KiTHub

Arriaza Barriga, Romina Carolina - rcab@kth.se
Chalumuri, Vivek - vivekc@kth.se
Cock, Jade - jadec@kth.se
Giorgis, Stavros - giorgis@kth.se

May 2019

Abstract

Reusing existing code can prove to be a precious gain of time. In this project, a subset of all the java repositories publicly available on Github has been crawled, processed and indexed into elastic search. The aim is to build a platform where a user can retrieve their function of interest if it has already been implemented. We evaluate this platform by computing the normalised discounted cumulative gain (nDCG) over three queries of various difficulties and test the user friendliness of the user interface. It is found that enabling the boosting functionality function improves the nDCG in most cases and that the engine is able to retrieve the functions of interest in the top results.

1 Background - Introduction

This project aims to ease KTH students's life by providing a searchable database of publicly available snippets of code. For that purpose, we apply information retrieval techniques in order to create a unique search engine able to browse through a subset of java files extracted from Github.

In the following section, related work concerning search engines created with elastic search is cited and explored. Then, the techniques applied to this project are detailed and motivated. Following this section, the performances of the search engine are investigated through a set of experiments whose results are also discussed in the section after. Finally, a complete overview and review of the project is conducted, introducing potential further work.

2 Related Work

Github already has a search engine able to look into repositories, code, commits, issues, packages, marketplace, topics, wikis and users. Github's search engines uses the open-source full-text search and analytics engine from the Elastic Search

API [1]. Around two billion documents made out of all types of publicly available data have been indexed in elastic search so far. Nonetheless, Elasticsearch is also used to investigate diverse types of errors thrown by users' code, as well as identifying whether a user's account has been hacked. Though we do not aim to go as far as to investigate the analytics functions of Elasticsearch in this project, we intend to extend the search engine function by retrieving the entire body of the function of interest rather then give a link to the repository containing the terms of our query.

In order to retrieve the necessary features out of a java file from python, only a few libraries are currently openly available. Javalang [4] is one of them and allows to parse for functions, constructors, abstract classes, interfaces, etc. as well as tokenising the whole code. In this project, we use the version 0.11.0.

3 Methods

3.1 Data Crawling

We needed to crawl the publicly available Github code which we did using the Github API [3]. For this purpose we had to define a subset to extract data from it. Our decision here was to crawl all available Github code related to KTH. For this purpose the query we crawled for was java code containing KTH in the descriptions, repository names or README files.

Unfortunately, the number of functions we crawled using this query were not enough for our evaluation since in some cases we had very few results for certain queries. In order to solve that and boost our results we decided to add more java repositories until we reach a number that would help us evaluate correctly our search engine. The final number of functions that was crawled was 246989.

3.2 Data Extraction

Before indexing the subset of java documents crawled from Github, the relevant features to conduct a search need to be extracted. For this purpose, Javalang library's 0.11.0 version was used in order to parse and tokenise the contents of each of the crawled java file. We extend the functionalities of this library by searching for the end line of the functions of interest, retrieving the Javadoc-format/documentation comments linked to the functions as well as checking for improper code (making sure that all opening bracket statements '{' are closed).

In this project, it has been decided to focus on retrieving functions rather than constructors, interfaces and abstract classes, as the first ones are the most likely to output ready-on-the-go reusable code. For each of those functions we retrieve information proper to that particular function:

- Name of the function

- Start line
- End line
- Modifiers (public/private, static)
- Whole body of the function
- Return type
- Javadoc comments
- Github Url
- Input variable types

as well as information proper to the class and repository the function belongs to such as:

- Name of the class the function belongs to
- Name of the repository (author_name/repository_name)
- Number of stars of the repository
- Number of forks
- Number of watchers

3.3 Search Engine

As our main search engine we used Elastisearch [1]. As mentioned above, Our design decision here was to focus only on retrieving functions, which were indexed and represented by an object described in 3.2.

Our search is based on the query which the user defines in the query section. We run a wildcard search with the query terms in every attribute we store for every function. Since we are searching for functions, in order to boost the results we also concatenate two terms queries and we boost more the concatenated one. We also boost the matches that come from certain attributes such as function names and class names. Furthermore, the user can apply filters in his search based on

- Return type of a function
- Class name that the function belongs to
- Types of the arguments to the function
- Modifiers of the function

In order to implement the filters we used the query string object from Elasticsearch which allows the user to use intersection and union queries in his filters.

