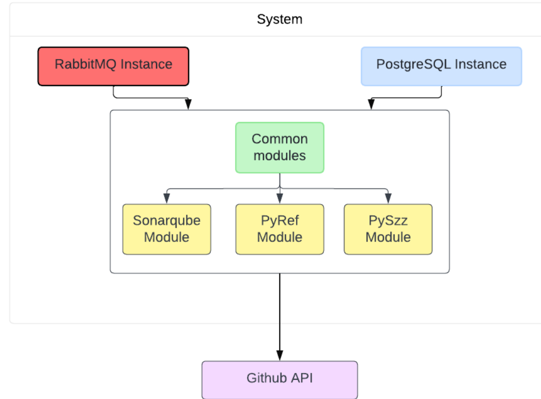

FIGURE 1. PyDs Architecture Diagram

Figure 1 represents a visual overview of the architecture of the system. It can be observed how the main system, comprised of common modules and the specialized tools module, communicates with the internal modules of the database instance and the task queue instance, and the external connection to the Github API service. Common modules implement command-line functionality and a wrapper over the Github API functionality. The modules of specialized tools implement wrappers over SonarQube [23], PyRef [3] and PySzz [22].

4.2. **TOOLS AND FLOWS.** The solution code base includes support for three software quality tools. The tools were chosen to serve the research objectives and because of their relevance in the software quality research space which provides ease of use for researchers to build datasets with the results of their execution. The remainder of this section follows a short tool introduction and the steps necessary for researchers to run the tools and extract data using our solution. More details about tool usage can be found in Section 6.

In the context of enhancing software refactorings, PyRef [3] emerges as a dedicated tool optimized for projects developed in Python. This tool conducts a comparative analysis between two versions of a project to accurately identify the refactorings that have occurred. PyRef is specifically engineered to detect a suite of nine method-level refactoring operations, which include: Rename Method, Add Parameter, Remove Parameter, Change/Rename Parameter, Extract Method, Inline Method, Move Method, Pull Up Method,

and Push Down Method. PyRef's ability to systematically identify and categorize method-level refactorings enhances its usefulness in gaining a more comprehensive understanding of software evolution and maintenance practices in Python-based projects.

In order to execute PyRef on a project, the PyRef repository must be cloned in the solution root folder. Afterwards, the first thing to do is initialize a repository in the system by calling the *Create repository* API url. The last step is calling the PyRef API url specifying the repository and the commit hash to be analysed. In case a commit hash is not provided, the latest release of the repository, if any, is fetched from the database. It is important to note that the tool will call the command ‘git rev-list ¡commit¿‘ and will compare all pairs of commits that the command returns. The excution results will be saved into the database.

The SZZ algorithm is used in software engineering to automatically identify bug-introducing commits in version control systems. It operates by tracing back from bug-fixing commits to the original commits where the bugs were introduced, using the version history of a software project. An open source implementation of the SZZ algorithm is PySZZ [22], a tool which we selected based on its Python implementation and command line execution capabilities, allowing quick integration with our tool.

For obtaining PySZZ data, the main two steps are preparing the input data for the tool and executing the tool on the input data. The configuration for SZZ is found in chapter 6. Calling the *create_input_file* and *execute* API endpoints on the desired repository will create the input file, will execute the tool on the input file and then, calling the *extract* endpoint, will save the result into the database.

SonarQube is a static code analysis tool designed to enhance software quality standards [23]. It seamlessly integrates into the development workflow, offering multi-language support for static analysis rules and classifying code based on the software quality dimensions of reliability, security, and maintainability. Since its inception in 2008, SonarQube has evolved significantly, as evidenced by its frequent updates and the scholarly attention it has received, including discussions in various scientific articles [15], [14] [19]. A noteworthy development in its evolution is the shift from traditional issue classifications such as bugs, vulnerabilities, and code smells towards the adoption of "Clean Code" principles. These principles are further delineated into categories such as consistency, intentionality, adaptability, and responsibility, each defining specific attributes of code quality. In order to execute the SonarQube flow, users have to configure SonarQube on their work stations. This can be done

either by following the installation steps from SonarQube's official documentation or by creating a docker container to hold the service. Subsequently, making sure that the repository was already initialized in the database, the endpoint *sonarQube/analyze* can be called. It is a POST method receiving a body containing the repository owner, repository name, release tag, or commit hash. It will execute SonarQube using the received information and save all the SQ issues and SQ measures found.

