Commit Message Tagging

Markus Reiter, Lukas Dötlinger & Michael Kaltschmid

June 15, 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 messages 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 requires a representative dataset of Git 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 languages, according to a StackOverflow survey [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 author 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 using 10-fold cross validation. A subset of automatically labelled commits was extracted and modified for training and 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 commit's changes, resulting in the correct application of a label. Afterwards, the methodology of the construction of the dataset and the classification approach is presented in section 3. In section 4, the implementation of the data scraping script, as well as that of 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 in section 5 and the conclusion of the work with suggestions for future work in section 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 their 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 dataset [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 too 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 collect and analyse commit messages. To achieve this, repositories of the top 10 most wanted languages, according to a yearly StackOverflow survey (seen in Figure 1), 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:

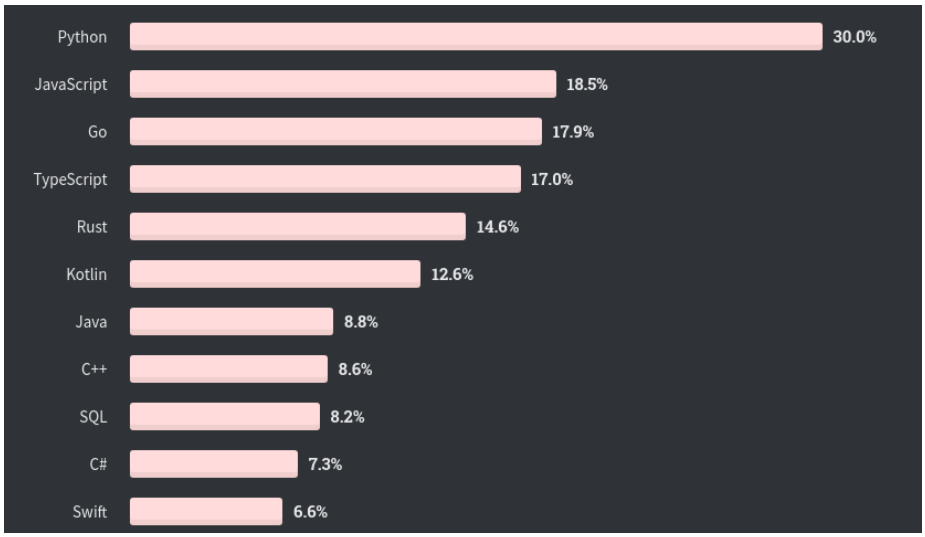

Figure 1: Top 10 Most Wanted Languages [3]

It has to be noted that not all languages depicted in Figure 1 have been chosen since SQL is a query language and not considered a general programming language. We therefore excluded SQL since it is not suited to be used as the main language for a repository. Swift has been included as the 10th language instead. Regardless of the percentages seen in Figure 1, 100000 commits of each language have been considered for the final set, 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 authors are distinguished by their e-mail addresses.

The language of choice for the programming part of the dataset retrieval 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 with little effort. 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. As those files used the same format and headers, they can easily be combined into one large file for convenience. However, the language is also included as an additional column in order to not lose this information.

The dataset was analysed using the NLTK library [10] and Python. The library was used to generate different word distributions and process the messages

themselves.

Since some repositories contain commits that are already tagged with some label, regular expressions were used to filter a subset of those commits. This subset was used to test a basic classification approach. The idea is to determine how good a simple approach, only based on text analysis, actually is.

For the classification approach, the logistic regression implementation from the Python module sklearn [11] was used. A subset of all extracted tags were used to generate a common set of labels. The feature vectors for the logistic regression were created by tokenising the commit messages. The validation of the classifier was done using 10-fold cross-validation.

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 extent. 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 popularity. Each star represents a user on GitHub who has starred this repository. Counters keep track of per-author (100), per-repository (10000) and global (1000000) commit limits. This means that at least 10 repositories per language are included. When the 100000th commit is reached, repositories for the next language will be fetched. 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 excerpt 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 [12]
- Gitmoji [13]

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. Gitmoji commits are mapped to follow the same naming as Conventional Commits using the lookup table seen in Listing 4. Commits which follow the Conventional Commits format are mapped using Listing 5. In the end, all commits which are tagged following the Angular Commit Message Guidelines [14] are treated as "known tags" and filtered. In addition to adding a tag to a commit, the tag is also removed from the original commit message.

