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]. In this paper an effort has been made to analyse Git commits and gather information on their intended basic operation by looking at the message of the commit. Therefore repositories of the top 10 most wanted programming language according to [3] have been investigated. To get a more accurate representation of each language, 100000 commits have been chosen. To enhance diversity, only the last 10000 commits of each selected repository were used. The analysis has been done in Python.

2 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:

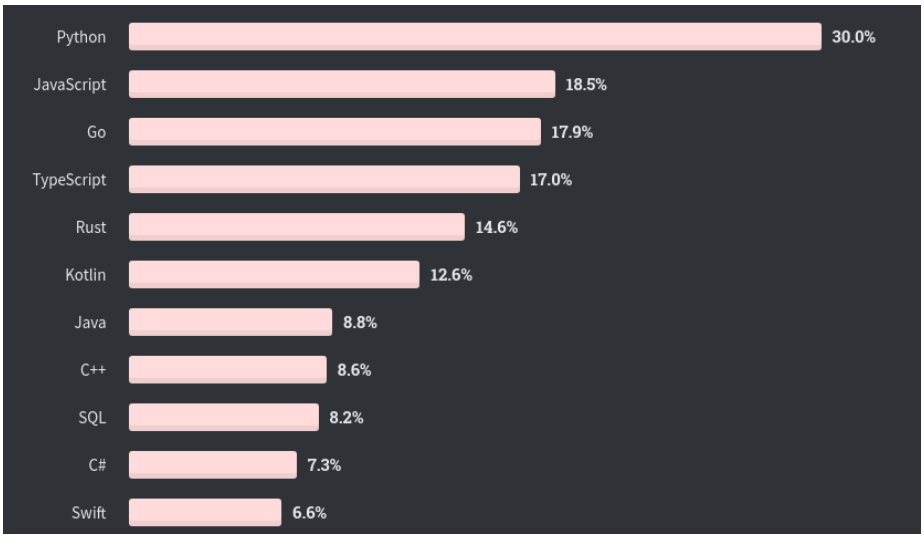

Figure 1: top 10 most wanted languages [3]

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 [4]. The results of the fetching process are furthermore cached in one CSV file per language.

3 Implementation

The implementation itself is comprised of three parts:

- scraping
- tagging
- training/testing

3.1 Scraping

The scraping process is a curious case since it was problematic to some extend. More in that regard can be found in section 4. 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`"
```

3.2 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 Challenges

Unfortuantelly 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 are in this case PyGithub for everything regarding scraping. After some testing however it was clear that this is not a feasable strategy as the GitHub API imposes a strict request limit of 5000 requests per hour and since we have to go over each commit in order to get the commit message, 5000 requests are reached in no time. Ultimately rethinking our strategy was the only option. This did not mean that we had to abonden PyGithub completely as we still used it to query the repositories. Gathering the commits was only really possible by actually cloning the repo and going over the commits with Git 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, since it often tends to be the case that one author has significantly more contributions to a repository than other authors of the same repository. Furthermore each scrape run was also time intensive considering the fact that each repository had to be cloned and processed.

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