BuGL

Documentation

Table of contents

Introduction
Dataset structure
Dataset Extraction
Folder structure of the database
Scripts

Introduction:

This document contains an overview of

- Structure of BuGL dataset
- Methodology for Dataset Curation
- Dataset Schema

Please refer to the corresponding paper for the motivation behind the construction of this dataset

BuGL dataset consists of 54 Github projects of four different programming languages namely C, C++, Java, and Python with around 10,187 issues (which are fixed via pull requests) with required metadata information (refer to dataset structure for more information on the metadata).

Dataset Structure:

BuGL dataset is available in two formats namely “json” and “xlsx” formats and it also contains the source projects from the Github.

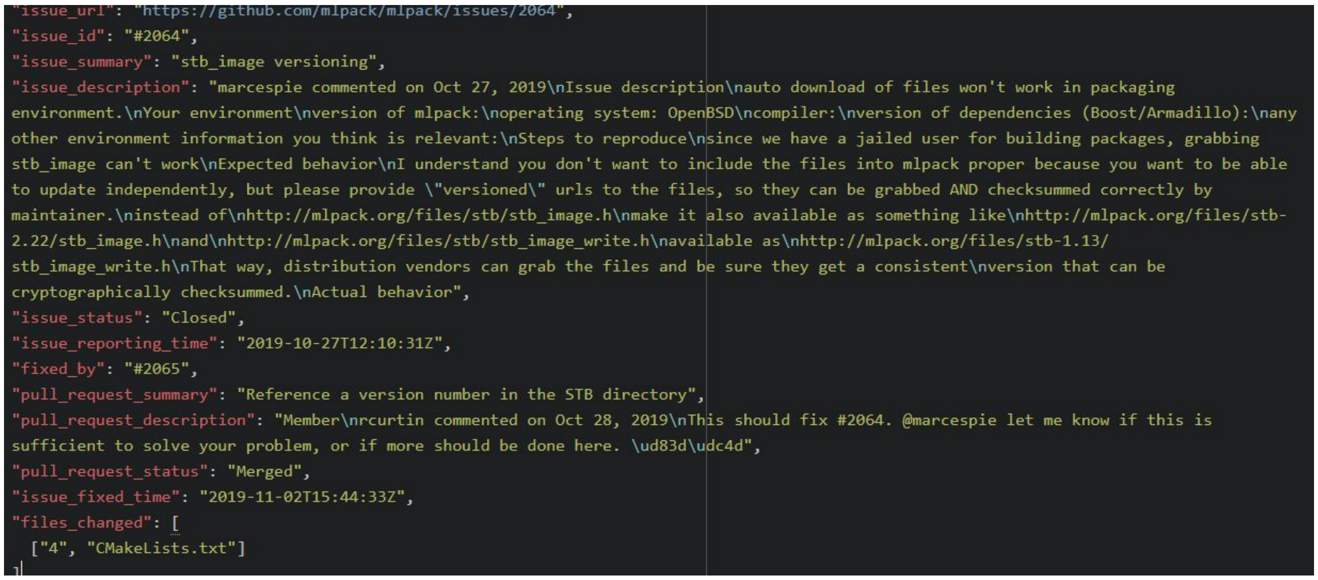
“xlsx” format contains the following metadata:

- Sr. No.
- Issue Url
- Issue ID
- Issue Summary
- Issue Description
- Issue Status
- Issue Reporting time
- Issue Fixed time
- Fixed By (Pull Request ID)
- Pull Request Summary
- Pull Request Description
- Pull Request Status
- Files changed
- Number of lines changed in each file