We can conclude this section by reiterating the importance of **deciding about a clear flow of data extraction** when it comes to integrating a tool. From setting the incipient data such as the repository to be analyzed, to writing the wrapping code over the tool interface, whether it is command-line or web-based, to finally extracting and processing data in an automated manner, each step has to be properly implemented and executed in the correct order so that data acquisition is successful.

More endpoints are available in the project codebase due to various experiments. They are left there for researchers to explore and use them as they see fit.

5. Database optimization

The codebase includes a database architecture and entities that were used in order to run different software engineering experiments that include data extraction and arrangement. Next, we will take a look at some database best practices and optimizations that can be done so that experiments run optimally and the dataset is arranged as needed.

- **Data Normalization:** Minimize redundancy, improve data integrity but balance to avoid overly complex queries. Try not to create cyclic dependencies between table and keep a tree structure. For example, the Repository can be the main table to which the other parts of the database connect, but the Repository will not reference any of it's dependents.
- **Indexing:** Accelerate record retrieval in frequently searched columns, balancing read performance with write overhead. The SQ Issue table, which can contain millions of results, can have an index on the *commit_hash* field.
- **Partitioning:** Divide large datasets into manageable segments for improved performance and easier management, tailored to query patterns. For example, tables for SonarQube and PyRef do not have any dependencies to eachother, being separated in their own semantic field, ensuring data integrity and proper separation.

- **Denormalization:** Introduce redundancy selectively to speed up read operations where beneficial, with careful consideration of trade-offs. Many-to-many tables can be added to avoid join links such as Repository - Issue - Timeline - Commit, or table fields that can reference the main entity (Commit references repository ID directly).
- **Concurrent Access and Locking Strategies:** Implement suitable locking mechanisms to maintain data integrity during simultaneous access, optimizing for the specific access pattern. Proper selection of the database engine ensures proper concurrent access, hence the choice of PostgreSQL over SQLite.
- **Efficient Query Design:** Craft queries to only fetch necessary data, using joins effectively, and optimize regularly based on usage.

6. Execution and Usage

6.1. **Installation.** There are a few steps that have to be completed in order for the tool to run successfully. The first step implies installing the project dependencies using Python's pip command. Afterwards, docker-compose [6] must be used in order to start the containers necessary for the queue orchestrator, the database and tools such as SonarQube. The tool is then started by running the default Django command for starting a server. An important part after starting the server is to run the database migrations in order to setup the correct database schema to use. The exact steps for installation are enumerated in Listing 1.

```
# Install the requirements
pip install -r requirements.txt
# Start the docker containers
docker-compose up -d
# Start the server
python3 manage.py runserver
# In another terminal, run the
# migrations for the database
python3 manage.py migrate
```

Listing 1. PyDs Setup Steps

6.2. **Configuration.** The tool configuration is done inside the main Django configuration file, namely 'settings.py' found in the main application folder. The settings file contains general information about configuring a Django project, such as the logging level, database connection credentials, installed applications, middlewares, celery queues and custom variable defined by the

user. Although the public repository will contain a pre-completed configuration file with examples for values, the code from Listing 2 exemplifies some of the important configuration variables and their meanings.

```
# SonarQube
SONARQUBE_URL = < SonarQube URL to call>
SONARQUBE_TOKEN = <project analysis tokens>
SONARQUBE_GLOBAL_TOKEN = <>
SONARQUBE_USER_TOKEN = <>
SONARQUBE_SCANNER_URL = <URL of the Sonar scanner if not
    installed locally>
SONARQUBE_PROJECT_KEY = <specific project key to scan>
SONARQUBE_USERNAME = <login username>
SONARQUBE_PASSWORD = <login password>

# Github
GITHUB_TOKENS = [<list of github authentication tokens
   for API calls]
GITHUB_ROOT_DIR = <root directory for github projects
   cloning>

# PySZZ
SZZ_INPUT_FILES_FOLDER = <location of files pre-prepared
    for PySZZ execution
SZZ_OUTPUT_FILES_FOLDER = <location of files after PySZZ
    execution>
SZZ_GITHUB_TOKEN = <specific github token to run only
   with PySZZ>
```