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 dataset, some filtering has to be done. With distinct regular expressions common message patterns are filtered from the commit message. With `#\d+`, pull request or issue numbers are normalised, since they start with a # and end with a number. Version numbers can also exacerbate 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 URLs in general also make tokenisation more difficult. Therefore, both of those patterns are also removed.

The next step is then to encode the tags. This can be done with the *LabelEncoder* from *sklearn*. The *sklearn* library also offers a *CountVectorizer* with which text documents can be converted into token counts. The actual training/testing process is done using a 10-fold cross validation which splits the data and yields train and test indices with which the train and test data can be defined. Both datasets 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, which is shown in Figure 3. Overall, we see that with or without limit, the number of tagged commits remains relatively small compared to the total number of commits.

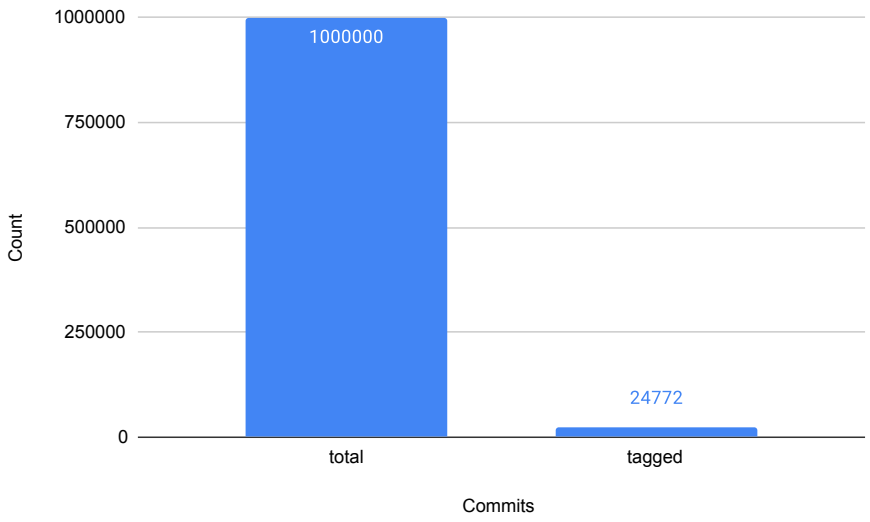

Figure 2: Total vs. Tagged Commits with Per-Author Limit

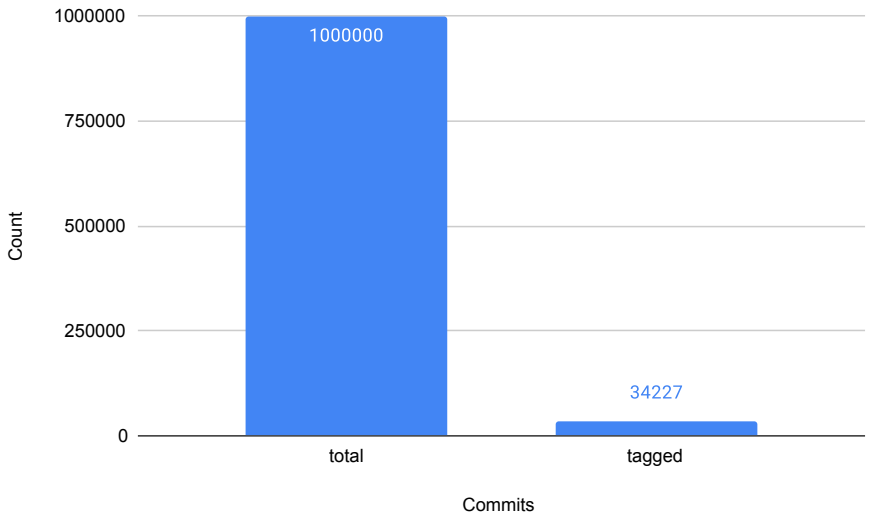

Figure 3: Total vs. Tagged Commits without Per-Author Limit

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.

