Contribution of source code identifiers to GitHub project similarity

Which other GitHub projects are similar to yours?

Juul Crienen[1]

Supervisor(s): Dr. Ing. Sebastian Proksch[1]**, Shujun Huang**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
28th January 2024

Name of the student: Juul Crienen
Final project course: CSE3000 Research Project
Thesis committee: Dr. Ing. Sebastian Proksch, Shujun Huang, Julia Olkhovskaia

Abstract

GitHub is an online platform that hosts millions of projects. Many of these projects have the same topic or share the same goal. Finding similar projects which can be used as role models, inspiration or examples can help developers meet their requirements faster and more efficiently. Previous studies have been successful in finding similar GitHub projects, but they do not share how well their proposed metrics indicate similarity.

Our research and analysis seek to find the contribution of source code identifiers to overall project similarity. We define project similarity and define each type of identifier we evaluate. After these steps, we extract the defined types of identifiers from a list of projects. From this list of projects, we use twenty projects as queries for our analysis. We then analyze all identifiers using techniques such as TF-IDF and LSA. Our findings are that combining all types of identifiers results in the highest chance of having the same topic when looking at the most similar project. We also find that splitting each identifier on its casing and combining all split identifiers results in the highest chance that the most similar project found is similar. We therefore see that source code identifiers are reasonably contributing to overall project similarities.

Keywords— GitHub, similar projects, source code identifiers, language processing, text analysis

1 Introduction

Platforms like GitHub provide easy access to millions of open-source software development projects. It is widely used by developers to share code and contribute to projects. Due to the vast amount of public projects and repositories available [4], navigating all these projects and repositories can be quite a challenge.

When working on a software project, developers can often find themselves in unknown territory and might not directly know how to continue with their project. It can therefore be relevant to find similar projects that can be used as role models, examples, or even inspiration. This allows developers to use techniques or patterns from similar projects. Existing research, such as **MudaBlue** [5], **CLAN** [6], **RepoPal** [10] and **CrossSim** [7], already propose different solutions to finding similar projects or repositories. Their findings describe for example categorizing multiple projects based on their shared goals or by finding similarity indices which indicate how two or more projects score in terms of similarity. Almost all of these tools claim they can outperform each other, but none of them indicate how well their methods contribute to the overall project similarity. This paper focuses on one set of identifiers, source code identifiers, to be used as a metric for finding similar projects. Source code identifiers are identifiers that are manually defined by the programmer. This raises the main research question:

How do source code identifiers contribute to similarities among different GitHub projects?

To find an answer to this main research question, we propose three secondary research questions. We study how well source code identifiers indicate similarity to find whether or not there is a contribution. Additionally, we investigate how well source code identifiers help in finding a similar project with the same topic. Finally, we are interested in the contribution of splitting identifiers based on their casing, and if this helps in indicating similarity.

This paper is directed at analyzing the impact of source code identifiers on overall project similarity. Together with four other papers, this paper aims to find an answer to the question: Which other GitHub projects are similar to yours?

We focus on analyzing the importance of source code identifiers when finding similar projects. GitHub stores all source code in repositories. Because we have to extract identifiers from this source code and analyze these identifiers, we adhere to the methodology described below.

We clone a set of repositories and collect all source code files from these repositories. We then extract source code identifiers from these repositories by parsing the source code into Concrete Syntax Trees (**CSTs**). By performing natural language processing techniques such as lemmatization, stemming and Latent Semantic Analysis (**LSA**) to model topics, we identify relationships between different projects. We extract the most similar project and compare this match to manually labeled test data. We analyze how often a match is successful or has the same topic. Additionally, we identify the importance of naming conventions by looking at identifiers that are split based on their casing. This splitting is done based on the casing found in different identifiers.

Since this research focuses on the contribution of source code identifiers to the overall project similarity, we first need to establish what we consider as project similarity. We then define a list of source code identifiers which we will use as a base of our evaluation. Finally, we define how we will split our identifiers and which naming conventions we are considering. Establishing the actual significance of source code identifiers will help to better understand how projects relate to each other and how their overall similarity can be modeled.

Results show that source code identifiers do contribute to finding similar projects. We further explore these results by analyzing the success rate of finding a similar project with the same topic and finding a similar project that is considered similar by manual evaluation. Source code identifiers are successful in finding a similar project with the same topic in 95% of the queries, with the 95% confidence interval being [0.85, 1.05]. Their success rate in finding a similar project is 60% with the 95% confidence interval being [0.38, 0.82]. Furthermore, our evaluation finds that splitting identifiers improves precision by 10% and can result in a better match.

All resources used and mentioned in this paper are published to Zenodo [2].

2 Background and Related Work

2.1 Background

GitHub. GitHub[1] is a web-based platform for software development and version control. It focuses on collaboration and allows software developers to store and manage their code. It uses Git, a tool that is mainly used for tracking changes in files, and adds many features on top of these Git features, such as bug tracking, feature requests, task management, and wikis.