Listing 2. Configuration Variables Example

6.3. **Usage.** The basic usage of the application is done through the REST API exposed through Django [5] views. Django has a MVT (Model View Template) architecture, allowing developers to write API endpoints in specialized *VIEW* classes. A subset of the available API calls is found in Table 1. One important note is that, for mining Github data, we have used a similar endpoint format as in the official Github API documentation, in order to offer familiarity for users who have prior experience with the Github API. For exemplification

purposes, we used the Ansible [2] and Pandas [16] repositories, as they are some of the largest Python Github open source projects.

6.4. **Data visualisation.** There are two options available for visualizing the extracted data. The first option is to utilize a specialized SQL visualization tool that enables the examination of the database, execution of queries, and visualization of the overall database structure. Alternatively, the Django admin panel can be used to gain insights directly into the selected tables included in the admin dashboard. Another way to view the data is by implementing fetch requests in the API views of the tool.

7. EXPERIENCE REPORT

Developing PyDs from the ground up inevitably came with its own unique set of obstacles. The following paragraphs outline the challenges encountered during the development process of PyDs.

- We have made the decision to use SQL for data serialization and database schema, instead of opting for the more direct JSON and/or CSV formats that are commonly used for raw data. Although the SQL approach may be more complex, we believe that the long-term benefits, such as optimization and a rich feature set, outweigh the disadvantages.
- The objective was to develop a universal approach for running external tools. Since each tool has its own specific set of instructions for execution, we were able to devise a general method by utilizing the command line capabilities and creating wrapper classes and modules for each individual tool.
- The selection of an appropriate database for automation is crucial. Initially, SQLite was utilized as the preferred database option. However, it was soon realized that SQLite has limitations in terms of capabilities and is not suitable for distributing the workload across multiple machines. As a result, PostgreSQL was chosen as it possesses the necessary capabilities and is compatible with cloud hosting.
- The limitation of data intake was also influenced by the rate limits imposed by open source project platforms. To address this challenge, we developed a wrapper class that can utilize access tokens from multiple researchers. This allows for continuous data retrieval, as when one token reaches its rate limit, the next token in the queue is automatically used.

Purpose/Meaning	**API Call**	**Method**
Create a repository database entry	`/mining/repo/github/pandas-dev/pandas`	POST
Delete a repository from the database	`/mining/repo/ansible/ansible`	DELETE
Fetch all issues for a repository	`/mining/repo/github/ansible/ansible/issues`	POST
Fetch a specific issue for a repository	`/mining/repo/github/ansible/ansible/issues/123`	GET
Extract timelines for all already extracted project issues	`/mining/repo/github/ansible/ansible/issues/timeline`	POST
Extract an issue's specific timeline	`/mining/repo/github/ansible/ansible/issues/4720/timeline`	GET
Run a SonarQube analysis for a specific Github issue	`/sonarqube/repo/github/pandas-dev/pandas/issue/36`	POST
Run a SonarQube analysis for a commit hash or release tag	`/sonarqube/analyze`	POST
Run PySZZ create input file	`/szz/repo/pandas-dev/pandas/create_input_file`	POST
Run PySZZ execute	`/szz/repo/pandas-dev/pandas/execute`	POST
Run PySZZ extract	`/szz/repo/pandas-dev/pandas/extract`	POST
Run PyRef on a repository	`/pyref/repo/ansible/ansible`	POST

