Discovering Digital Siblings

Quantifying Inter-Repository Similarity Through GitHub Dependency Structures

Mateusz Rębacz

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
January 28, 2024

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

Abstract

Open Source developers typically use Git repositories to transparently store the source code of projects and contribute to the code of others. There are millions of repositories actively hosted on platforms such as GitHub. This presents an opportunity for sharing knowledge between related projects – the so-called *digital siblings*. Finding repositories similar to one's own can allow for better developer collaboration and knowledge transfer. However, due to the large volume of projects, manually locating digital siblings of a project can be difficult. Hence, this paper proposes a novel approach, based on the dependency structures of GitHub repositories, that allows for calculating inter-repository similarity and subsequently querying for similar projects. We aim to answer the research question: How can the dependency structures of GitHub repositories be leveraged to find their digital siblings? This research includes an empirical evaluation of various similarity metrics and clustering techniques for GitHub repositories. Our results show that dependency structures are a reliable characteristic for measuring similarity between projects. We also identify the specific metrics and clustering techniques as particularly efficient. Lastly, we propose and evaluate a composable similarity metric to allow our findings to be combined with the research of the other Research Project group members.

1 Introduction

Open Source development is a collaborative approach to creating software where developers from around the world can freely access, modify, and contribute to a project's source code. Git repositories are typically used in Open Source projects for storing the project's code and tracking changes. They allow developers to contribute to existing projects, fork codebases for new initiatives, and leverage each other's work to build more complex software systems. [Spinellis, 2012]

Currently, GitHub, the largest platform for Open Source development, holds over 100 million repositories [Warner, 2018]. This vast number of projects creates a unique knowledge base for developers, allowing them to learn from repositories similar to their own. We introduce the term *digital siblings* to refer to repositories that have similar goals or share the same problem domains. An example of such could be a set of all Android social-media applications. Finding digital siblings allows their developers to collaborate, improves code reuse, and simplifies the implementation of boilerplate-heavy operations such as CI/CD workflows. These factors are very desirable for developers, however, given the immense number of GitHub repositories manually locating digital siblings becomes a daunting task. An automated solution is needed.

Hence, we investigate dependency structures found within GitHub repositories as a metric for calculating inter-repository similarity which will lay the groundwork for automatically querying digital siblings. Dependency structures refer to the graph of external libraries and frameworks that a project relies on for its functionality. Analyzing these structures gives us insight into the project's technical choices such as the frameworks and tools used. We theorize that this information can uncover the project's goals and problem domain, and can therefore be used as a metric to evaluate project similarity.

The objective of this paper is to investigate how dependency structures can be used to measure similarity between GitHub repositories. The research is conducted as part of a Research Project (RP) group at the TU Delft, where each member focuses on another metric for GitHub similarity analysis. We guide our research by investigating the following research questions:

- **Main research question** (RQ1): How can the dependency structures of GitHub repositories be leveraged to find their digital siblings?
- **RQ2**: What metrics, derived from analyzing dependency structures, most accurately quantify the similarity between GitHub repositories?
  Justification: By finding an effective similarity metric based on dependency structures that can discern between digital siblings and dissimilar projects, we show that dependencies can be used to effectively measure the similarity between two repositories.
- **RQ3**: Which clustering methods are most effective in grouping GitHub repositories into clusters mirroring similar problem domains?
  Justification: By finding an effective clustering technique, based on dependency structures, that can cluster groups of digital siblings, we show that dependencies are an effective metric for categorizing and querying large sets of repositories.
- **RQ4**: How can dependency structures as a similarity metric be composed with the similarity metrics investigated by the other RP group members?
  Justification: By finding a composable similarity metric we allow our research to be combined with the work of the other RP group members. This allows a more accurate, multi-modal similarity metric to be developed in the future.

We demonstrate that dependency structures are indeed a robust and practical metric for finding digital siblings. We show the most effective similarity metrics, and clustering techniques for grouping similar GitHub repositories. Lastly, we propose and evaluate a repository similarity metric, based on dependency structures, which is composable with the work of the other RP group members.

In the next chapter, we explore the existing body of research to contextualize our study. Chapter 3 details the methodology used to answer our research questions. Chapter 4 presents our empirical findings and delves into the analysis of these results. In Chapter 5 we reflect on the implications of our findings in the broader context of software development, we discuss practical applications, and the future work stemming from our findings. Chapter 6 summarizes the key insights and contributions of our research. Finally, Chapter 7 outlines how our methods and findings are considered

through an ethical lens and assesses the reproducibility of our study.

