Analysing commit messages

June 14, 2021

1 Introduction

With the rise of open source software, more and more big corporations incorporate free software in their stack. This also means that the amount of meaningful software that is available online on platforms like GitHub [1], is ever increasing. GitHub, as the name already suggest, offers the ability to host Git repositories. Git is a distributed version control system [2]. One of the most important ideas when working with Git is writing expressive commit message for code changes. This helps others to understand what the changes are intended to do or why they are needed. Using automatic labelling for commits based on their message would therefore help users to quickly understand the intended basic operation without reading the message of the commit. However developing such approaches require a representative dataset of GitHub commits.

In this paper an effort has been made to construct a new dataset containing commit messages of various repositories from many popular languages. Therefore repositories of the top 10 most wanted programming language according to [3] have been investigated on GitHub. To get a more accurate representation of each language, 100000 commits have been chosen. To further enhance diversity, only the last 10000 commits of each selected repository were used, while also a general restriction has been enforced to limit the amount of commits from a single user in the set. A Python script has been developed to automatically construct the dataset.

Afterwards the dataset has been analysed and different metrics are compared and discussed regarding its contents. To check the suitability of the set, a basic commit classification approach has been implemented and validated suing 10-Fold cross validation. A subset with already labelled commits has been extracted and modified for testing the developed approach.

The next section discusses some related work about commit messages in general and their analysis in correlation with metrics to infer the correct meaning of the commits changes resulting in the correct application of a label 2. Afterwards, the methodology of the construction of the dataset and the classification approach is presented 3. In section 4 the implementation of the data scraping script, as well as the data analysis script is discussed. Some challenges discovered during implementation and planning of the methodology are discussed in

section 6. This is followed by the results of the analysis and the classification approach r5 and the conclusion of the work with suggestions for future work 7.

2 Related Work

The idea of classifying commits is based on the concept of labelling messages with a specific tag. Already labelled commit messages can be seen as correctly labelled, as the author is unlikely to label his commit incorrectly. Furthermore we have many authors for our selection of messages essentially giving us a collaborative labelling approach, which has already been well researched to result in a representative data set [4].

While this study only focuses on the lexicographical analysis of the commit message, we can also infer a lot of information about the meaning of a commit through other factors like code changes. From this we can conclude if a commit only fixes some mistakes or introduces completely new features by analysing its size [5].

We can also use commit messages to infer information about the changes themselves, which can be especially useful when doing version control analysis. Automatically tagging and understanding the idea or reason behind a commit can reduce the effort of others working with the same codebase. Correct labelling can reduce the effort of filtering commits relevant to a certain part of the codebase or make it easier for developers to find relevant changes by other contributors [6]. An example would be to check if a commit introduces a fix for an issue. For some repositories it was determined that the amount of changes directly contributes to the likelihood of being a fix, [7]. However, this might not be true for all codebases in general.

While gathering information using all metrics of a commit can yield good results, only analysing the message itself is less trivial. Using many different repositories for commit resources also greatly increases diversity, which can ultimately lead to lower accuracy of a predictive model [6, 8].

The most important factor in commit classification is the selection of labels, as they can greatly influence the prediction results. However, applying to many labels might result in a very complex classification problem, where commits cannot be properly fitted to a single category ultimately resulting in wrong predictions [8].

3 Methodology

As already alluded to in section 1 the goal of the project is to analyse commit messages. To achieve this repositories of the top 10 most wanted languages were used. This list was used as a reference, since it has a great assortment of languages that are in high demand and also relevant in today's development climate. The languages in question are:

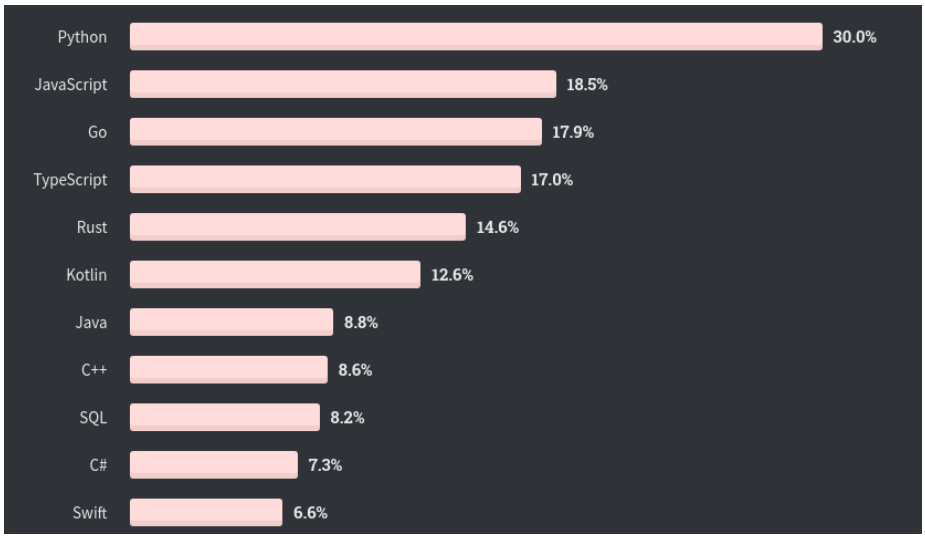

Figure 1: top 10 most wanted languages [3]

It has to be noted that not all languages depicted in Figure 1 have been chosen since SQL is not suited for analysis and therefore Swift has been chosen instead. Regardless of the percentages seen in Figure 1, 100000 commits of each language have been considered for analysis, where 10000 is the maximum per repository. To further diversify the results a limit has been imposed regarding the amount of commits per author. This limit has been set to 100, since commit messages by the same author are very likely to use a similar order of words, even across different repositories.

The language of choice for the programming part of the analysis is Python. Python is well suited for this project as it has rich support for natural language processing related tasks. Additionally the language is great for quick prototyping of ideas and with the help of tools like Jupyter Notebook, documents with graphs and text can be created in no time. Especially during the scraping process the library PyGithub was put to good use. PyGithub is a Python library that offers typed interactions with the GitHub API v3 [9]. The results of the fetching process are furthermore cached in one CSV file per language.

4 Implementation

The implementation itself is comprised of three parts:

- scraping
- tagging
- training/testing

4.1 Scraping

The scraping process is a curious case since it was problematic to some extend. More in that regard can be found in section 6. The basic idea of the scraping process was to fetch repositories and their commits from GitHub. This was done with the help of PyGithub. The code itself for the process of repository gathering is rather simple. A basic array with all languages was defined as seen in Listing 1, which is then used to iterate over and actually search for repositories for the corresponding language with the specific GitHub query syntax as seen in Listing 2.

Listing 1: Array of all languages for scraping

```
languages = [
  'python',
  'javascript',
  'go',
  'typescript',
  'rust',
  'kotlin',
  'java',
  'c++',
  'c#',
  'swift',
]
```

Listing 2: GitHub query syntax

```
g.search_repositories(query=f'language:{language}', sort='stars
    ', order='desc'):
```

This then yields all repositories sorted by *stars* in descending order. Stars is a metric on GitHub to show a repository's significance. Each star represents a user on GitHub who has starred this repository. In the code `itertools.islice (commit_iter, 10000)` limits a repository to 10000 commits. A global counter keeps track of all the commits processed and if the 100000th commit is reached, repositories for the next language will be fetched. This means that at least ten repositories per language are considered. Because the process of scraping is time and space intensive the results are cached. Per language one CSV file is created. The file consists of rows with four columns:

- repository
- language
- author
- message

With this structure all important information is included in the file which is essential for further processing. An example of one file can be seen in Listing 3.