JSON format of the data element

```
{
        “open_issue”  :{
                                        “S.No of open issue” : {
                                                    “Issue_id” : “value1”,
                                                    “Issue_url” : “value2”,
                                                    “Issue_summary” : “value3”,
                                                    “Issue_desrcription”: “value4”,
                                                    “issue_status” : “value5”,
                                                    “issue_reporting_time”  : “value6”
                                              }
                           },
        “Closed_issue” :{
                                        “S.No of closed issue” : {
                                                    “issue_id” : “value1”,
                                                    “issue_url” : “value2”,
                                                    “issue_summary” : “value3”,
                                                    “issue_desrcription”: “value4”,
                                                    “issue_status” : “value5”,
                                                    “issue_reporting_time”  : “value6”,
                                                    “issue_fixed_time” : “value7”,
                                                    “fixed_by” : “value8”,
                                                    “pull_request_summary”: “value9”,
                                                    “pull_request_description”:“value10”,
                                                    “pull_request_status”: “value11”,
                                                    “files_changed”: “value12”
                                              }
                            }
}
```

Each json and xlsx files are named after the name of the Github repository. Fig 1 is an example of the metadata information of a closed issue in json format and Fig 2 explains the database schema.



```
"issue_url": "https://github.com/mlpack/mlpack/issues/2064",
"issue_id": "#2064",
"issue_summary": "stb_image versioning",
"issue_description": "marcespie commented on Oct 27, 2019\nIssue description\nauto download of files won't work in packaging
environment.\nYour environment\nversion of mlpack:\noperating system: OpenBSD\ncompiler:\nversion of dependencies (Boost/Armadillo):\nany
other environment information you think is relevant:\nSteps to reproduce\nsince we have a jailed user for building packages, grabbing
stb_image can't work\nExpected behavior\nI understand you don't want to include the files into mlpack proper because you want to be able
to update independently, but please provide \"versioned\" urls to the files, so they can be grabbed AND checksummed correctly by
maintainer.\ninstead of\nhttp://mlpack.org/files/stb/stb_image.h\nmake it also available as something like\nhttp://mlpack.org/files/stb-
2.22/stb_image.h\nand\nhttp://mlpack.org/files/stb/stb_image_write.h\navailable as\nhttp://mlpack.org/files/stb-1.13/
stb_image_write.h\nThat way, distribution vendors can grab the files and be sure they get a consistent\nversion that can be
cryptographically checksummed.\nActual behavior",
"issue_status": "Closed",
"issue_reporting_time": "2019-10-27T12:10:31Z",
"fixed_by": "#2065",
"pull_request_summary": "Reference a version number in the STB directory",
"pull_request_description": "Member\nrcurtin commented on Oct 28, 2019\nThis should fix #2064. @marcespie let me know if this is
sufficient to solve your problem, or if more should be done here. \ud83d\udc4d",
"pull_request_status": "Merged",
"issue_fixed_time": "2019-11-02T15:44:33Z",
"files_changed": [
  ["4", "CMakeLists.txt"]
]
```

