

Software Analytics

Bug Triaging

Joy Albertini, Jacob Salvi, Adam Kasperski, Anuj Kumar

Data collection

Issues collections

Initially we tried to collect the issues using the 'pygithub' library.

We set up the 'github' object, `Github(auth=auth, per_page=100)`, to fetch 100 issues per page and then tried to get the issues by iterating over the 'PaginatedList' returned by the `repo.get_issues(state="closed")`.

This method revealed to be quite slow since a much larger number of requests than expected were performed. With 220000 issue and 100 issues per page a mere 2200 requests should have sufficed, avoiding the 5000 request per hour limit github has in place.

Instead a much larger number of requests were performed and we became subject to the timeout put in place by github.

To solve this problems we have completely ditched the 'pygithub' library, opting to make the request manually.

```
1 base_url = "https://api.github.com/repos/microsoft/vscode"
2 url = f"{base_url}/issues?state=closed&page={current_page}&per_page=100&\
3 direction=asc"
4 response = requests.request("GET", url, headers=headers, data=payload)
```

By fetching the issues in ascending order we should get issues with issues number increasing from number 1 onward. We limited our search to issues in the closed state, as can be seen from the previous code snippet, and we only kept the issues with exactly one assignee, as shown in the following snippet.

```
1 issues_to_keep = [issue for issue in issues if issue["number"]
2 <= max_issue_number and len(issue["assignees"]) == 1]
```

That said, it seems that the github api itself has a small bug. Page 1921 had an out of place issue.

This page contained issues numbered as [205001, 205002, 205003, 230000, 205004, ...], the obvious outlier was causing issues by triggering an early termination.

We saved all of the issues data as received in a json file to be used for the pre-processing. Having all the raw data allowed us to study which part of the data was useful without having to fetch it again.

We also saved the number of commits per user, but limiting our search to the commits in the 'main' branch.

Commits collections

We extracted commits from the **main branch** using the GitHub API. Attempting to extract commits from **all branches** proved challenging due to the high number of commits and GitHub API rate limits. To overcome this, we utilized the `git log` command to retrieve all commits locally. However, mapping these commits to usernames was difficult because commits lacked author names or IDs. We considered fetching usernames by commit SHA through the API, but the rate limits again restricted this method. Consequently, extracting commits from all branches was limited by API constraints and incomplete author information.

Pre processing

The data had to be pre-processed before being given to the model for training. Of all the information we retrieved in the previous step, we decided to keep only the 'title', 'body', 'id', 'number', 'url', 'assignee', and the 'labels'. The title and body required the most preprocessing, given that they contain the bulk of the text. From the raw data we obtained, we first removed pull requests. Given that the bodies of the issues are written in Markdown, we have used a Markdown library to parse them. Some elements, such as HTML tags, links, and images have been removed during the parsing. The code blocks are kept exactly as is. The main parts that have been modified then are headings, paragraphs, and lists.

All text was converted to lowercase, and contractions were expanded to standardize word forms. Emojis and special characters were removed to decrease noise in the data, and any residual URLs and email addresses were eliminated. Mentions and hashtags were removed as well. We also normalized whitespace by collapsing multiple spaces into a single space, and we removed the diacritics. The text was then tokenized, punctuation was removed from tokens, with the exception of keeping the octothorpe in the word 'C#' and 'F#', since these are programming language names, and module names. Tokens longer than 40 characters were excluded. We removed common English stopwords and lemmatized the remaining words to their base forms. We used lemmatization as we thought

that reducing the word to its base form instead of stemming it would preserve more semantic meaning and yield better results in subsequent steps.

Predictor

Even though we kept seven columns during preprocessing we actually used only four for training. These four are the 'title', 'body', 'labels' and 'assignee'. The reasoning behind using the 'title' and 'body' is pretty straightforward, these two columns contain the most information about the issue and are thus of the utter most importance.

We need to pass the 'assignee' in order to predict future assignees, we only used the assignee 'id' since it being a number didn't require any particular encoding for the training and could easily be mapped back to an assignee name. After a few attempts, interleaved with improvement of the preprocessing, we decided to include the issues labels in the corpus used to train the model. Our reasoning is based on the fact that an user is likely to work on similar issues, and therefore be assigned issues with similar labels.

That said, particularly new issues might not have had any label assigned to them yet.

In the first iteration we made use of logistic regression for the bug classification which led to an accuracy of 30% . Upon further reading we realized that logistic regression struggles with high dimensional embeddings provided by Roberta further it also does not handle well overlapping boundaries like assignees solving similar bugs and leads to incorrect classification. These aspects limited the improvements that could be made to improve accuracy.

Finally, we decided to use the 'RobertaForSequenceClassification' as our base model for the training. This was a much better choice as it overcomes the limitations of logistic regression which lead to a much better accuracy for the model.

Results

All issues

With all issues included in the training corpus the model took multiple hours to train and we achieved an accuracy of roughly **61%**.

Recent issues

Limiting our training to recent issues, the model was significantly faster to train and we achieved an accuracy of roughly **55%**.

Interface

We included a simple terminal-based interface to query the top five most likely assignee given an issue number.

```
Models loaded from /Users/jacobsalvi/Master/3_semester/SoftwareAnalyt
BugTriaging
Insert an issue number:
    200
User: hedi number of commits: 1882
User: chrisdias number of commits: 88
User: sandy081 number of commits: 7915
User: jrieken number of commits: 11650
User: brett cannon number of commits: 4

Submit
Clean
Exit
```

To run it is sufficient to run the following code, given that the python environment is set up as described in the README.

```
1 python3 src/tui/tui.py
```

Conclusion

In the project we had the chance of developing a tool for automated bug triaging leveraging using machine learning.

Developing this tool we came to appreciate both the strength of such an approach to solve this kind of problem and the challenges faced in developing it.

We also appreciated doing this kind of work in a quite realistic context, given that the vscode repository was used, and can picture many practical applications of similar techniques to solve similar challenges in industry.

It is also our opinion that we achieved satisfactory results in terms of accuracy.