Listing 3: Excerpt of *typescript.csv* with commits from the repository *microsoft-/vscode*

```
repository,language,author,message
microsoft/vscode,typescript,roblourens@gmail.com,"Fix #126087"
microsoft/vscode,typescript,penn.lv@gmail.com,":notebook:
    differenciate editor focus and list view focus"
microsoft/vscode,typescript,me@tylerleonhardt.com,"always hide
    quickinput on iPad when focus is lost fixes #125284"
microsoft/vscode,typescript,connor@peet.io,"fix: use inline
    sourcemaps in watch task"
microsoft/vscode,typescript,alexdima@microsoft.com,"update `
    monaco.d.ts`"
```

4.2 Tagging

The next step is to tag commits. We distinguish between two categories of tagged commits:

- Conventional commits
- Gitmoji commits

A specific regular expression for each category has to be matched in order to tag the commit. The lookup in case of a match is done by searching for the specific tag in a dictionary. Each category also has a specific dictionary. In case of a conventional commit Listing 5 will be searched, Gitmoji commits Listing 4 otherwise.

Listing 4: Dictionary for Gitmoji mappings

```
gitmoji_mappings = {
  ':memo:':              'docs',        # Documentation
  ':zap:':               'perf',        # Performance
  ':fire:':              'remove',      # Removal
  ':sparkles:':          'feat',        # Feature
  ':bug:':               'fix',         # Bug Fix
  ':recycle:':           'refactor',    # Refactor Code
  ...
}
```

Listing 5: Dictionary for conventional tag mappings

```
tag_mappings = {
  'bug':           'fix',
  'testing':       'test',
  'documentation': 'docs',
  'feature':       'feat',
  'gui':           'ui',
  ...
}
```

4.3 Training & Testing

In order to properly train and test the data-set some filtering has to be done. With distinct regular expressions certain message patterns are filtered from the commit message. With `#\d+` pull request number or issue number will be removed, since they start with a # and end with a number. Version numbers can also excerbate tokenisation and therefore the simple regular expression `\b\d+(\.\d+)+\b` takes care of basic version notations that are common on GitHub. Email addresses and URL in general also make tokenisation more difficult. Regular expressions for both of those patterns are also necessary. Once all patterns are taken care of, tokenisation can commence.

The next step is then to encode the labels. This can be done with the *LabelEncoder* from *sklearn*. The *sklearn* library also offers a *CountVectorizer* with whom text documents can be converted into token counts. The actual training / testing process is done with a ten-fold cross validation which splits the data and yields train and test indices with whom the train and test data can be defined. Both data sets are then used within the logistic regression to get the final results.

5 Results

For evaluating our dataset generation, we first collected commits without any limits imposed on commits from a single author. We later added a per-author limit in order to avoid bias being introduced by including a vast amount of commits from a single author.

Figure 2 shows that around 25000 commits (2.5%) out of a total of 1000000 commits are tagged when limiting the number of commits per author to 100. Without this limit, the number of tagged commits slightly increases to about 35000 commits (3.5%) out of 1000000. Overall, we see that with or without limit, the number of tagged commits remains relatively small compared to the total number of commits.