2 Literature Review

There have been several previous approaches to querying software repositories based on their similarity. Each has contributed uniquely to our understanding of how repositories can be related and grouped.

- **MUDABlue** is an approach that relies on Latent Semantic Analysis of the source code text to assign each repository a category. [Kawaguchi et al., 2006] This approach is quite effective and can be applied to repositories using any technology or coding language. However, the reliance solely on lexical analysis may miss deeper structural or functional similarities between repositories, and therefore degrade accuracy.
- **CLAN** focuses specifically on Java projects, and utilizes package class hierarchies as its basis for determining repository relationships. By comparing the way Java projects are structured, CLAN achieves a higher accuracy than MUDABlue [McMillan et al., 2012]. However, its applicability is limited to Java repositories only and therefore lacks robustness.
- **RepoPal** leverages GitHub stars metadata and Readme file contents to ascertain similarities between projects. Interestingly while it does not consider any code-centric metrics, the approach provides a higher accuracy and confidence than the previously mentioned CLAN. It is also much more robust than the previous metrics [Zhang et al., 2017].
- **CrossSim** is an approach that combines various similarity metrics including both source code identifiers and repository metadata, to provide a more comprehensive similarity score. Combining code-centric and metadata-centric metrics makes the method more robust to differences in repository structures and technical underpinnings. CrossSim therefore performs more accurately than the previously mentioned techniques [Nguyen et al., 2018].

Each technique explores a different characteristic of software projects and proposes a different similarity metric. However, no previous work has focused on dependency structures as a metric for repository similarity analysis. This knowledge gap motivates our research into this topic.

3 Methodology and Experiment Setup

We split our research into several key sections using the research sub-questions as a guideline. We begin with Data Collection, which involves compiling a labeled list of GitHub repositories for our experiments. Next, the Dependency Extraction section outlines how we extract and store dependency information from each repository in our dataset. The Data Selection section outlines how we split our dataset based on dependency types. Then, in the Repository Vectorization section, we transform the dependency data into a quantifiable vector format.

Subsequently, the experiments are conducted on the vectorized repository data in three key sections:

- **Similarity Metrics** (answers *RQ2*): We investigate the performance of a range of repository similarity metrics.
- **Clustering Techniques** (answers *RQ3*): We explore the performance of various clustering techniques in effectively grouping digital siblings.
- **Adjusting for Composability** (answers *RQ4*): We explore a hybrid approach utilizing clustering to reduce the dimensionality of the dependency data and create a similarity metric composable with the work of the other RP group members.

3.1 Data Collection

Following a discussion with the RP peers, we decided to focus exclusively on Java repositories using either the Maven or Gradle build systems. This choice aims to maintain consistency in the dataset and avoid biases that might arise from the use of different code and dependency ecosystems. Java, being a widely used and established programming language, offers a vast array of software projects spanning different problem domains [StackOverflow, 2023]. We have selected the following three categories of software:

- **Minecraft Modifications (*mc-mod*)**: Software intended to modify or extend the functionality of Minecraft game clients.
- **Minecraft Plugins (*mc-plugin*)**: Software intended to extend the functionality of Minecraft game servers.
- **Android Libraries (*android-lib*)**: Software and tools extending the functionality of Android applications.

These sets were chosen due to their active use of dependencies and varying degrees of dependency overlap. For each category, we identified the top 50 most-starred repositories on GitHub. The use of the star count as a selection criterion served a dual purpose: it indicated a repository's popularity and community engagement, and it also suggested high maturity in the repository's development, which is often accompanied by well-structured dependency systems.

A significant challenge faced during data collection was the inconsistency in the use of build systems across repositories. Some repositories lacked a properly set up build pipeline, which would make the automatic extraction of dependencies challenging. To address this, we refined our data collection criteria to include only those repositories with a well-defined build pipeline. Our dataset can be seen in appendix A.

3.2 Dependency Extraction

The extraction of dependencies was automated using a tool developed specifically for this study. Each repository in the dataset is cloned and extracted as follows:

- For **Maven** projects, the tool executes the following terminal command in each project directory containing a pom.xml file.

```
$ mvn dependency:tree
```

This command generates a dependency tree representation, detailing both direct and transitive dependencies of the project. The output is then parsed, extracting relevant dependency information.