Figure 1: Sample Bug report

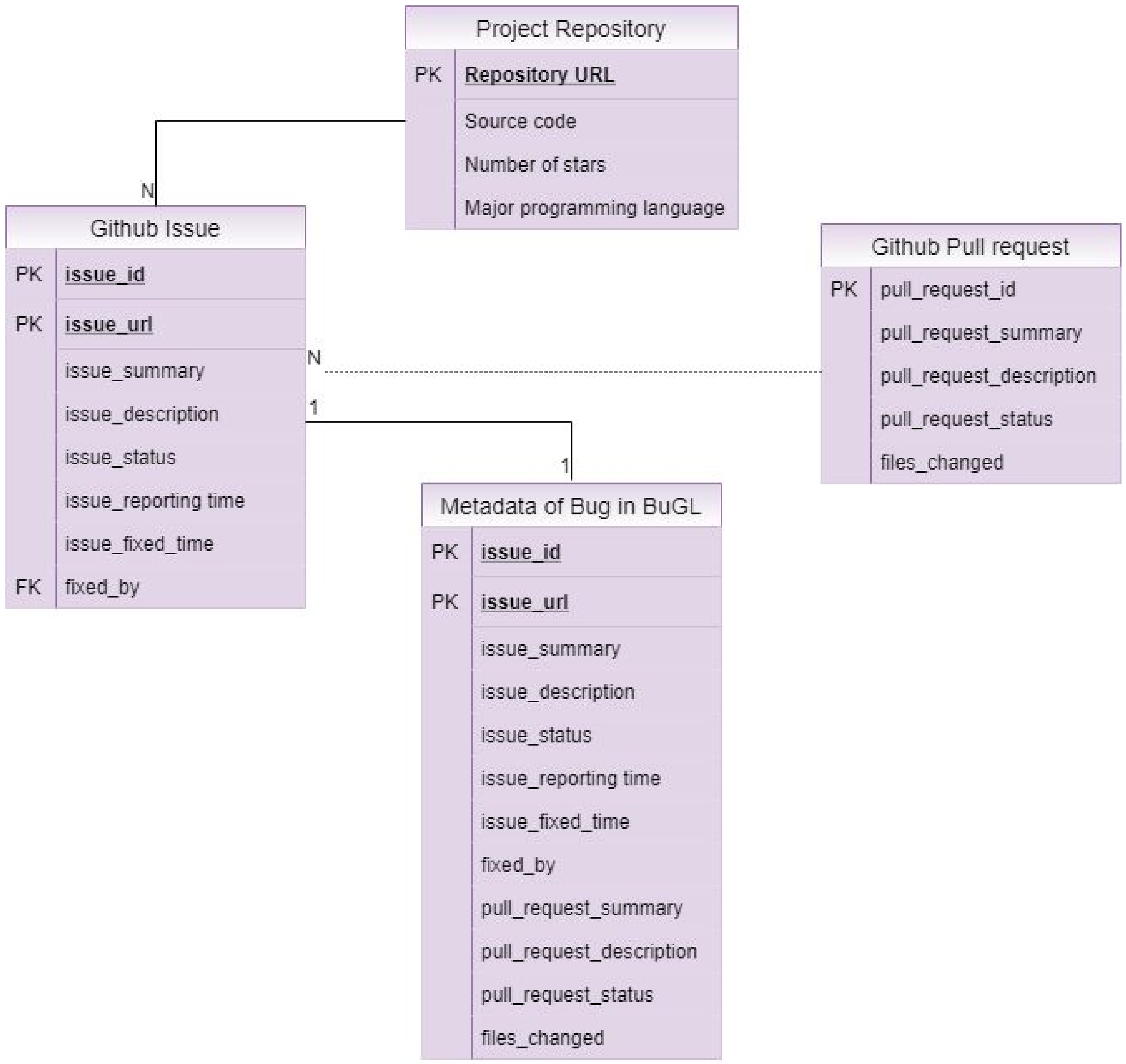


Figure 2: Database schema

Dataset Extraction:

The dataset was collected using self-written python scripts using selenium and chrome driver. Steps for extracting the dataset:

Step1:

The first step is to select active and popular open-source Github projects. The feature, Github stars, can be used as an index to measure the user's interest in the project. To select the projects we used the following criteria.

1. Initially using Github linguistic feature we selected the major language of the project.
2. Sorted the projects according to the number of stars.
3. Manually selected those projects with at least **500** issues and pull requests.
4. Repeated steps 1 to 3 for the remaining languages.

We carefully selected the projects with at least **500** issues and pull requests with the intuition that at least a good number of issues are closed by the pull requests to be included in our dataset.

In this way, we selected a total of **54 projects** from C, C++, Python, and Java such that each programming language has around 2500 issues/bugs that are closed by merging the pull requests.

Step 2:

Then downloaded the source code (in .zip format) of all the selected projects and stored them according to their major language in respective folders. i.e., C, C++, Python, Java.

Path of the projects w.r.t to the root directory is **"Language\Projects\"** where Language is either C or C++ or Java or Python.

Step 3:

Manually analyzed Github page for CSS selectors like classname, id, XPath etc., which can be used to automate the process of extracting the required metadata from the issues and pull requests in the Github. Using these selectors we wrote a python script that extracts the metadata with the help of chrome driver and selenium.