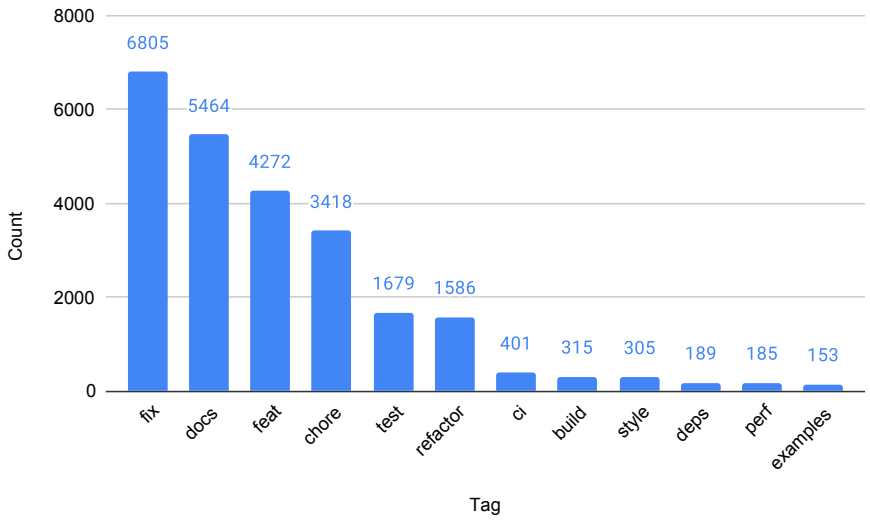

Figure 4: Tag Distribution with Per-Author Limit

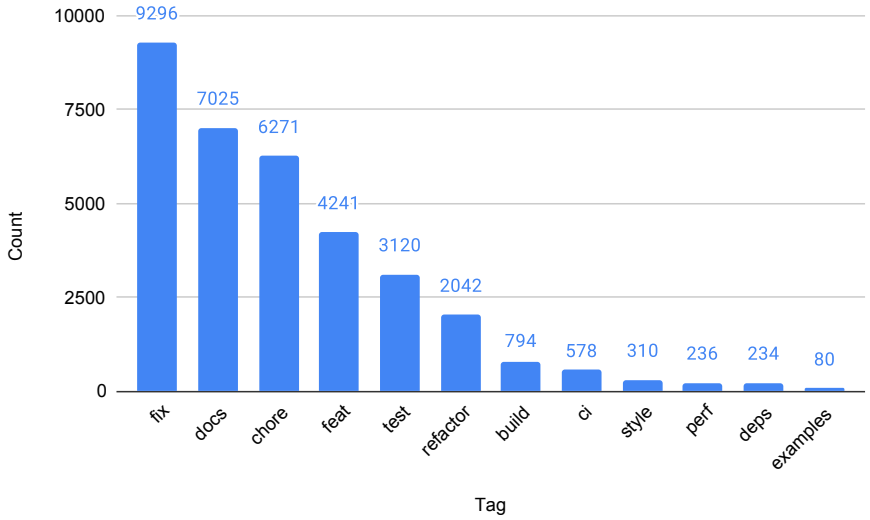

Figure 5: Tag Distribution without Per-Author Limit

When comparing the results of the logistic regression, we can clearly notice a difference between the two sets. With unlimited messages per author we get about 4.15% improvement over the set with author limitations. This can be

related to the bias introduced by authors contributing a very high amount of commit messages. We can conclude from the distribution of labels that those authors primarily are maintenance bots.

Looking at the F1-score macro, we can further conclude that the high increase compared to the set with author limits is due to those extra commits. As previously shown, the additional commits contain a high amount of "chore" tags, which directly relates them to bots. The uniformity of bot commits increases the likelihood of correct classification, resulting in high accuracy for those labels. As F1-macro does not consider the proportion for each label in the dataset, we get a higher score for the set without author limitations.