On top of our core search engine we implemented an API using flask [2] and Python. When our search engine is started, it indexes all the documents that are provided in a json format in Elastic Search and exposes the endpoint to the functions we created for:

- Process the request sent to the API
- Create the JSON format to be used in the body of search requests to define the queries for Elastic Search
- Process the response generated by Elastic Search

3.4 Relevance and Scoring

Elastic Search by default uses Lucene's Practical Scoring Function [5]. It combines Boolean model, TF/IDF, and vector space models to rank the documents. All the calculations are implemented in the background with further normalization to improve efficiency. It is advised in the Elastic Search documents to not tweak this scoring algorithm for efficiency reasons.

With that being said, in order to tune the relevance in our search engine we used query boosting instead. Considering, our search engine is aimed to retrieve relevant piece of code we boost the fields Class Name, Function Name, Function body and Java doc in decreasing order. Further, we keep a default boost value of 1 for other parameters like github-url, repo-name, parameters, etc. Such boosting allows us to increase precision by lowering the score for the less important matches from repo-names and urls.

Since we index code and not some regular text, we have noticed that it is common to have joint words like "BinarySearch" and "matrixMultiplication" as function names or variable names. And at the same time, it is common to type " Binary Search" and "Matrix Multiplication" while searching. To increase the relevance of results in such cases which we expect to be quite common, we use query expansion followed by term boosting. A search query "Binary Search" will be expanded to "Binary Search BinarySearch" and the joint term "BinarySearch" in the expanded query will be boosted such that documents containing "BinarySearch" will be ranked higher than the documents with just the "Binary" or "Search" or "Binary Search". In our experiments, we have found that this implementation results in retrieval of much higher number of relevant documents for common queries.

4 Evaluation

In order to evaluate the search engine, we build 3 queries of different difficulties, and computed the normalized discounted cumulative gain (nDCG) on average relevance scores. For that purpose, we asked random people (from Q building) to rank the relevance of the first 20 documents from our engine for the same three queries. We also performed an experiment to check how much we can improve by query boosting and term boosting as described in 3.4 in comparison to default scoring of Elasticsearch. People ranking were not informed if they were using a boosted engine or a default one. Moreover, we also ranked results from GitHub's search for same queries to draw a meaningful comparison. In addition to this, we asked a random sample of people to evaluate the user friendliness of both interfaces on a scale from 0 to 5.

4.1 Query evaluation

In this part, we compute nDCG for the following queries with and without boosting by using the average ranking scores:

- "BubbleSort"
- "Matrix multiplication"
- "Binary Search" with return type "int" and "private" modifier

. We ranked the results on a scale of 0 to 3 with the following criteria:

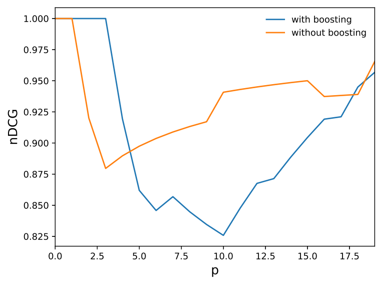
0. Contains no information about the function the user is looking for
1. Does not contain any explicit information about the function of interest, but can lead to it through the URL or pointers present in the function
2. The function could be used with some refactoring and adaptation to the user's purpose
3. The function can be used as is for the user's purpose.

The results are displayed in table 1. As expected, the nDCG is higher for the boosting case, except for the first query. In general, when boosting is not used, the search engine uses all the fields of each document equally. Consequently, it has a tendency to rank documents which match query term in their url or repo-name similarly to documents that have the query term in the function body. This leads to unexpected results. Nevertheless, thanks to the high weight on the name fields (which is higher than the body of the code), the top results after boosting are more relevant for the query. This is also displayed in the figures 1b and 1c.

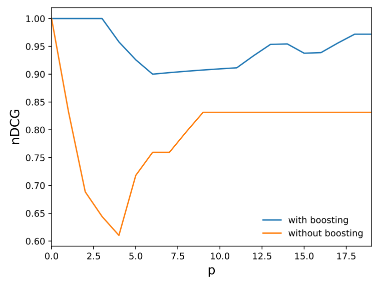
The results for "bubblesort" can be explained with the selection of weights for each field. The "function name" field has a higher weight, but not higher enough to avoid the body of the code to give a high score if it names the query several times (in the title is names once maximum). Moreover, the difference between the gains is small.