As of November 2023, GitHub has over 420 million total repositories, of which more than 284 million are public [4].

Concrete syntax trees. Concrete syntax trees are used to represent the syntax of source code in a tree form. Each node in the tree is either a root node, branch node, or leaf node. The leaf nodes contain data that directly relates to the original source code. The trees contain all elements that can be found in the original source code. These can include for example: keywords, operators, parentheses, commas, and semicolons. They are often used by IDEs for tasks like syntax highlighting and code formatting.

Tree-sitter. Tree-sitter is a parsing library for programming languages. It can be used for a large amount of both object-oriented and non-object-oriented programming languages, such as Java, C#, and Python. [2] It parses source files and generates concrete syntax trees to represent the source code file. These syntax trees allow the tool to be used for primarily syntax highlighting.

TF-IDF. The **TF-IDF** score helps identify the importance of a term within a document while still considering its importance across a set of documents. The TF-IDF score for a term in a document is obtained by multiplying its term frequency (**TF**) and its inverse document frequency (**IDF**). The term frequency measures how often a term appears in a document. It is calculated by dividing the amount of times the words occur in a document by the total amount of terms in that document:

$$\text{TF}(t, d) = \frac{\text{Number of times } t \text{ appears in } d}{\text{Total number of terms in } d}$$

The inverse document frequency measures the importance of a term across multiple documents. Words that appear more often are seen as less important and therefore given a higher **IDF** score. It is calculated by taking the logarithm of the total number of documents in a corpus (set of documents) divided by the number of documents containing a specific term and adding one:

$$\text{IDF}(t, D) = \frac{\text{Total number of document in corpus } D}{\text{Total number of documents containing } t + 1}$$

The **TF-IDF** score is then calculated by multiplying the **TF** value and the **IDF** value:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

[1] https://github.com/

[2] https://tree-sitter.github.io/tree-sitter/

Latent Semantic Analysis. Latent Semantic Analysis is a technique in natural language processing that can identify hidden relationships between terms in a set of documents (corpus). It is commonly used to perform document clustering, document summarization, and topic extraction. It can calculate the similarity between documents by representing them in a reduced semantic space, capturing the underlying hidden relationships between documents.

2.2 Related Work

MUDABlue. MUDABlue is a tool that automatically categorizes software systems [5]. It focuses on source code identifiers to try and identify the topics of a software system. It uses latent semantic analysis to determine relationships between identifiers and thus can determine if multiple pieces of software systems have the same topic and/or are similar to each other. It does not rely on pre-defined category sets. The categories are generated purely by the source code that is inputted. **MUDABlue** helps to find similar repositories by looking at its source code, but it does not look into how well source code identifiers contribute to similarity.

CLAN. CLAN is an application that detects similar Java applications by looking at their semantic anchors, mainly API calls [6]. It argues that if the same API calls are used in two different applications, their similarity index should be higher than for applications that do not share any API calls. Their tool can be run on Java applications and according to their findings, it performs with a higher precision than **MUDABlue** [6].

RepoPal. RepoPal is a tool proposed by Y. Zhang et al. [10] that can find similar repositories based on three heuristics. These heuristics are:

1. Projects that are starred by the same users within a short period are likely to be similar to one another.
2. Projects that are starred by similar users are likely to be similar to one another.
3. Projects whose README files contain similar contents are likely to be similar to one another.

They calculate two relevance scores based on these three heuristics and based on these calculated scores, they have built RepoPal, a recommendation system to detect similar repositories.

CrossSim. CrossSim is a tool used for identifying similar projects [7]. It makes use of graphs to represent open-source software. This graph includes relationships between projects. These relationships include project dependencies, shared developers, user stars, relationships between users and projects, and more. Their dataset[3] compares the results of **MUDABlue**, **CLAN**, **RepoPal** and **CrossSim** and they find that their tool outperforms the other tools.

3 Methodology

3.1 Experimental setup

In order to successfully find the significance of source code identifiers in the similarity between GitHub projects, the