The results however are very promising overall. When using the author limit, we are basically working with a very balanced set in terms of commit message diversity while still getting acceptable results with an accuracy of 61.02% and an F1-score micro of 61.02%. The lower F1-score macro of 44.79% is more likely to be a result of insufficient training data, as half of the labels only have a small number of entries in the data set. The exact results, with and without per-author limit, are displayed in Table 1 and Table 2, respectively.

Accuracy	61.02%
F1-score micro	61.02%
F1-score macro	44.79%

Table 1: 100 commit messages per author

Accuracy	65.17%
F1-score micro	65.17%
F1-score macro	53.71%

Table 2: Unlimited commit messages per author

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 straightforward 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 via 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 repository and going over the commits with the Git command line utility itself. Usually this is 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. Once the limits on commits per author was in place, we had to clone 120GB worth of repositories. This is due to 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 scraping 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 necessarily 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

In this paper we presented an approach to construct a new dataset containing commit messages of various repositories from many popular languages on GitHub. In order to complete the project, we first had to scrape repositories, followed by tagging commits and finally using the data to test and train a logistic regression model. While most of the development process was straightforward and as expected, some problems arose especially during the scraping part. API limits are among the worst offenders in that regard. Fortunately we could prevail by circumventing those barriers and find a different approach.

The results are interesting because of the differentiation between per-author limit and no per-author limit. Naturally, once such a limit is in place the amount of high quality commit messages decreases and best practices are not followed in some cases. The report reflects that notion as the amount of tagged commits decreased from about 35000 to about 25000 from a total of 1000000 commit messages once the limit was in place. The actual tag distribution however followed roughly the same trajectory, lead by “fix” which is not really surprising considering the fact that most commits are bug fixes. Looking at the accuracy, F1 micro and F1 macro numbers, the contrast between author limits can be seen in Table 2 and Table 1 as well, as all the metrics are lower with the limit. Besides that fact, the numbers are not unusual and about what we expected

initially.

For future work the criteria could be changed. Focusing on repositories with tagged commits and in general fewer small repositories could yield better results. Furthermore, focusing on specific languages with a different importance metric when choosing codebases, could also be interesting.

References

[1] *GitHub: Where the world builds software*. URL: https://github.com/ (visited on 06/12/2021).

[2] *Git*. URL: https://git-scm.com/ (visited on 06/12/2021).

[3] *2020 Developer Survey*. URL: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted (visited on 06/12/2021).

[4] Scott A. Golder and Bernardo A. Huberman. "Usage patterns of collaborative tagging systems". In: *Journal of Information Science* 32.2 (2006), pp. 198–208. DOI: 10.1177/0165551506062337. URL: https://doi.org/10.1177/0165551506062337.

[5] Abram Hindle, Daniel M. German, and Ric Holt. "What Do Large Commits Tell Us? A Taxonomical Study of Large Commits". In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR '08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 99–108. ISBN: 9781605580241. DOI: 10.1145/1370750.1370773. URL: https://doi.org/10.1145/1370750.1370773.

[6] Audris Mockus and Lawrence Votta. "Identifying Reasons for Software Changes using Historic Databases." In: Jan. 2000, pp. 120–130. DOI: 10.1109/ICSM.2000.883028.

[7] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. "When Do Changes Induce Fixes?" In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083147. URL: https://doi.org/10.1145/1082983.1083147.

[8] G. E. D. Santos and Eduardo Figueiredo. "Commit Classification using Natural Language Processing: Experiments over Labeled Datasets". In: *CIbSE*. 2020.

[9] *PyGithub*. URL: https://github.com/PyGithub/PyGithub (visited on 06/14/2021).

[10] *Natural Language Toolkit*. URL: https://www.nltk.org/ (visited on 06/12/2021).

[11] *scikit-learn: Machine Learning in Python*. URL: https://scikit-learn.org/stable/ (visited on 06/12/2021).

[12] *Conventional Commits*. URL: https://www.conventionalcommits.org/ (visited on 06/15/2021).

[13] *Gitmoji*. URL: https://gitmoji.dev (visited on 06/15/2021).

[14] *Angular Commit Message Guidelines*. URL: https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#-commit-message-guidelines (visited on 06/15/2021).