- For **Gradle** projects, the tool executes the following command in the repository's root directory.

  ```
  $ ./gradlew projects
  ```

  This command outputs a list of all Gradle projects within the repository. The tool then iterates over each project, executing the following command, where PROJECT is the id of the Gradle project.

  ```
  $ ./gradlew PROJECT:dependencies
  ```

  Similarly to Maven, this command provides a detailed dependency tree including both direct and transitive dependencies. The output is then parsed, extracting relevant dependency information.

The extracted dependency data for each repository is stored in a JSON file `mined.json` [Rębacz, 2024]. This file is structured as an array, with each element corresponding to a unique cloned repository and storing the repository name, author, category, and a comprehensive list of extracted dependencies. For each dependency, its identifier, and type (direct or transitive) are recorded.

3.3 Data Selection

We create two sets of repositories from the initial dataset, to assess the impact of including transitive dependencies in our evaluation of repository similarity metrics. We construct the following sets:

- **All Dependencies (*deps-all*)**: This set includes all available dependencies for each repository, both direct and transitive.
- **Direct Dependencies (*deps-direct*)**: Includes only the direct dependencies for each repository.

3.4 Repository Vectorization

Vectorization of repositories involves transforming the extracted dependency data into binary vectors. We scan all repositories in the dataset and create a set of all unique dependencies used, $\mathscr{D}$. Each repository is then assigned a binary vector v where each entry corresponds to a dependency from $\mathscr{D}$. A value of 0 or 1 is assigned to the entry based on the presence or absence of the dependency in the repository, respectively.

$$v_i = \begin{bmatrix} uses(r_i, \mathscr{D}_0) \\ uses(r_i, \mathscr{D}_1) \\ ... \\ uses(r_i, \mathscr{D}_k) \end{bmatrix} \tag{1}$$

$$uses(r, d): \begin{cases} \mathbf{1} \text{ if } d \in r \\ \mathbf{0} \text{ otherwise} \end{cases} \tag{2}$$

3.5 Comparing Similarity Metrics

To address the sub-question RQ2 we investigate applying various similarity metrics to the vectorized sets of repositories and measuring their performance. The investigated metrics include Euclidean Distance, XOR Similarity, and AND Similarity.

Performance Metric The performance of the similarity metrics is measured as follows. We calculate the average similarity score for repositories within the same category and compare it against the average similarity score between repositories in different categories. This is defined as follows:

Assume we have a set of repositories R and a set of categories C. For each category $c \in C$, let R_c be the set of repositories in category c, and $R_{\text{outside},c}$ be the set of repositories not in category c. The average similarity within a category c, denoted as $S_{\text{inner},c}$, and the average similarity outside of a category c, denoted as $S_{\text{outside},c}$, are calculated as follows:

$$S_{\text{inner},c} = \frac{1}{|R_c|^2 - |R_c|} \sum_{r_i \in R_c} \sum_{r_j \in R_c, r_i \neq r_j} \text{sim}(r_i, r_j) \tag{3}$$

$$S_{\text{outside},c} = \frac{1}{|R_c| \cdot |R_{\text{outside},c}|} \sum_{r_i \in R_c} \sum_{r_j \in R_{\text{outside},c}} \text{sim}(r_i, r_j) \tag{4}$$

Here, $\text{sim}(r_i, r_j)$ is the similarity score between repositories r_i and r_j, $|R_c|$ is the number of repositories in category c, and $|R_{\text{outside},c}|$ is the number of repositories not in category c.

Finally, our performance indicator is the similarity-difference factor SDF_c, defined as follows:

$$SDF_c = \frac{S_{\text{within},c}}{S_{\text{outside},c}} \tag{5}$$

The similarity-difference factor determines the effectiveness of each similarity metric in distinguishing repositories of the same category from those of different categories. The metric that shows the largest factor SDF_c is deemed the most efficient.

The following sections describe the specifics of each of the investigated similarity metrics.