[3] https://github.com/crossminer/CrossSim/tree/master/dataset/

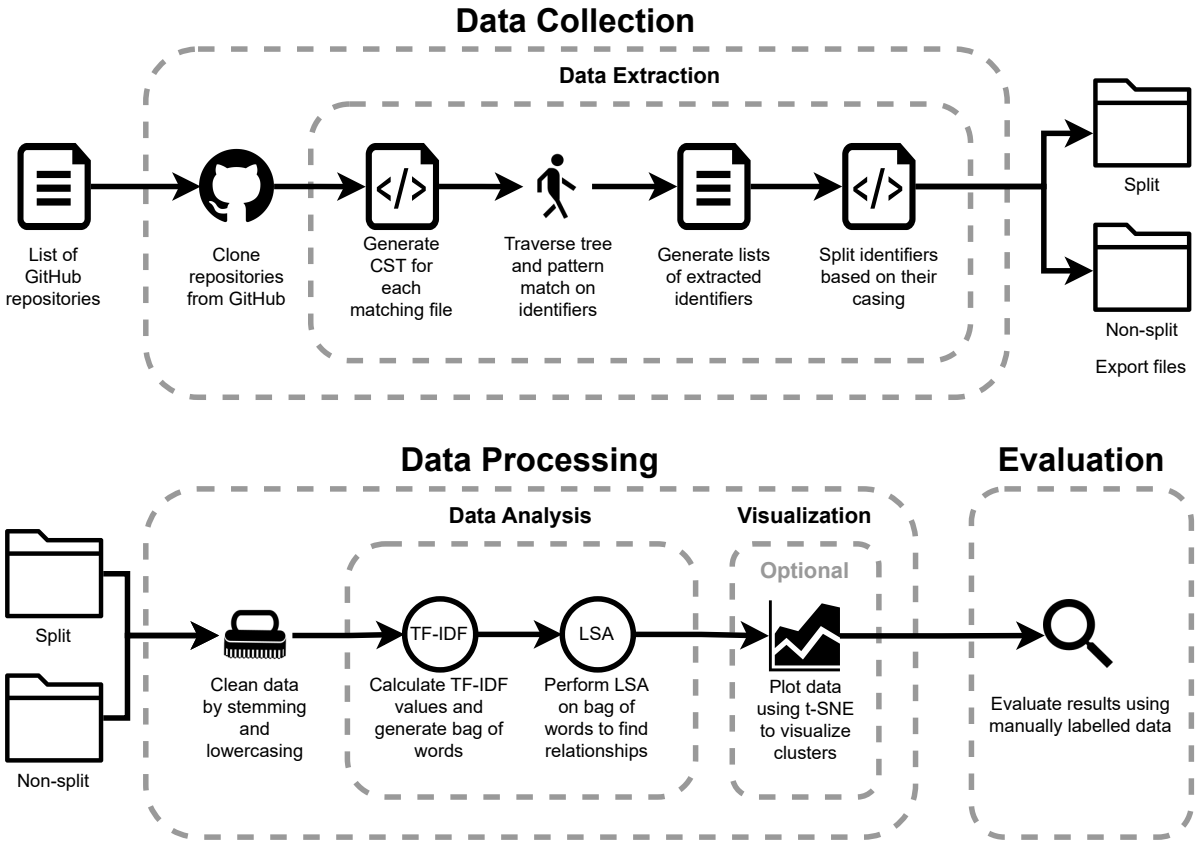


Figure 1: Methodology flow

methodology is divided into three parts. As shown in figure 1, the three parts consist of data collection, data processing, and evaluation. Before any data collection or processing is done, we have to specify some definitions.

3.1.1 Definition of source code identifiers

This paper focuses on similar projects and source code identifiers. To find similar projects, we first have to define what similarity amongst GitHub projects is. The following definition is used for evaluation:

> *Two or more GitHub projects are similar to each other if they share at least one topic and have at least something in common.*

This statement can be interpreted as follows: if two or more projects share a topic, they are more similar to each other than they are similar to projects without this topic. Additionally, two or more projects are more similar if they have the same goal. As an example, a goal can be that two or more projects provide examples for Spring Boot. A project can contain multiple topics and these topics are manually defined by the project admin/owner. By using a GitHub search tool, which will be discussed in section 3.1.3, it is easy to generate a list of generally similar projects based on our definition. This is done by inputting a topic.

This research limits itself to the evaluation of Java projects. Since Java contains multiple different types of identifiers, we have to find a suitable definition for what we consider as source code identifiers. In short, we are only considering identifiers that are manually defined by the programmer. The following types of identifiers meet this criteria:

Global variables: Global variables in Java source code are considered variables that can be accessed (at least) from anywhere within the same class. They are defined outside of methods. In the example in figure 2, the variables `globalString`, `globalInt` and `globalIdentifier` are considered global variables. There is no distinction between modifiers.

Local variables: Local variables are variables that can only be accessed from inside specific functions. In the example in figure 2, the variables `localString` and `x` are considered local variables, although the latter will be rejected in future steps due to its length.

Class names: Class names are the identifiers defined by the user to identify a class. In the example in figure 2, the identifier `ExampleClass` is considered a class name.

Method names: Method names are the identifiers used in a class to identify its methods and/or functions. In the example in figure 2, the identifier `exampleMethod` is considered a

```
public class ExampleClass {
    private String globalString;
    public static int globalInt;
    public OtherClass globalIdentifier;

    private void exampleMethod(String parameterString) throws IOException {
        String localString = 'example';
        int x = 0;
        return x;
    }
}
```

Figure 2: Example source code to help identify source code identifiers

method name. There is no distinction between its access modifiers, return type, or any other modifiers.