TABLE 1. API Requests Overview

8. RESEARCH POSSIBILITIES AND EXTENSION

PyDs is a software solution that can be valuable both for researchers and independent developers. For researchers, it provides a working framework for running complex experiments in an automated way, ensuring data extraction for very large datasets. While initially created for software quality experiments, including running specialized software quality tools in order to uncover maintainability, technical debt and reliability attributes of software projects, it can be extended to allow artificial intelligence integration, it can support multiple database engines such as MongoDB and MySQL and it can be enriched to extract data from other versioning and issue tracking systems such as Bitbucket and Jira. Independent developers can greatly benefit from the PyDs dataset generation tool by gaining enhanced insights into their projects' maintainability and reliability, and by integrating analytics solutions on the extracted data to suit their specific needs. The tool's ability to integrate with various development tools and database engines streamlines the development workflow while offering opportunities for skill enhancement through interaction and extension of the tool. This makes PyDs a versatile and valuable resource for independent developers looking to innovate, improve project health, and efficiently manage their software development processes.

PyDs is also offered as an open source project, allowing community contributions and being subject of the FreeBSD license [1]. The source code can be found by accessing the following link: `https://figshare.com/s/5dd7e88ba4e329acfa4a`.

8.1. **Possible scenario for tool usage.** In order to expose the features of the tool, we imagine a possible research scenario in which using PyDs Builder can be beneficial. The objective of this study is to monitor and improve software quality throughout the development lifecycle of a project. The aim is to track the project's quality trajectory, from commit to commit, and visualize the evolution of issues to make informed decisions for continuous quality improvement.

The methodology involves a streamlined process using the data mining tool.

- Selection of a Python Project: Choose a project with a sufficiently large code base.
- Data Extraction: Utilize PyDs Builder API to fetch project data and issues from GitHub.
- Quality Analysis: Analyze the project using SonarQube through the integration with PyDs Builder.
- Issue Prioritization and Resolution: Identify critical issues affecting quality and address them systematically.

- Quality Trajectory Assessment: Evaluate changes in quality metrics over time to gauge improvement or deterioration.

Upon analyzing the extracted data, notable trends in code quality metrics can be observed, particularly in the occurrence and distribution of specific types of issues across various developmental stages. Patterns between SonarQube-reported issues and GitHub issues can be associated, shedding light on the collaborative dynamics of the project contributors as they addressed quality concerns.

Using PyDs allows for tracking the quality trajectory of the project, identifying both problematic and beneficial commits and changes. This analysis not only identifies areas that need improvement, but also facilitates proactive interventions to increase overall project quality and stability. The insights gained provide a detailed view of the project's quality dynamics, highlighting both strengths and areas for improvement in software development practices.

9. CONCLUSIONS

Data mining and open source project analysis have been one of the important subjects of academia in recent years, with data availability comprising multiple programming domains being one of the main factors for its ascendance.

In this paper, we have introduced PyDs, a Python with Django Framework solution that enables researchers to generate datasets for various scientific criteria, primarily focusing on software quality attributes experiments. However, it also allows for potential expansion to apply artificial intelligence to software engineering. We have covered the conceptual design of the application, its underlying principles, and delved into the implementation steps and different perspectives. We have provided detailed instructions for setting up, configuring, and using the application to facilitate quick onboarding for readers. Finally, we have explored potential avenues for extending the application and shared our experience in developing this intricate tool.

The subsequent stages of the application involve its practical implementation in scientific settings and research projects. We are confident that PyDs can serve as a reliable solution in these scenarios, allowing researchers to utilize it for data extraction in popular software quality tools such as SonarQube and SZZ algorithm implementations. Currently, PyDs only supports Github, but there are future plans to expand its capabilities to integrate with the Jira ticketing system and extract CI/CD pipelines data from platforms like Jenkins, as we believe project building steps and performance indicators can provide valuable research data. In the end, PyDs Builder has been successfully utilized

to create a dataset focused on open-source Python projects [18], highlighting its flexibility and practical utility.

References

[1] The freebsd license, 2023.