Note: The CSS selectors used in our code may vary from time to time and browser to browser (we used chrome driver) as they depend on the Github website.

Step 4:

Next step is to extract the metadata and curate the data in the required format. We are storing the data in "json" and "xlsx" formats.

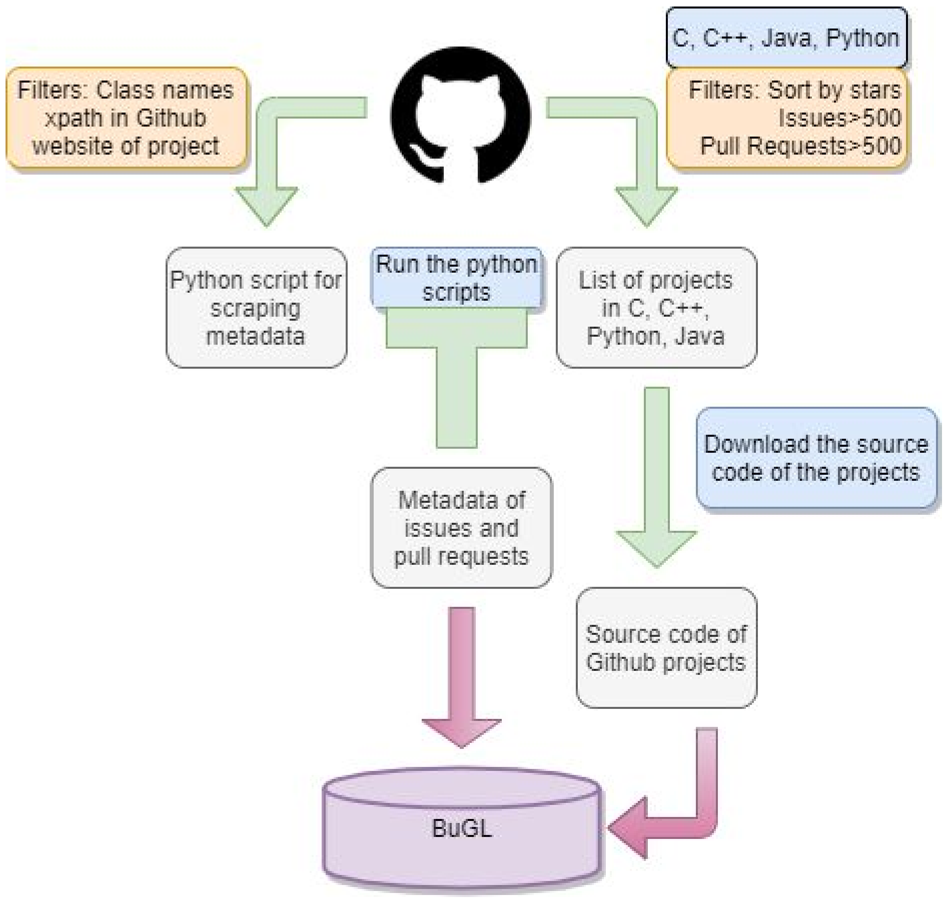

C, C++, Java, Python
Filters: Class names xpath in Github website of project
Filters: Sort by stars Issues>500 Pull Requests>500
Python script for scraping metadata
Run the python scripts
List of projects in C, C++, Python, Java
Download the source code of the projects
Metadata of issues and pull requests
Source code of Github projects
BuGL

Figure 3: Methodology for curating the dataset

Metadata along with source code files of the Github projects constitutes BuGL.

Folder structure of the dataset:

The folder structure of BuGL:

```
\BuGL
    \C
        \JSON
            Repository_name1.json
            Repository_name2.json
            ……….
        \XLSX
            Repository_name1.xlsx
            Repository_name2.xlsx
            ……….
        \Projects
            Repository_name1-master.zip
            Repository_name2-master.zip
            ……….
    \C++\
    \Java\
    \Python\
    \BuGL.xlsx
```

Note: All the paths are with respect to the root directory.