Euclidean Distance The Euclidean Distance metric computes the distance between repository dependency vectors in a multi-dimensional space. For two repository vectors v_i and v_j it is defined as:

$$D_{ij} = \sqrt{\sum_{k=1}^{n} (v_{ik} - v_{jk})^2} \tag{6}$$

$$\text{Similarity} = -D_{ij} \tag{7}$$

Since each entry in a repository vector represents the presence (1) or absence (0) of a specific dependency, the squared differences in the elements of v_i and v_j accumulate to a higher value when there are fewer shared dependencies. Consequently, a lower Euclidean Distance signifies a greater overlap in dependencies between the two repositories. Hence, we

use the Negated Euclidean Distance $-D_{ij}$ as our similarity score. This aligns with identifying digital siblings, as repositories with numerous shared dependencies are likely to be operating in similar problem domains.

XOR Similarity The XOR Similarity metric computes the proportion of shared dependency information between two binarized repository vectors and uses it as a similarity measurement. For two repository vectors v_i and v_j, it is defined as:

$$v_{xor} = v_i \oplus v_j \tag{8}$$

$$nonzero(v) = |\text{Non-zero entries in } v| \tag{9}$$

$$\text{Similarity} = 1 - \frac{nonzero(v_{xor})}{|v_{xor}|} \tag{10}$$

Since repositories sharing a higher number of dependencies will have more matching entries in their vectorized representation, this will lead to fewer non-zero entries in the v_{xor} vector. Consequently, a higher similarity score indicates more shared dependencies, which helps in identifying digital siblings.

AND Similarity The AND Similarity metric computes the proportion of dependencies used by both binarized repository vectors out of the total number of dependencies used. For two repository vectors v_i and v_j, it is defined as:

$$v_{and} = v_i \wedge v_j \tag{11}$$

$$nonzero(v) = |\text{Non-zero entries in } v| \tag{12}$$

$$\text{Similarity} = \frac{nonzero(v_{and})}{max(nonzero(v_i), nonzero(v_j))} \tag{13}$$

Repositories sharing a higher number of dependencies will have a higher number of non-zero entries in the v_{and} vector. Consequently, their similarity score will be high, which helps in identifying digital siblings. To ensure that the similarity is bound between 0 and 1, the score is divided by the largest number of dependencies used between the two repositories.

3.6 Comparing Clustering Techniques

To address the sub-question RQ3, we investigate applying various clustering techniques to the vectorized sets of repositories. We compare the clusters produced by the investigated techniques to the reference clustering - clusters corresponding to the manually labeled categories in our dataset. Our objective is to identify the clustering technique that best replicates the reference clustering. The investigated clustering techniques include Agglomerative Clustering, K-Means Clustering, and Density-Based Clustering.

Computing the Affinity Matrix We first compute an affinity matrix that stores a similarity score between every pair of repositories in our dataset. The similarity score is computed using the most effective similarity metric as found in 3.5.

Assuming we have a set of n repositories R, the affinity matrix A is an $n \times n$ symmetric matrix where each element a_{ij} represents the similarity score between repository r_i and repository r_j. The affinity matrix is computed as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \tag{14}$$

$$a_{ij} = \text{similarity}(r_i, r_j) \tag{15}$$

In this matrix, the diagonal elements (a_{ii}) represent the self-similarity of the repositories and are set to the maximum similarity score. The off-diagonal elements are calculated based on the chosen, most-effective similarity metric (e.g., Euclidean distance, XOR similarity, etc.).

Performance Metrics We evaluate each clustering technique using the Rand Index and the Normalized Mutual Information (NMI) score. These metrics compare the clusters produced by each technique to the reference clustering.

- The **Rand Index** (RI) measures the similarity between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and ground-truth clusterings. [Rand, 1971] Given clusterings, C and K, of a set of n elements, the Rand Index is defined as:

$$\text{RI}(C, K) = \frac{a + b}{\binom{n}{2}} \tag{16}$$

  where:
  - a is the number of pairs of elements that are in the same cluster in C and in the same set in K,
  - b is the number of pairs of elements that are in different clusters in C and in different clusters in K,
  - $\binom{n}{2}$ is the total number of possible pairs.

  The RI ranges from 0 (no agreement between predicted and ground truth clusters) to 1 (perfect agreement).

- The **Normalized Mutual Information** (NMI) score measures the shared information between two clusterings. [Vinh et al., 2010] Given clusterings, C and K, of a set of n elements, the NMI is calculated as follows:

$$\text{NMI}(C, K) = \frac{2 \times I(C, K)}{H(C) + H(K)} \tag{17}$$

  where:
  - $I(C, K)$ is the mutual information between clusterings C and K,
  - $H(C)$ and $H(K)$ are the entropies of clusterings C and K respectively.

