Analysis of natural language complexity used by open-source developers

Introduction to Natural Language Processing

Komlichenko Ilya, Krzemiński Piotr, Monicz Kamil,
Piotrowska Weronika, Wojnarowski Marcin

June, 2022

Contents

1 Motivation

In order to improve our NLP skills, we are going to complete simple natural language analysis. The topic of choice should be close to our hearts so we stay engaged throughout the project. In addition, the source data should be freely accessible, so it is easy to work with. And so, our analysis will focus on the natural language complexity used by developers of various programming languages who contribute to open-source platforms (such as GitHub).

We hypothesize that language with a lower barrier of entry (such as Python) will exhibit a lower natural language complexity than languages that expect a higher level of expertise (such as C). Additionally, we are aware of different demographics of programming languages thus expect scientific/research focused languages (such as Julia) programmers will naturally use a more complex natural language to express ideas/problems.

In the end the following set of programming languages was chosen:

1. Python
2. C
3. JavaScript
4. Golang

We believe this subset will represent a wide range of kinds of programmers: Python used by beginners/scientific community, C used by low level developers, JavaScript used by everyone, and Golang used by modern system's programmers. Thus they have a different target demographic so we believe it should show some noticeable differences. Another reason this subset was chosen is the availability of resources: these language are quite popular therefore much data is present for it in GitHub (discussed in greater detail in section 2.2).

2 Data sourcing

The linguistic data will be primarily scraped from the GitHub issues, where developers, using natural language, often describe bugs or feature requests related to some piece of software. And since the code there is publicly available, it is easy to assign a specific programming language to it.

2.1 Text processing

Every issue will be assigned a complexity/readability score based on industry-standard algorithms and bucketed into their respective programming language. For example, one could compute Flesch-Kincaid Grade Level or check the frequency of passive sentences.

2.2 Scraping procedure

To gather the needed data, the scraping procedure was organized into three steps described in the following subsections.

2.2.1 Getting raw data

Using the official GitHub API we first fetch a list of users which we want to examine. To improve the quality of issues which we will be later studying, we made an assumption that GitHub users with more followers in general write issues of higher quality. We intuitively correlate higher followers count with a greater professionalism and higher activity on the platform. For each programming language a list of top 1000 (in terms of amount of followers) users with at least 170 followers[1] was downloaded. An additional reason as to why the previously mentioned subset of languages was chosen is that these (unlike other languages which we had wished to investigate, such as purely scientific programming languages) do have a 1000 users with a reasonable amount of followers. Only personal accounts where taken into account, so organizations were filtered out since they represent more than a single person. A user was considered to be a **P** language programmer if majority of their repositories are written in **P**[2].

Afterwards, once we have a list of users (in total 4000 users) we proceed to fetching issues written by them. Top 100 issues created by a given user were downloaded. Only the body of the posted issue is saved, the thread that follows is ignored. Issues are ranked by the amount of interactions[3] that they have received, thus top 100 issues refer to top 100 most active issues. Here, we make a similar assumption that more active issues correlate with a higher quality of natural language, for instance an RFC with naturally generates more traffic.

Finally, the gathered data is saved to a `csv` file containing the original author, their programming language, and the body of the issue. Since not every user had 100 issues, the final amount of scaped was 233,686 (where the upper bound is 400,000 if every user had 100 issues). We believe this is a very comfortable amount and even if after cleaning we will lose 50%, this still will be enough.

Technical details Scraper was written in the Rust programming language using the octocrab library which wraps the official GitHub API. For all requests GitHub's search API was used, which has an unfortunate limit of 30 requests per minute. In each search call 100 items can be retrieved thus the whole scraping procedure spends $\frac{400000}{100 \cdot 30} = 133.(3)$ minutes on waiting for the limit rate alone. The scraper can be found in section 5.

2.3 Cleaning up the data

TODO

Data scraped from GitHub will be cleaned up by removal of any non-natural language texts such as code blocks. We suspect that the dataset will be still full of true-negatives and will require manual labor to review our samples. Finally, the dataset will simply consist of two dimensions: text (issue content) and a label (programming language).

We want to focus on a specific subset of languages with a fairly balanced distribution between classes, also taking into account the repetition of issue posters.

3 Functionalities

After scrapping the data, we are planning to perform following operations in order to give us a more complete insight

1. Basic text analysis – this includes average length of sentences, average length of words, most common word occurring etc. We are expecting developers of more complex programming languages to use more sophisticated language in the reported issues, this includes less grammar mistakes, longer sentences and words.

2. Frequency of passive sentences – in order to identify passive sentences we will use Part Of Speech tagging (POS). In English, a passive sentence can be defined by the place of the verb in the sentence and the form of this verb. We suspect the data to have high frequency of passive sentences, as it describes development issues.