Parameters: Parameters are identifiers passed to a function or method. In the example in figure 2, the identifier `parameterString` is considered a parameter. There is no distinction between its possible types.

Type identifiers: The final identifiers that could be considered a source code identifier are type identifiers. Although they are not used in this paper, it might be worthwhile mentioning these identifiers since they could help contribute to the final results. They are currently not used because, in Java, type identifiers often refer to either internal objects or external libraries. Since the focus of this research is purely local source code, these are neglected. In the example in figure 2, the identifiers `String`, `int` and `OtherClass` are considered type identifiers.

We also evaluate whether combining these identifiers influences the output. This results in a new class: *all*.

3.1.2 Definition of naming conventions

Programmers tend to use different casings and naming conventions in terms of identifiers. To evaluate whether these different casings impact the contribution of source code identifiers, we analyze the normalized identifiers and the non-normalized identifiers. We define the normalized identifiers as split identifiers and the non-normalized identifiers as non-split identifiers. The identifiers are split using the regular expression shown in figure 3, which supports all the casings described next.

```
(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|_|-
```

Figure 3: Regular expression for splitting identifiers

Snake Case: Snake Case uses an underscore to distinguish its words. An example would be `snake_case_example`. In the regular expression used, underscores are a criterion for splitting the identifiers.

Camel Case: Camel Case uses lowercase for its first word. The first letter of all other words is capitalized. This results in `camelCaseExample`.

Pascal Case: Pascal Case is similar to Camel Case. The difference between the two is that the first letter is capitalized as well. An example is `PascalCaseExample`. The first part of the regular expression is able to recognize Camel Case and Pascal Case.

3.1.3 Data Collection

The data collection process makes use of a GitHub search tool proposed by the SEART (SoftwarE Analytics Research Team) group [3]. This tool makes it possible to search GitHub projects based on multiple inputs such as topics, number of commits, programming language, number of issues, and more. It is capable of generating a CSV output that contains all matching projects with its metadata. We then evaluate the list of projects and all repositories are cloned to a temporary folder. By making use of TreeSitter, we generate a **CST** to successfully find matching identifiers. To successfully parse the source code into **CSTs**, a Java implementation of Tree-sitter, found on GitHub [4], is used. It is compiled on a Unix-based operating system and requires additional grammar for each desired programming language. The current compilation of the library includes support for Java, C#, and JavaScript. A fork of the Java implementation of Tree-sitter including the used grammars is published to Zenodo [2]. The **CST** is then traversed while maintaining the current and previous states. This allows us to pinpoint the exact location of the identifier, thereby enabling the detection of the type of source code identifier. By utilizing our definition of source code identifiers, we extract all relevant identifiers. Subsequently, these identifiers are stored in lists categorized by type and per project. The next step splits identifiers based on their casings, as described previously. This allows the analyzing tool to find hidden semantics in both the original data and the modified data. Both types are considered to find a precise conclusion.

The data is finally stored in two separate folders: one folder contains the original data, while the other contains the data with the identifiers being split.

3.1.4 Data Processing

We carefully clean the data obtained from the first part using existing libraries from Python's Natural Language Toolkit (**NLTK**) [1]. Identifiers are stemmed and lowercased to better identify similarities. The collection of pre-processed data

[4] https://tree-sitter.github.io/tree-sitter/

Scale	Description	Score
Dissimilar	The two projects are completely different	1
Neutral	The two projects share a little in common	2
Similar	The two projects share something in common	3
Highly similar	The two projects share many things in common and can be considered the same	4

Table 1: Similarity scores as listed in **CrossSim** [7]

is then converted into a matrix of **TF-IDF** features using SciKit's TfidfVectorizer [8]. This vectorizer makes it possible to filter out words. Words that occur less than two times or occur in more than fifty percent of the documents are ignored. This generates a bag of words, which contain the set of all words paired with their **TF-IDF** value. The resulting data is then analyzed using **LSA**, which utilizes SciKit's TruncatedSVD [8]. This **SVD** performs dimensionality reduction on the previously found data and results in a **LSA** matrix. Since this matrix is not interpretable by humans, it can be useful to use tools to visualize this data. The optional visualization step is therefore introduced to plot the data to manually evaluate clusters. **t-SNE** [8] is used to visualize the high-dimensional data. We then create a 2D plot and visualize these results to show clustering based on **LSA** together with **TF-IDF**.

In an approach proposed by Sun et al. [9], **LSA** is not used, but instead a cosine similarity matrix is generated to find similar projects. Their input consists of description files and source code files. Instead of extracting specific source code identifiers, their proposed solution uses full source code files. They remove words shorter than three characters and remove numbers and other meaningless identifiers. Our proposed method uses **LSA** combined with **TF-IDF** on the type of identifiers defined in section 3.1.1.