  The mutual information $I(C; K)$ is given by:

$$I(C, K) = \sum_{c_i \in C} \sum_{k_j \in K} p(c_i, k_j) \log \frac{p(c_i, k_j)}{p(c_i)p(k_j)} \tag{18}$$

and the entropy of a clustering, $H(C)$, is calculated as:

$$H(C) = -\sum_{c_i \in C} p(c_i) \log p(c_i) \tag{19}$$

In these equations, $p(c_i)$ is the probability of a point belonging to cluster c_i in clustering C, $p(k_j)$ is the probability of a point belonging to cluster k_j in clustering K, and $p(c_i, k_j)$ is the joint probability of a point belonging to both clusters c_i and k_j.

The NMI Score ranges from 0 (no mutual information) to 1 (perfect correlation).

The technique with the highest RI and NMI scores is deemed the most efficient. It indicates that the clustering produced closely resembles the reference clustering, effectively separating the repositories by their problem domains.

The following sections describe the specifics of each of the investigated clustering techniques.

Agglomerative Clustering Agglomerative Clustering starts by treating each repository as a separate cluster, and then iteratively merges clusters based on the affinity matrix A and the specified target number of clusters. With each iteration, the two repositories with the highest similarity are merged into a cluster. Clustering is complete once the number of clusters is equal to k [Anderberg, 1973].

The key characteristics of this algorithm are its simplicity and the ability to utilize our pre-computed similarity metrics as the basis for merging adjacent clusters. However, the approach is best suited for hierarchical data structures which might impede performance.

In our implementation, we used the following parameters for the clustering algorithm:

- *Affinity Matrix*: We use the previously calculated affinity matrix A.
- *Number of Clusters (k)*: There are three distinct categories in our labelled dataset. Therefore $k = 3$.

K-Means Clustering K-Means Clustering divides data into k distinct, uniform clusters, based on the distances between data points. First, k centroids are initialized. Then, each repository is assigned to the cluster with the closest centroid. Subsequently, these centroids are recalculated iteratively to minimize the average distance within each cluster. [MacQueen, 1967]

The algorithm has been chosen for its simplicity and its wide use in literature for classifying vectorized data. K-Means is known to perform best on datasets that have uniformly shaped and sized clusters [Ikotun et al., 2023], which should make it especially suitable for our dataset.

In our implementation of K-Means, we used the following parameters:

- *Distance Matrix (D)*: We use the previously calculated affinity matrix A for this purpose. To treat it as a distance matrix, we transform each entry from a similarity score to a distance metric by negating it.

$$D = -A \tag{20}$$

- *Number of Clusters (k)*: There are three distinct categories in our labelled dataset. Therefore $k = 3$.

Density-Based Scan Clustering DBSCAN is distinct from K-Means and Agglomerative Clustering as it forms clusters based on the density of data points in the dataset. First, a random point is chosen as a centroid. If the point is located in a dense region, it will remain as a new cluster, otherwise, the algorithm continues iteratively to the next point. Clusters are merged based on the Epsilon ϵ parameter [Ester et al., 1996].

The focus on data point density makes DBSCAN outperform other clustering techniques on datasets where clusters are not uniform in their shape or size. It has been chosen to investigate the behavior of a more robust clustering technique on our dataset.

In our implementation of DBSCAN, we used the following parameters:

- *Affinity Matrix*: We use the previously calculated affinity matrix A.
- *Epsilon (ϵ)*: The parameter controlling the size of the computed clusters. The ϵ value of *0.5* was selected. This was done through trial and error aiming to have the algorithm generate the same number of clusters as the number of manually labeled categories in our training dataset.
- *Minimum Points (minPts)*: This number specifies the minimum number of points required to form a dense region, essential for a set of points to be identified as a cluster. We chose *minPts* as 5, based on the clustering behavior observed on the training dataset.

3.7 Adjusting for Composability

This research is carried out as a part of the RP group, where each researcher investigates a different characteristic for comparing GitHub repositories. These characteristics include source code identifiers, documentation, build configuration, GitHub metadata, and dependencies. The over-arching goal is to create a multi-modal metric for measuring similarity between GitHub repositories which combines all of these characteristics.