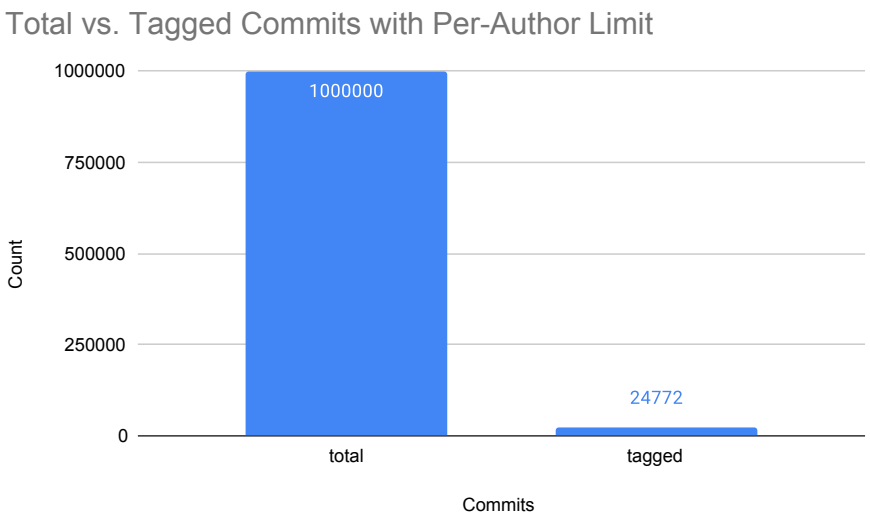

Figure 2: Total vs. Tagged Commits with Per-Author Limit

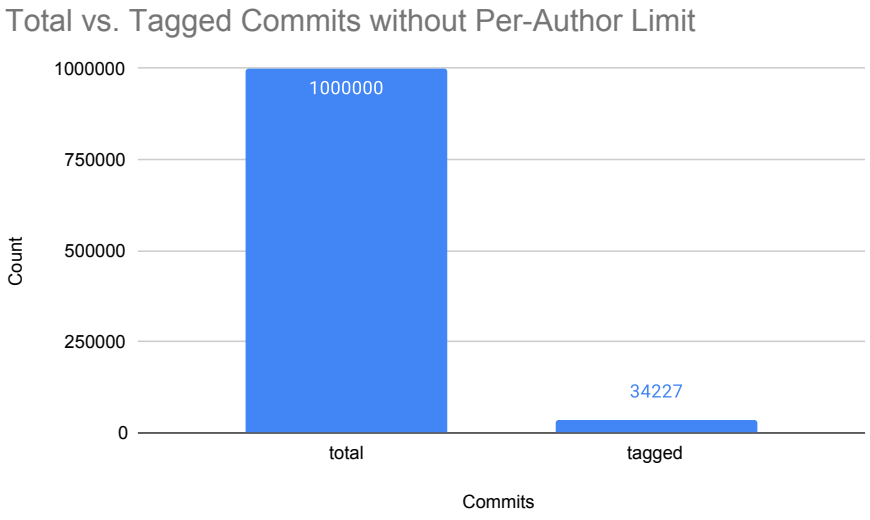

Figure 3: Total vs. Tagged Commits without Per-Author Limit

In Figure 4 and Figure 5, we can see the distribution of commit tags with and without the mentioned per-author limit, respectively. When comparing both graphs, we can see that without a per-author commit limit, the number of

commits tagged with “chore” almost doubles while commits tagged with “feat” stay more or less the same. This is likely due to commits being authored by bots, which mainly applies to automatic maintenance tasks, i.e. chores.