3.1.5 Datasets

A dataset generated by the GitHub Search Tool is used for our analysis. The dataset contains 570 repositories of five different categories. Twenty of these repositories are randomly selected and used as queries. For each query, the most similar repository is selected for evaluation. The dataset and queries are published online [2].

3.2 Research Questions

As mentioned previously, the main research question for this paper is:

> *How do source code identifiers contribute to similarities among different GitHub projects?*

To successfully answer the main research question, we propose the following secondary research questions:

Q_1. *How well do source code identifiers indicate similarities between GitHub projects?*

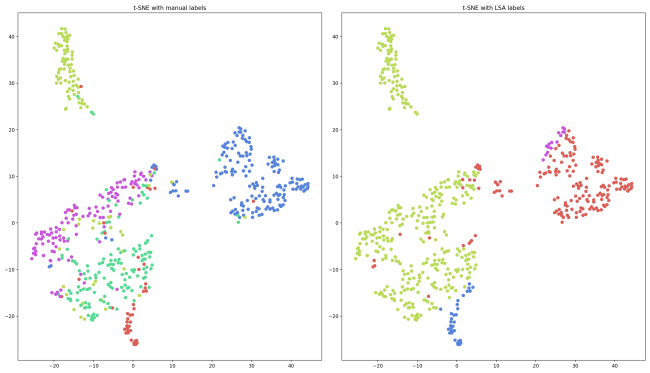
This research question helps to answer the main research question by looking at whether source code identifiers are a good indicator for measuring similarity. We are interested in this because it will help to determine whether or not source code identifiers contribute at all. Our goal is to find a confidence interval that indicates how successful our analysis is in finding a similar project. Our method starts after we have followed the data collection and data processing steps described in figure 1. After we obtain our **LSA** matrix, we calculate the cosine similarity using SciKit's Pairwise Metrics [8]. For each query, we find the most similar project. The results are then stored for manual evaluation.

Each match is compared to manual labeling done by a total of two experienced Java developers, including the author. All the labeling done by one developer has been checked by the other developer to ensure consistency. Potential conflicts are discussed and resolved in such a way that there are no disagreements. It is checked if the two projects have similarities by looking at the projects' goals. If the projects share the same goal or implement the same algorithm for a different resource, the match is classified as either similar or highly similar, depending on how well they match. If the two projects do not have anything in common, they are classified as dissimilar. If they share only little in common, for example, if they are both extensions for some software but do not share the same type of extension or goal, the match is classified as neutral. If the match is classified as dissimilar or neutral, the match is considered a *false positive*. If the match is classified as similar or highly similar, it means that in the case of that specific match, source code identifiers are an indicator of similarity, and are therefore considered a *true positive*. The scores given by the Java developers range from one to four and are the same as the similarity scores used in **CrossSim** [7]. For reference, table 1 lists the four similarity scores.

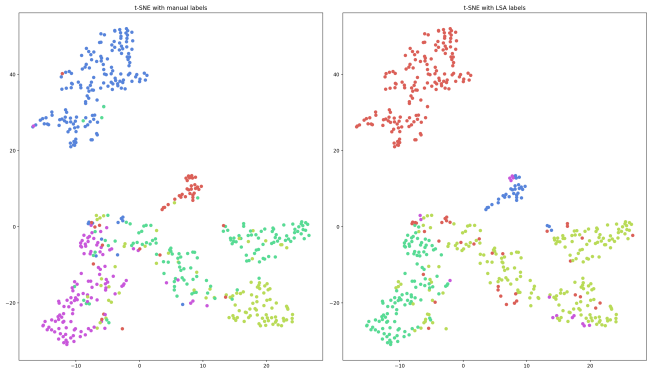
For all matches, we evaluate how many of them have been classified as similar or highly similar. For each type of identifier, both split and non-split, as well as all identifiers combined, we calculate the confidence intervals. Section 4 highlights all relevant results.

Q_2. *Do source code identifiers help find a similar project with the same topic?*

This research question helps to answer the main research question by determining whether source code identifiers are a good indicator for finding a similar project that has the same topic. As described in 3.1.4, the data processing step introduces an optional step to visualize our data. The plots generated using **t-SNE** are used to manually interpret our data. We also analyze the matches by comparing the topic of the query with the topic of the match. We are calculating the confidence intervals for the binary values [0, 1] with 0 equalling non-matching topics and 1 equalling matching topics. Section 4 highlights all relevant results.