The similarity metrics discussed in 3.5 compare vectorized repositories (3.4) where each vector can consist of hundreds of entries, proportional to the number of unique dependencies in the dataset. This presents a challenge in composability, as such large vectors would dominate when combined with the similarity metrics proposed by the other RP group members. To address the sub-question RQ4, we investigate an approach to measuring repository similarity using dependency structures that can be seamlessly combined with other metrics.

Composable Approach We propose an approach that utilizes our previous work on clustering (3.6) and similarity metrics (3.5) as a means for dimensionality reduction producing a composable repository similarity metric. The approach involves constructing a pre-trained model M and using it to compress dependency information of unseen repositories into k-dimensional characteristic vectors. Subsequently, these characteristic vectors are used to measure similarity between repositories using Euclidean distance. The approach is outlined as follows:

- **Training / Validation Split**: Our dataset is split randomly into a training set and a validation set in an 80:20 proportion respectively to eliminate over-fitting biases.
- **Training**: The model M is trained by computing k clusters from the training dataset using the process outlined in 3.6. For each cluster, we extract a distinct centroid vector m_i. The computed centroids are then stored as a part of our model, $m_i \in M$.
- **Inference**: Given two unseen repositories r_i and r_j, with their dependency vectors v_i and v_j, we compute their respective characteristic vectors c_i and c_j.

  A characteristic vector c_i is constructed with k entries, each entry corresponding to the Euclidean distance between the repository dependency vector v_i and each centroid in the pre-trained model M.

$$c_i = \begin{bmatrix} dist(v_i, m_0) \\ dist(v_i, m_1) \\ ... \\ dist(v_i, m_k) \end{bmatrix} \tag{21}$$

$$dist(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{22}$$

  We then define the similarity metric between r_i and r_j as the Euclidean distance between their characteristic vectors c_i and c_j.

$$D_{i,j} = dist(c_i, c_j) \tag{23}$$

We theorize that repositories with similar dependency structures should have similar resulting characteristic vectors. Therefore, the distance between characteristic vectors can be used as a similarity metric.

In this approach, the similarity is computed between the characteristic vectors, each of size k, as opposed to the repository dependency vectors, which may vary in size based on the dataset used. This dimensionality reduction allows our approach to be composable with metrics of other RP group members. A characteristic vector of a multi-modal metric can be constructed by appending the characteristic vectors of two or more composable metrics.

Performance Metric To evaluate the performance of our composable approach, we use the previously introduced similarity difference factor (3). First, we train the model M using the training set. Then, we evaluate the performance by running inference on the evaluation set and calculating the SDF. We compare how using different similarity metrics (3.5) and clustering techniques (3.6) in the training step can impact performance. For Agglomerative and K-Means clustering, we run the evaluation with variable values of k: $k = 2$, $k = 3$, $k = 5$, $k = 10$, and $k = 15$

4 Evaluation

Our findings are presented in three parts. In section 4.1 we compare the performance of repository similarity metrics, as outlined in 3.5. Then in section 4.2 we compare the performance of repository clustering techniques, as outlined in 3.6. Lastly, in section 4.3 we evaluate the performance of our composable approach as outlined in 3.7.

4.1 Similarity Metrics

In this section, we answer *RQ2* by comparing the performance of similarity metrics introduced in 3.5. We compare how well each metric distinguishes between similar repositories (belonging to the same category) and those that are dissimilar (belonging to different categories), using the similarity difference factor (SDF) as the performance indicator.

Euclidean Distance	**deps-all**	**deps-direct**
Inner Similarity	-9.2536	-6.6766
Outer Similarity	-11.6626	-8.9789
SDF	1.2603	1.3448
XOR Similarity	**deps-all**	**deps-direct**
Inner Similarity	0.9652	0.9727
Outer Similarity	0.9486	0.9560
SDF	1.0175	1.0175
AND Similarity	**deps-all**	**deps-direct**
Inner Similarity	0.2946	0.3340
Outer Similarity	0.0987	0.1305
SDF	2.9842	2.5592

Table 1: Comparing the Components of the SDF of Repository Similarity Metrics

Similarity Metric	**Similarity Difference Factor**	
	deps-all	**deps-direct**
Euclidean Distance	1.2603	1.3448
XOR Similarity	1.0175	1.0175
AND Similarity	**2.9842**	2.5592

Table 2: Comparing the SDF of Repository Similarity Metrics

We observe in Table 2 that the AND Similarity metric has the highest SDF out of all the investigated metrics for both the *deps-all* and *deps-direct* sets of repositories. The SDF score of ~ 2.9842 indicates that, on average, the AND Similarity metric assigns three times higher similarity values when two repositories belong to the same reference cluster, as opposed to repositories belonging to different clusters. The metric is therefore highly effective at distinguishing digital siblings from dissimilar repositories.