[1] 170 followers was chosen as the cutoff because it was the argmax such that all four programming languages have at least 1000 users each

[2] Quoting the GitHub documentation: "*`language:javascript` users [...] with a majority of their repositories written in JavaScript.*"

[3] An interaction consists of either a comment or a reaction

3. Flesh-Kincaid Grade Level – by evaluating the Flesh-Kincaid grade level, we can determine whether the text is readable to an average, non-developer person. It measures the length of the sentences and syllables per words, which indicated the easy to read or not. We expect developers of more complex programming languages to write harder to read issue descriptions.

$$0.39\left(\frac{\text{total words}}{\text{total sentences}}\right) + 11.8\left(\frac{\text{total syllables}}{\text{total words}}\right) - 15.59$$

4. Flesh readability score – similarly to Flesh-Kincaid Grade Level, this score is used to measure level of difficulty when reading a text. We compute the score as follows:

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

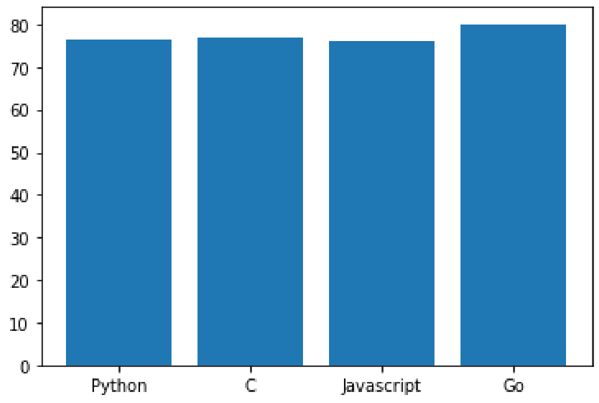
We hope that after obtaining this data, there will be interesting conclusions that there exists dependency between a developer's primary programming language and the natural language he or she uses when describing issues on GitHub.

4 Results

4.1 Basic text analysis

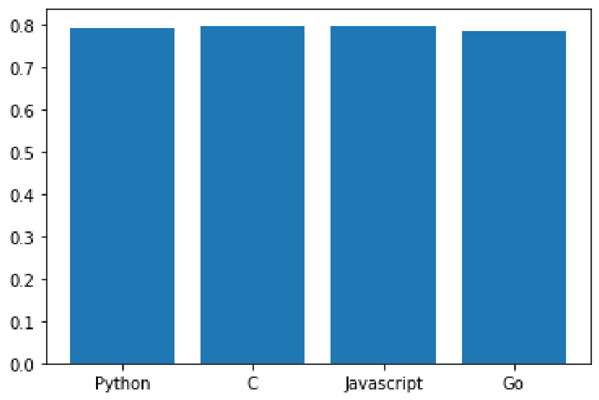
1. Average text length

   This graph presents average length of an issue description. We can observe that this value is similar for every analyzed language.



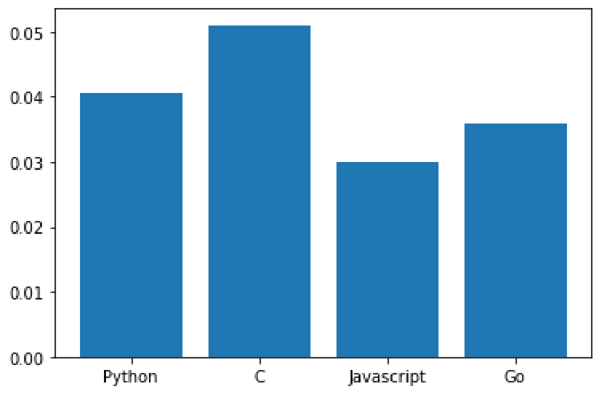
2. Average lexical diversity of a single issue

   This value is the average lexical diversity of a single issue. This value is obtained by dividing the set of tokens over the number of words in a single issue. The value is then averaged for each language. We can observe that this issue is similar among every language.



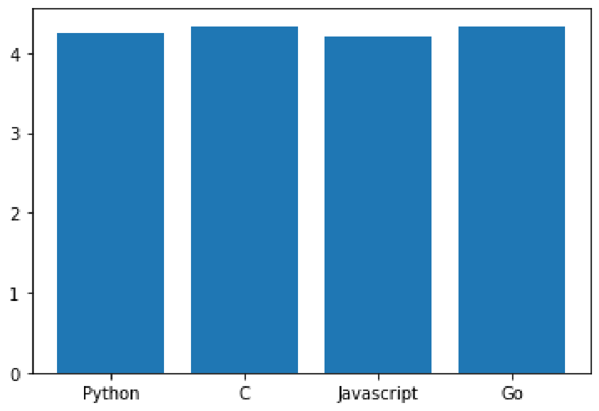
3. Total lexical diversity

   This value is the lexical diversity of all issues among the language. It is obtained by dividing set of tokens over number of all words in all issues. We can see that the lexical diversity is small for every language, which means all of the programmers write similar issue descriptions. However, C programmers tend to use the most diverse descriptions.