(a) t-SNE performed on the LSA matrix for combined identifiers



(b) t-SNE performed on the LSA matrix for combined split identifiers

Figure 4: t-SNE plots showing clustering and topics as target labels

Type	Mean	Count	Standard deviation	Confidence α= **0.05**
all	0.55	20	0.5104	0.2237
all_split	0.60	20	0.5026	0.2203
class_names	0.35	20	0.4894	0.2145
class_names_split	0.60	20	0.5026	0.2204
global_variables	0.40	20	0.5026	0.2204
global_variables_split	0.50	20	0.5130	0.2248
local_variables	0.35	20	0.4894	0.2145
local_variables_split	0.45	20	0.5104	0.2237
method_names	0.35	20	0.4894	0.2145
method_names_split	0.45	20	0.5104	0.2237
parameters	0.40	20	0.5026	0.2203
parameters_split	0.40	20	0.5026	0.2203

(a) Results for project similarity

Type	Mean	Count	Standard deviation	Confidence α= **0.05**
all	0.95	20	0.2236	0.0980
all_split	0.80	20	0.4104	0.1799
class_names	0.70	20	0.4702	0.2061
class_names_split	0.80	20	0.4104	0.1799
global_variables	0.80	20	0.4104	0.1799
global_variables_split	0.80	20	0.4104	0.1799
local_variables	0.85	20	0.3663	0.1606
local_variables_split	0.95	20	0.2236	0.0980
method_names	0.90	20	0.3078	0.1349
method_names_split	0.80	20	0.4104	0.1799
parameters	0.85	20	0.3663	0.1606
parameters_split	0.80	20	0.4104	0.1799

(b) Results for topic matching

Table 2: Analysis for each type of identifier for similar repositories and matching topics

Q_3. ***What are naming conventions in source code and how do naming conventions of source code identifiers contribute to the similarities between GitHub projects?***

Naming conventions among source code identifiers can impact similarity scores. By looking at both split and non-split data we can conclude whether it is better to use the raw or processed data to find project similarity. The methodology used in research question 1 and research question 2 is repeated for the split identifiers. We then compare the results of the non-split identifiers to the results of the split identifiers. This allows us to find their contributions to the overall similarity.

4 Results

The following section highlights all relevant results to the secondary research questions proposed in Section 3.

Q_1. Table 2a displays the results from the similarity evaluation described previously. This table has been visualized into a bar chart that shows the confidence intervals. This can be seen in figure 5. This table and figure show the average amount of times the most similar project found is labeled similar or highly similar using the manual labeling described previously for each of the twenty query projects. Each bar in figure 5 shows the type of identifier with its casing being either split or non-split. When combining all split identifiers, or looking at the split *class names*, we see that their influence on similarity is the highest. Both succeeded in 60% of the queries, with a 95% confidence interval of 0.6 ± 0.22. This shows that combining and splitting all source code identifiers can correctly indicate similarity with a confidence interval of [0.38, 0.82] and are therefore useful for finding similar projects.

Q_2. Initial plots generated using **t-SNE** shown in figure 4 show that **LSA** is useful for correctly finding a project with the same topic. Projects that are closer to each other are more related to each other than projects further away. Figure 4a shows the results generated by our analysis for the combined identifiers visualized in 2D. The plots show projects with each color representing the actual topic to which the project belongs. The right plot shows the same clustering where each color represents the predicted topic. Although the colors are different for both plots, one can see that most clusters share a color and are thus identified correctly. Figure 4b shows the same visualization for the combined split identifiers.

Table 2b displays the results from the topic evaluation described previously. This table is visualized using bar charts which can be seen in figure 6. Figure 6 displays the average amounts with their confidence intervals of correctly identified topics in the query set of twenty projects. As one can see, considering the types of identifiers, the combined identifier *all*, together with the split identifier *local variables*, performs the best. They both result in a 95% correctness aver-

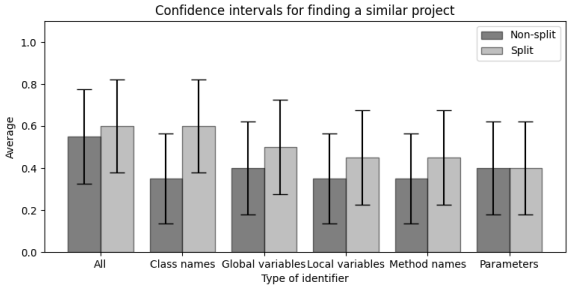


Figure 5: Results for finding a similar project

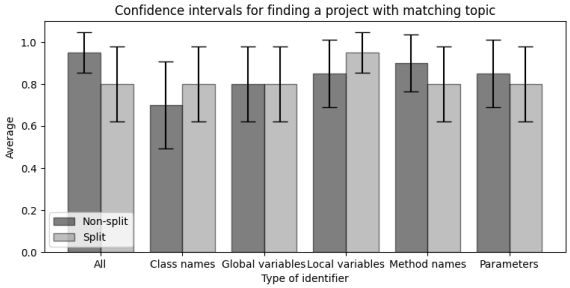


Figure 6: Results for finding a project with matching topic

age with a 95% certainty that their confidence interval is 0.95 $\pm$ 0.098. In general, all types of identifiers, either split or non-split, perform well in correctly matching with a project having the same topic. The only type of identifier that performs worse than the others is *class names*. We can therefore say that source code identifiers do strongly help in finding a similar project with the same topic.