The superior performance can be attributed to the fact that AND Similarity is size-agnostic. It focuses on the number of common dependencies between repositories, without being influenced by the total number of dependencies that each repository uses. This attribute makes the AND Similarity the most suitable in our context in which the dependencies are not guaranteed to be uniformly distributed among the repositories.

A surprising result is the emergence of set *deps-all* as the most effective for AND Similarity. This set includes all dependency information, both direct and transitive. Since transitive dependencies are not directly referenced by the project, we expected that including them might artificially elevate similarity scores for unrelated repositories, hence degrading the SDF. However, we can see that compared to set *deps-*

direct, including transitive dependencies improved the performance by 16.6%.

The effectiveness of AND Similarity in distinguishing digital siblings from dissimilar repositories validates that dependency structures are an effective characteristic for measuring inter-repository similarity.

4.2 Clustering Techniques

In this section, we answer *RQ3* by comparing the performance of repository clustering techniques introduced in 3.6. We compare which clustering techniques most effectively group repositories into clusters that reflect their manually labeled categories. The performance is evaluated using the Rand Index (RI) and the Normalized Mutual Information (NMI) score. Following the results from 4.1, we choose the AND Similarity metric and repository set *deps-all* as the basis for our investigation into clustering techniques.

Clustering Technique	RI	NMI
Agglomerative Clustering	0.2950	0.0216
K-Means Clustering	0.7484	0.5192
DBSCAN	**0.7898**	**0.5475**

Table 3: Comparing the Rand Index and Normalized Mutual Information of Repository Clustering Techniques

We observe in Table 3 that the DBSCAN algorithm has the highest RI and NMI scores out of all the investigated clustering techniques, followed closely by the K-Means algorithm. The RI of 0.7898 indicates that $\sim 79\%$ of items in the clustering constructed by DBSCAN match our reference clustering.

This was a surprising result, as we expected the K-Means algorithm to be the most effective. K-Means is known to perform the best on datasets with uniformly sized/shaped clusters. We know that the clusters in our dataset are uniformly sized as that was a pre-condition for our data collection (3.1). However, a possible explanation is that the clusters are not *shaped* uniformly in the multi-dimensional dependency space. In that case, the result would reflect the technical advantages of DBSCAN over the other investigated clustering methods. DBSCAN takes the density of the data into account, locating clusters around points of highest density. This allows the technique to perform better on datasets that are not uniformly distributed.

The ability to identify clusters that closely resemble the reference clusters validates that dependency structures are an effective metric for categorizing and querying large sets of repositories.

4.3 Composable Approach

In this section, we answer *RQ4* by evaluating the performance of the Composable Similarity metric introduced in 3.7. We compare how effective this similarity metric is at distinguishing between similar repositories (belonging to the same category), and those that are dissimilar (belonging to different categories), using the similarity difference factor (SDF) as the performance metric.

The Composable Similarity metric utilizes the previously introduced similarity metrics (3.5) and clustering techniques (3.6) in the *training* step. We choose to only use the AND Similarity for training the model M because of its superior performance as shown in 4.1. However, for the clustering technique, because of the similar performance results of DBSCAN and K-Means clustering (4.2), we choose to evaluate the Composable Similarity with both DBSCAN and K-Means used in the *training* step.

Clustering Algorithm	SDF of Composable Metric
K-Means ($k = 2$)	1.6818
K-Means ($k = 3$)	1.9180
K-Means (k = 5)	**2.1859**
K-Means ($k = 10$)	1.8917
K-Means ($k = 15$)	1.8901
DBSCAN ($k = 3$)	1.8944

Table 4: SDF of the Composable Repository Similarity Metric Trained Using Different Clustering Algorithms

We can see in Table 4 that using the K-Means clustering algorithm ($k = 5$) to train the Composable Similarity Metric results in the highest SDF value. The SDF score of ~ 2.1859 indicates that, on average, the Composable Similarity metric assigns two times higher similarity values when two repositories belong to the same reference cluster, as opposed to repositories belonging to different clusters. The metric is therefore highly effective at distinguishing digital siblings from dissimilar repositories.