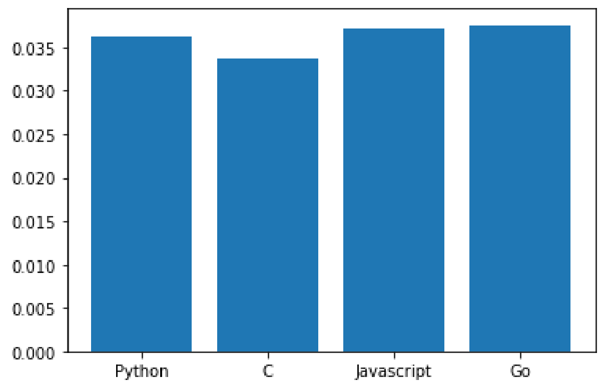
4. Average word length

   This is the average word length for every language. We can observe that this value is similar among all languages.



5. Percentage of words over 10 characters

   Here is the percentage of words over 10 characters used in all issue descriptions among all languages. We can observe that all the programmers use long words with similar frequency.

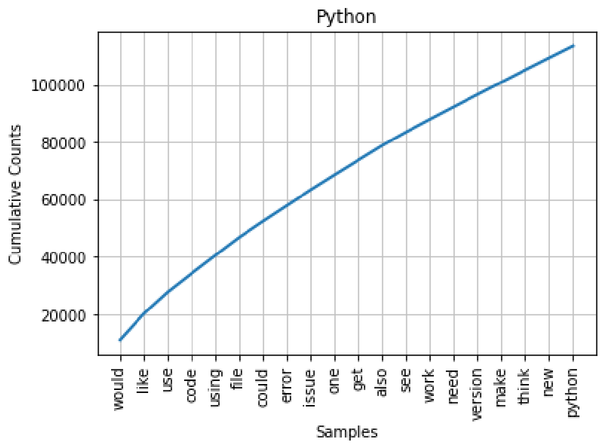


4.2 Frequency distributions

In this section, we present the frequency distribution of tokens in the text. We can see the most common words used, as well as the cumulative value of this words.

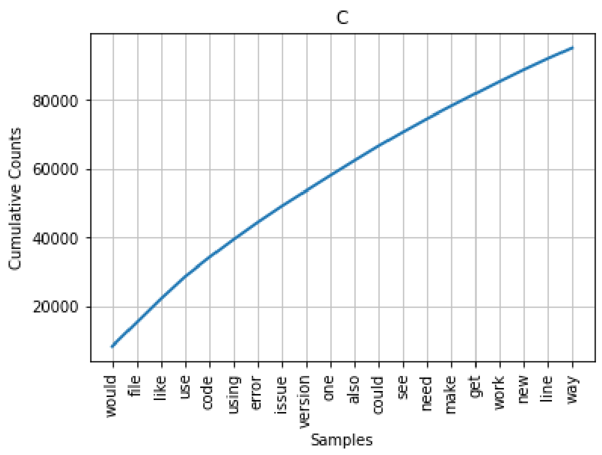
1. Python

   In Python, the most common words are would, like, code, use and using. Given over 2 900 000 tokens in the text, top 20 words make up to 37% of the whole data.



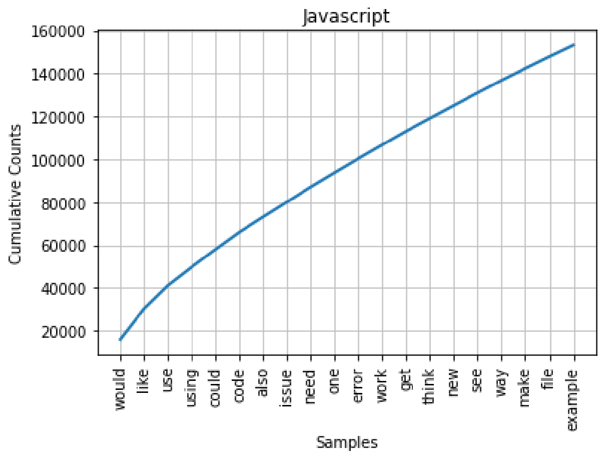
2. C

   In C language, the most common words used are would, file, like, use and code. We can see that, given over 2 600 000 tokens in the text, top 20 words made up to 34% of the total text.