Q_3. When looking at our findings for $\mathbf{Q_1}$ we can see that in general, splitting the identifier results in a higher chance of finding a similar project. On average, this results in a 10% higher chance of finding a similar project compared to non-split identifiers. Although not splitting the identifiers can help find similar projects, it is generally more useful to split the identifiers. Results for $\mathbf{Q_2}$ show that not splitting the identifiers might result in a better chance of finding a similar project with the same topic. On average, not splitting the identifiers results in a 1.5% higher chance of finding a similar project with the same topic. Since splitting the identifiers results in a 10% higher chance of finding a similar project and non-splitting the identifiers results in a 1.5% higher chance of finding a similar project with the same topic, we find that, in general, splitting the identifiers yields better results.

5 Responsible Research

5.1 Reproducibility

This research relies on existing implementations of techniques such as **TF-IDF** and **LSA**, and language processing techniques for stemming and tokenization. For each existing implementation used, we reference the original library to ensure reproducibility. Additionally, all resources used to conduct our analysis are published to Zenodo [2], including all source code and datasets. The results are published for people who want to either use the results or conduct their experiments based on our results.

Some existing implementations require specific parameters. Using different parameters can impact the outcome of the analysis. Parameters that directly impact the results are mentioned in this paper. Parameters that only affect visualizations have not been discussed but can be found in the published source code. These parameters do not affect the outcome of this research but can influence one's perspective of the plotted results.

The evaluation of this research is done by manually labelling data using a set scale of scores. This process uses two experienced Java developers, including the author. Both the author and the other developer used the same approach and discussed matches that were unclear. This ensures that bias is minimized. All manually labelled data is published as well.

Each step in the methodology, every metric, and all evaluation steps to obtain our results have been described in detail in such a way that it is possible to reproduce this research.

5.2 Threats to validity

Our definition of similarity and our definition of naming conventions are carefully crafted using common findings in programming. Small changes in these definitions can impact the results and therefore change the outcome of our analysis. It is therefore important to adhere to the definitions presented in this paper when using our dataset or results as a base.

The plots presented in figure 4 are created using dimensionality reduction performed using **t-SNE**. However, this does mean that another iteration can result in small changes in these plots. These plots are only used as an initial interpretation of the results, and the exact results are presented in table 2.

Our evaluation required a set of twenty queries which are then used to generate the results discussed previously. This set of queries is randomly generated, and selecting a different set of queries can affect the outcome of this experiment.

A small number of mistakes made in labelling the data can minimally affect the results, but because two people were involved in performing this task, errors are minimized.

6 Discussion

6.1 Reflection on results

The results mentioned previously show that source code identifiers contribute to GitHub project similarity. These conclusions have been made by visualizing data generated by **TF-IDF** and **LSA** as well as comparing our similarity results to a manually labeled dataset. Results from CrossSim [7] show that **MudaBlue** [5], **CLAN** [6], **RepoPal** [10] and **CrossSim** [7] all succeed in finding similar repositories, with some tools performing better than the others. In particular, **MudaBlue** uses source code identifiers to categorize software systems. These results, however, do not show which type of source code identifiers contribute the most, whether it is useful to

split identifiers, and whether their contributions are meaningful. Our results show that source code identifiers do contribute to finding similar repositories, and we further evaluate the results to find the performance of each metric, both split and non-split.

Our analysis shows that for all combined non-split identifiers, the most similar project has a matching topic in 95% of the cases with reasonably high confidence. A reason that non-split identifiers might perform better than split identifiers in this scenario is that projects that share the same topic often tend to use the same method names or parameter names. Splitting these identifiers can cause a match to be too general. Therefore, not splitting the identifiers increases precision and results in a higher average.

Splitting the identifiers improves the ability to determine the most similar project. An explanation for this would be that non-split identifiers are generally too rare and not found in a lot of other projects. Splitting these identifiers therefore results in multiple words which can yield multiple matches and therefore show a more suitable match. There is a smaller chance that their topics are the same but a higher chance that they share the same goal.

Our results show that source code identifiers do contribute to finding similar repositories and similar repositories with matching topics. This would indicate that in general, source code identifiers are a reasonable metric for measuring similarity. An explanation for these results would be that most projects that have a similar goal implement the same types of functionality and therefore often tend to use the same identifier names. Combining all these identifiers results in a bigger set of words that are being matched and therefore usually results in a better match. This explains why the combined set *all split* performs the best. There are more words to match on and therefore a higher match precision.