Query	nDCG with boosting	nDCG without boosting
"bubblesort"	0.9568	0.9654
"matrix multiplication"	0.9719	0.8314
"Binary Search", Return: "int", Modifier: "Private"	0.9561	0.7214

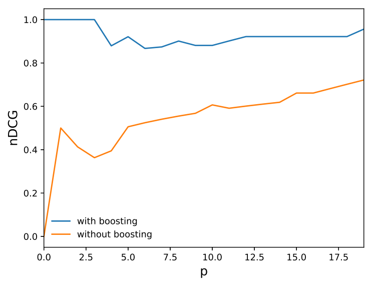
Table 1: nDCG values for each query with and without boosting.



(a)



(b)



(c)

Figure 1: nDCG evolution along the addition of each document (p) for different queries: (a): "bubblesort", (b): "matrix multiplication", (c): advance query "binary search int private"

4.2 User interface evaluation

Two distinct samples of students were asked to rank the KiTHub and Github search user interface according to practicality, and aesthetic on a scale from 0 - 10. The design of both interfaces had their advantages and scored similarly. KiTHub's interface was simple, while github's was sober and people were generally used to it.

Practicality wise, there was a clear preference for KiTHub's interface as it returned the whole body of the function contrarily to Github which returns a snippet of code containing the query terms. The only downside to KiTHub's was that to enter a query the user had to press search and could not press the "Enter" key.

5 Discussion

As expected, deactivating the boosting feature in the search made KiTHub's engine retrieve many functions irrelevant to the original query. Indeed, regardless of whether the query terms are merely found in the github url associated to the repository, in the class name or in the function name, the algorithm retrieves the function irrespectively of where the query term is located. Activating the boosting made a significant difference and allowed the functions implementing the query of interest to be retrieved before those whose class carried the same name as the query term. Github's interface did not allow to retrieve functions so the whole file was evaluated for relevance instead in order to erase the bias "Did not retrieve a function" in the evaluation. Nonetheless, because it only showed a short snippet of code where the query was to be found and linked to files that were sometimes very lengthy, it was tedious to check that what was retrieved was what was needed. The relevance of the fetched did not suffer too much for very common functions, but did for queries that could be expanded such as "Binary Search" into "Binary Search Tree". In the latter case, many irrelevant documents were retrieved as the engine failed to make the difference between the expanded query and the original query.

6 Conclusion

In this project, we create a search engine able to browse through a subset of the java files published on Github. Each crawled file is tokenised and parsed with Javalang. In order to extract all the necessary information, javalang's library is extended to allow the retrieval of the javadoc/comments located above each of the functions, retrieve the end line of a function and check for improper code. The processed information is then indexed in ElasticSearch. The evaluation function of the latter being already efficient and optimised, improvements were made in the boosting where the fields Class Name, Function Name, Function Body and Javadoc were favoured in decreasing order.

We evaluate the performances of KiTHub's engine by collecting relevance indicators from different students on three different queries: "Bubblesort", "Matrix multiplication" and "Binary search, return type : int, modifier: private". Each participant was asked to give a score from 0 to 3 to the first 20 top results of each query. It is noted that enabling boosting seems to eliminate irrelevant documents merely pointing to the functions or whose class/repository name contained the query terms. When comparing Github's and KiTHub's search engine taking away the fact that Github does not retrieve function but code, it has been noted that github was a bit tedious to use but that the design of both interfaces were equally pleasing to the eyes.

In essence, we hope to save precious minutes of your time by providing a platform returning reusable pieces of code, more specifically to return java functions. In most cases, if the function of interest is present in the crawled subset of java files, the engine will return the appropriate function.

References

[1] Elastic Search github use. https://www.elastic.co/fr/use-cases/github. Accessed: 2019-05-19.

[2] Flask. http://flask.pocoo.org/. Accessed: 2019-05-19.

[3] Github API. https://developer.github.com/v3/. Accessed: 2019-05-19.

[4] Javalang repository. https://github.com/c2nes/javalang. Accessed: 2019-05-19.

[5] Lucene's Practical Scoring Function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practica scoring-function.html. Accessed: 2019-05-19.