<u>Scripts</u>:

Prerequisites:

1. Python3
2. Chrome driver
3. Selenium

Dataset.json: This is a json file with

- Key as name of the repository
- Value as Github link of the project (Note: Github link should end with '\'. Otherwise it will show error)

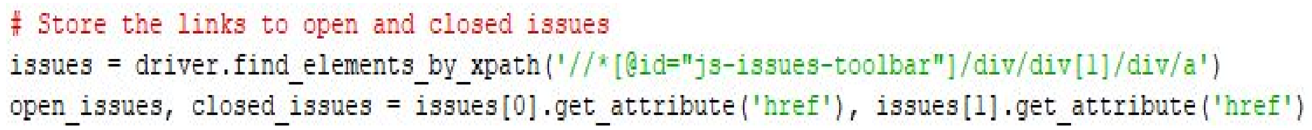
DataExtraction.py: Python script is written to extract the metadata of the issues and pull requests.
Changes need to be made in the python script for extracting the metadata:

- Path to the chrome driver needs to be updated to the location in your system.
- Dataset.json files need to be updated with the respective github repository links and to be present in the same folder as DataExtraction.py

Note: Python scraping file is based on the CSS selectors like classname, id, xpath of the Github page. These selectors might change when the Github page is updated.

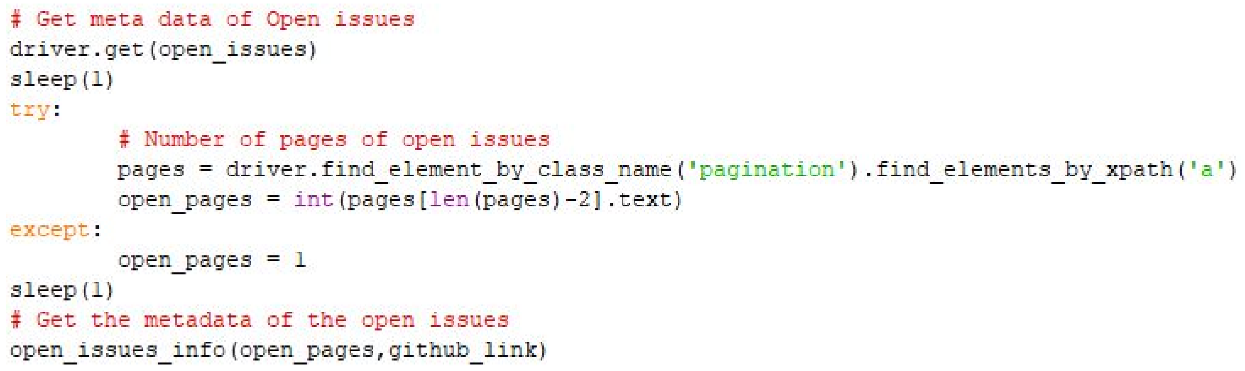
Working of the scraping file (DataExtraction.py):

Step 1: Opens the GitHub page and stores the links of open issues and closed issues.



```
# Store the links to open and closed issues
issues = driver.find_elements_by_xpath('//*[@id="js-issues-toolbar"]/div/div[1]/div/a')
open_issues, closed_issues = issues[0].get_attribute('href'), issues[1].get_attribute('href')
```