6.2 Future Work

During our initial definition of source code identifiers, we mentioned *type identifiers*, but did not consider this a source code identifier. Results show that combining all identifiers yields the highest success rate, but the impact of *type identifiers* is not known. A recommendation is therefore to research the influence of *type identifiers* on the similarity in GitHub projects. The analysis conducted in this paper consists of 570 projects of which twenty are used as queries. Analyzing a larger set of projects and/or queries can have a positive or negative impact on the results. Another recommendation is therefore to research the effect of a larger test set and query set on this analysis. It might also be worthwhile looking into different clustering techniques that can either improve or worsen the topic clustering results. Therefore, it can be interesting to research how well different clustering techniques can model GitHub projects when only looking at source code identifiers.

The identifier extractor used in this paper is published online [2]. This extractor works but does currently not use error handling to its full potential. We, therefore, recommended further developing this tool to make extracting identifiers faster and error-free. Our final recommendation is to combine our research with the research of other team members to find a high-performing metric for finding similar GitHub repositories.

6.3 Conclusions

Finding similar GitHub projects can be useful in software development, as these can be used as role models, examples, or inspiration. Existing research shows that different methods proposed (**MudaBlue** [5], **CLAN** [6], **RepoPal** [10], **CrossSim** [7]) succeed in finding similar repositories, but do not indicate how well their proposed metric performs in terms of overall project similarity. Since this research focuses solely on source code identifiers, we seek to find the contribution of source identifiers to similarities among different GitHub projects. We first create a definition for project similarity and find which identifiers we are considering. Our proposed method collects data from different GitHub projects by cloning the projects and parsing the source code into Concrete Syntax Trees to extract the defined identifiers. We then perform natural language processing techniques on this data to find hidden relationships between different projects.

Our analysis shows that source code identifiers can be a key feature in identifying similar projects. We define similarities in GitHub projects and the types of identifiers that we consider in this paper. Furthermore, we seek to find an answer to the main research question:

> *How do source code identifiers contribute to similarities among different GitHub projects?*

We analyze the effectiveness of source code identifiers in indicating similarity. Additionally, we investigate how well source code identifiers help in finding a similar project with the same topic. Finally, we are interested in evaluating whether splitting identifiers based on their casing can help indicate similarity.

Combined and split source code identifiers, on average, contribute about 60%, with a 95% confidence interval ranging from 38% to 82%, to these similarities. Evaluating all types of identifiers analyzed results in finding that combining all identifiers yields the best results. We have analyzed whether splitting identifiers impacts similarity results and have found this to be true in a positive sense.

Furthermore, we have found that source code identifiers are useful in finding a similar repository with a matching topic. Results show that combining all identifiers results in a 95% correctness average with a 95% certainty that their confidence interval is 0.95 ± 0.098.

We can conclude that based on our analysis, source code identifiers are a reasonable metric for finding similar projects and therefore do contribute to similarity.

References

[1] S. Bird, E. Klein and E. Loper. *Natural language processing with Python*. 1st ed. OCLC: ocn301885973. Beijing ; Cambridge [Mass.]: O'Reilly, 2009. 479 pp. ISBN: 978-0-596-51649-9.

[2] J. G. M. Crienen. *github-source-code-identifiers: 1.0-SNAPSHOT*. Version SNAPSHOT. 22nd Jan. 2024. DOI: 10.5281/ZENODO.10551141.

[3] O. Dabic, E. Aghajani and G. Bavota. *Sampling Projects in GitHub for MSR Studies*. 8th Mar. 2021. arXiv: 2103.04682[cs].

[4] K. Daigle and GitHub. *Octoverse: The state of open source and rise of AI in 2023*. The GitHub Blog. 8th Nov. 2023. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/ (visited on 27/11/2023).

[5] S. Kawaguchi et al. "MUDABlue: an automatic categorization system for open source repositories". In: *11th Asia-Pacific Software Engineering Conference*. 11th Asia-Pacific Software Engineering Conference. ISSN: 1530-1362. Nov. 2004. DOI: 10.1109/APSEC.2004.69.

[6] C. McMillan, M. Grechanik and D. Poshyvanyk. "Detecting similar software applications". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE 2012). Zurich: IEEE, June 2012. ISBN: 978-1-4673-1066-6 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227178.

[7] P. T. Nguyen et al. "CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Prague: IEEE, Aug. 2018, pp. 388–395. ISBN: 978-1-5386-7383-6. DOI: 10.1109/SEAA.2018.00069.

[8] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. ISSN: 1533-7928.

[9] X. Sun et al. "Personalized project recommendation on GitHub". In: *Science China Information Sciences* 61.5 (20th Apr. 2018), p. 050106. ISSN: 1869-1919. DOI: 10.1007/s11432-017-9419-x.

[10] Y. Zhang et al. "Detecting similar repositories on GitHub". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt, Austria: IEEE, Feb. 2017, pp. 13–23. ISBN: 978-1-5090-5501-2. DOI: 10.1109/SANER.2017.7884605.