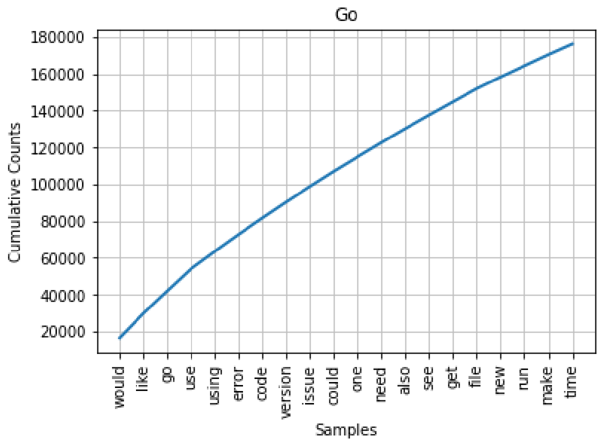
3. JavaScript

   In JavaScript, the most common words are would, like, use, using, could. Given 3 700 000 tokens in the whole text, top 20 words make up to over 40% of the whole data.



4. Golang

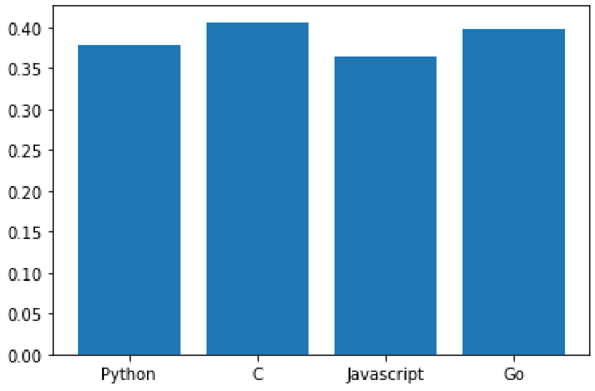
   In Go, most common words are would, like, go, use, using. Given over 4 500 000 tokens, the top 20 make up to 40% of the whole text.



We can observe that among all languages, issue comments contain variations of words would and use. The most informative feature is that C programmers use word "*file*" a lot, probably to low-level file access provided by C. Golang programmers use word "*go*" a lot, which may come from the language name.

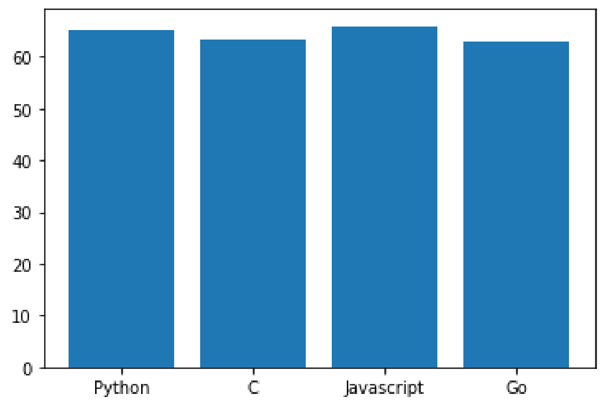
4.3 Frequency of passive sentences

As expected, the frequency of passive sentences is high for all languages (in comparison to 10% in most books), as it describes specific bugs in the code. One can see that C programmers use passive sentences the most, achieving almost 40% of total passive sentences.



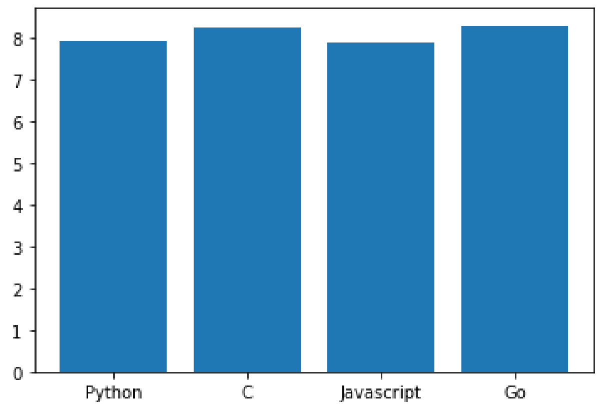
4.4 Flesch readability score

We can see that all of the languages fall into 60-70 interval, which leads them to be classified as easily understood by 13- to 15-year-old students. This score takes into account number of syllables per word and sentence length, which, as established earlier, was not high in most cases.



4.5 Flesch-Kincaid grade level

In this case, all languages scored below 10, which makes them extremely difficult to read, understood only by university graduates. However this metric also takes into account number of words and syllables per sentence, it is much different than the previous Flesch metric. However, in most cases any text has higher Flesch reading score than Flesch-Kincaid grade level.



5 Attachments

1. scraper.zip – Scraper source code