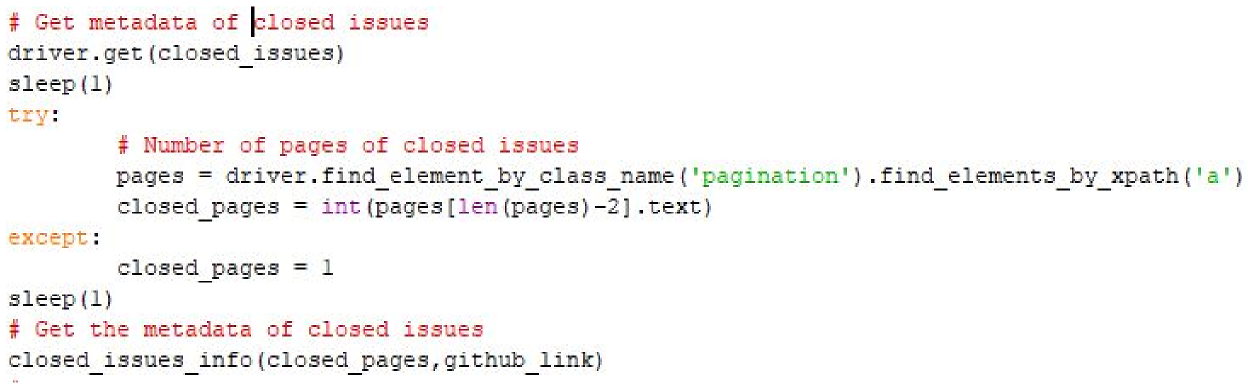
Step 2: Opens the open issues page and traverses through all open issues and stores the required metadata from them in json format.



```
# Get meta data of Open issues
driver.get(open_issues)
sleep(1)
try:
        # Number of pages of open issues
        pages = driver.find_element_by_class_name('pagination').find_elements_by_xpath('a')
        open_pages = int(pages[len(pages)-2].text)
except:
        open_pages = 1
sleep(1)
# Get the metadata of the open issues
open_issues_info(open_pages,github_link)
```

open_issues_info -> function that traverses through all the open issues to get the metadata.

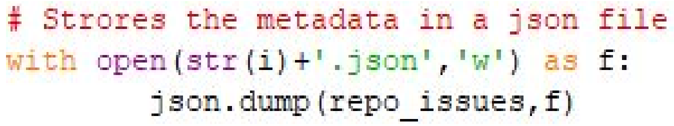
Step 3: Opens the closed issues page and traverses through all closed issues and stores the required metadata (also from the pull request if the issue is closed by merging a pull request) from them in json format.



```
# Get metadata of closed issues
driver.get(closed_issues)
sleep(1)
try:
        # Number of pages of closed issues
        pages = driver.find_element_by_class_name('pagination').find_elements_by_xpath('a')
        closed_pages = int(pages[len(pages)-2].text)
except:
        closed_pages = 1
sleep(1)
# Get the metadata of closed issues
closed_issues_info(closed_pages,github_link)
```

closed_issues_info -> function that traverses through all the closed issues to get the metadata of issues as well as the pull request (if any) that closed the issue.

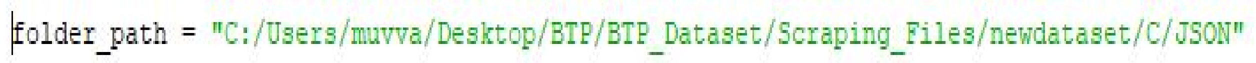
Step 4: Stores the output in the json format with the name of the file as the name of the repository.



```
# Strores the metadata in a json file
with open(str(i)+'.json','w') as f:
        json.dump(repo_issues,f)
```

For the open issues, fields related to pull requests will not be there. For closed issues, fields related to pull requests will be there and they will be either empty (if the issue is not closed by pull request) or filled with some details (otherwise).

jsontoxlsx.py: This file is used to convert the **json** file with the metadata information of issues to **xlsx.** Path to the folder where json files are present need to be changed for the script to work.



```
folder_path = "C:/Users/muvva/Desktop/BTP/BTP_Dataset/Scraping_Files/newdataset/C/JSON"
```

Output:

- xlsx file with the name as name of the json file
- File wise number of issues closed by pull requests
- Total number of issues closed by pull requests

Language	No. of projects	Closed pull requests	Closed Issues	Open Issues	Issues closed by pull requests
C	21	51408	36617	8608	2462
C++	11	37198	30227	3607	2222
Python	12	32454	39760	6767	2626
Java	10	47258	44557	4210	2877

Table 1: Language wise statistics of BuGL