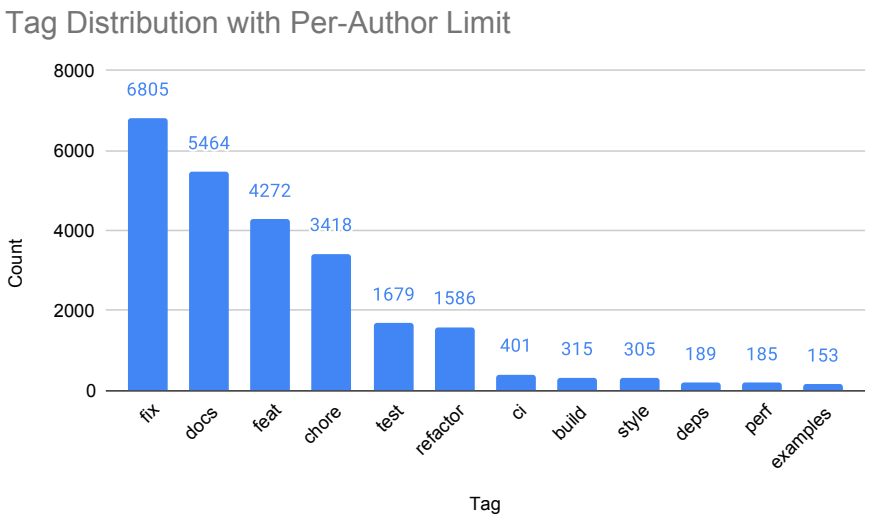

Figure 4: Tag Distribution with Per-Author Limit

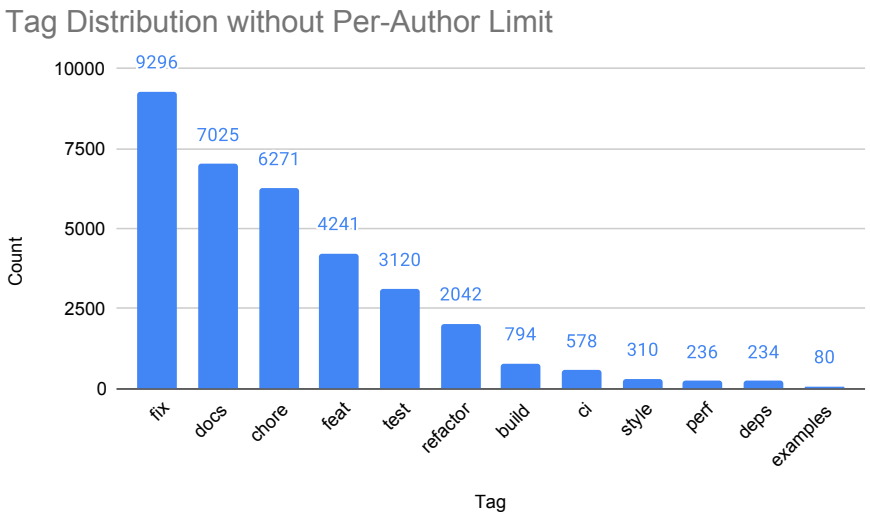

Figure 5: Tag Distribution without Per-Author Limit

Unlimited commits per author:

Total Accuracy: 0.4543248613056252 Total F1 micro: 0.4543248613056252 Total F1 macro: 0.24588792995091063

100 commits per author:

Total Accuracy: 0.47098819013975735 Total F1 micro: 0.47098819013975735 Total F1 macro: 0.23656990457552882

6 Challenges

Unfortunately the project could not be completed without its fair share of challenges. Starting off with the scraping process which was not as straight forward as it originally might have seemed. Since all repositories are from GitHub it was an obvious choice to use the GitHub API, in this case PyGithub, for everything regarding scraping. After some testing however it was clear that this is not a feasible strategy as the GitHub API imposes a strict request limit of 5000 requests per hour. Though we can use a single request to get a repository, getting the messages for all commits requires a separate API request for each commit. Since we have to go over each commit in order to get the commit message, 5000 requests are reached in no time with the goal of 100000 commit messages per language.

Ultimately rethinking our strategy was the only option. This did not mean that we had to abandon PyGithub completely as we still use it to query the repositories. Gathering the commits was only really possible by actually cloning the repo and going over the commits with the Git command line utility itself. Usually this is obviously a reasonable way of fetching commits, however with the amount of repositories we are working with, this was rather cumbersome, since the disk space requirements were considerable. Even more so once the limits on commits per author was in place. This is the result of a few authors having significantly more contributions to a repository than other authors of the same repository. However this is to be expected as open source project maintainers often work in a smaller group for a long time duration. Furthermore each scrape run was also time intensive considering the fact that each repository had to be cloned and processed.

Another challenge was the selection of repositories itself. In the end we settled on sorting them by stars. However this is not necessary a proper metric to determine popularity. Since every user can star an unlimited amount of repositories, the possibility of stars contributed by fake accounts cannot be excluded. Furthermore users might star a repository for different reasons like, appreciation or bookmarking. It would probably be more accurate to use a combination of stars, forks and watchers. Our goal however is to collect a set of commits, which is representative of the whole set of commits on GitHub. Since we have a great variety of repositories for each language, we can accept a weaker indication of popularity.

7 Conclusion