[2] Ansible, I., et al. Ansible: Radically simple IT automation. `https://github.com/ansible/ansible`, 2023.

[3] Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., Bavota, G., and Lanza, M. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2021), pp. 136–141.

[4] Chaturvedi, K., Sing, V., and Singh, P. Tools in mining software repositories. In *2013 13th International Conference on Computational Science and Its Applications* (2013), pp. 89–98.

[5] Django Software Foundation. Django.

[6] Docker, Inc. Docker: Empowering app development for developers, 2023. Accessed: 2024-02-17.

[7] Dueñas, S., Cosentino, V., Robles, G., and Gonzalez-Barahona, J. M. Perceval: software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (New York, NY, USA, 2018), ICSE '18, Association for Computing Machinery, p. 1–4.

[8] Elmishali, A., Stern, R., and Kalech, M. An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence 69* (2018), 147–156.

[9] Fiechter, A., Minelli, R., Nagy, C., and Lanza, M. Visualizing github issues. In *2021 Working Conference on Software Visualization (VISSOFT)* (2021), pp. 155–159.

[10] Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. Lean ghtorrent: Github data on demand. pp. 384–387.

[11] Jr., J. M., Santana, R., and Machado, I. Grumpy: an automated approach to simplify issue data analysis for newcomers. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering* (New York, NY, USA, 2021), SBES '21, Association for Computing Machinery, p. 33–38.

[12] Kourtzanidis, S., Chatzigeorgiou, A., and Ampatzoglou, A. Reposkillminer: identifying software expertise from github repositories using natural language processing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2021), ASE '20, Association for Computing Machinery, p. 1353–1357.

[13] Krogh, G. v., and Spaeth, S. The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems 16*, 3 (2007), 236–253.

[14] Lenarduzzi, V., Lomio, F., Taibi, D., and Huttunen, H. On the fault proneness of sonarqube technical debt violations: A comparison of eight machine learning techniques. *CoRR abs/1907.00376* (2019).

[15] Lenarduzzi, V., Saarimäki, N., and Taibi, D. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering* (Sept. 2019), PROMISE'19, ACM.

[16] McKinney, W., et al. pandas: a powerful Python data analysis toolkit. https://github.com/pandas-dev/pandas, 2023.

[17] Midha, V., and Palvia, P. Factors affecting the success of open source software. *Journal of Systems and Software 85*, 4 (2012), 895–905.

[18] Moldovan, V.-A., Berciu, L.-M., and Patcas, R.-D. The python software quality dataset. In *50th Euromicro Conference Series on Software Engineering and Advanced Applications* (2024).

[19] Molnar, A.-J., and Motogna, S. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (New York, NY, USA, 2020), ESEM '20, Association for Computing Machinery.

[20] Molnar, A.-J., and Motogna, S. A study of maintainability in evolving open-source software. In *Evaluation of Novel Approaches to Software Engineering* (Cham, 2021), R. Ali, H. Kaindl, and L. A. Maciaszek, Eds., Springer International Publishing, p. 261–282.

[21] RabbitMQ Team. Rabbitmq: Open source message broker. https://www.rabbitmq.com/, 2023. [Online; accessed 10-February-2024].

[22] Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., and Oliveto, R. A comprehensive evaluation of szz variants through a developer-informed oracle. *Journal of Systems and Software 202* (2023), 111729.

[23] SonarSource. Sonarqube: Continuous code quality inspection tool, 2023. [Online; accessed 10-February-2024].

[24] Spadini, D., Aniche, M., and Bacchelli, A. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2018), ESEC/FSE 2018, Association for Computing Machinery, p. 908–911.

[25] Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P. J., Samoladas, I., and Stamelos, I. Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science 233* (2009), 5–28.

[26] Wangoo, D. P. Artificial intelligence techniques in software engineering for automated software reuse and design. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)* (2018), pp. 1–4.

Babeș-Bolyai University, Faculty of Mathematics and Computer Science, 1 Mihail Kogălniceanu, Cluj-Napoca 400084, Romania

Email address: liviu.berciu@ubbcluj.ro