This result is surprising, as we were expecting that training M using the DBSCAN clustering would result in the highest SDF performance. This is due to our previous results showing that DBSCAN produces the most accurate clusterings given our dataset (4.2). In this case, however, the K-Means algorithm emerges as the more effective approach for training M.

The effectiveness of Composable Similarity in distinguishing digital siblings from dissimilar repositories validates that dependency structures can be used effectively as a composable similarity metric. This allows for more accurate, multimodal similarity metrics to be developed in the future.

5 Discussion

We can now answer RQ1 by providing evidence showing that dependency structures of GitHub repositories can indeed be leveraged to find digital siblings of these projects.

- **Dependency structures can be used to measure the similarity between two projects.** We have shown this by finding an effective similarity metric based on dependency structures – the AND Similarity (4.1).
- **Dependency structures can be used to effectively categorize and query large sets of repositories.** This is shown in 4.2 by finding an effective clustering technique capable of grouping sets of digital siblings.
- **Similarity metrics based on dependency structures can be composed with the work of other RP group members.** This is shown by constructing and evaluating an effective, composable repository similarity metric based on dependency structures (4.3).

In this section, we reflect on the implications of our findings in the field of software development, the practical applications of our work, and outline potential directions for future research.

5.1 Impact on Software Development

Our research has the potential to greatly improve the day-to-day work of Open Source developers. By enabling programmers to identify GitHub repositories similar to their own, our approach fosters a more efficient exchange of code, solutions, and ideas. This leads to quicker learning, reduced redundancy in development, and higher reuse of robust, well-tested components. Consequently, our findings not only expedite the development process but can also enhance the overall quality of software projects. This contribution marks a step towards a more interconnected Open Source community.

5.2 Practical Applications

Optimization of CI/CD Pipelines Tools could be created to analyze highly efficient repositories similar to one's own, and automatically suggest optimizations for CI/CD configurations, improving efficiency and reliability in software deployment processes.

Automating Dependency Management and Security A system could be developed to automatically analyze and suggest updates or improvements to a project's dependencies based on the configuration of more mature, yet still thematically similar software projects. This would help maintain up-to-date and secure dependencies, reducing the risk of vulnerabilities.

5.3 Future Work

Longitudinal Studies on Dependency Evolution Conducting longitudinal studies to observe how dependency structures evolve in successful projects. This could provide insights into the lifecycle of software dependencies, guiding developers on when to adopt, update, or retire certain dependencies.

Quicker Dependency Extraction While our study revealed that the repository similarity metrics perform best on data sets including both transitive and direct dependencies (4.1), the difference in performance is small. If transitive dependencies are not considered, it is possible to create a more efficient dependency extraction pipeline that extracts dependency data directly from the Gradle and Maven files without needing to run the build pipeline. This has the potential to significantly improve the efficiency of the dependency extraction step, decreasing the time needed per repository, and hence allowing more repositories to be analyzed.

Expanding to Multiple Programming Languages and Ecosystems While our current study focuses primarily on Java repositories, extending the research to include multiple programming languages and their respective package ecosystems could provide a more comprehensive understanding of how dependency structures impact project similarity across diverse development environments.

6 Conclusions

The Open Source ecosystem provides a virtually infinite knowledge base for developers from all over the world. It is, however, challenging to find relevant information in this expansive network hosting millions of software projects. Finding projects similar to one's own allows for better developer collaboration, improves code reuse, and simplifies the implementation of boilerplate-heavy operations. The motivation behind this research was to allow Open Source developers to easily find similar GitHub repositories. We specifically tackled the challenge of quantifying the similarity between GitHub repositories, based on their dependency structures. Our primary objective was to find how dependency structures can be used to identify "digital siblings" – repositories that have similar goals or share the same problem domains.

The key research questions and conclusions were:

- **How can the dependency structures of GitHub repositories be leveraged to find their digital siblings?**: We have shown and evaluated the most effective similarity metrics and clustering techniques which can identify groups of digital siblings in a set of GitHub repositories.

  Our work paves a way for a large-scale approach capable of querying the entire GitHub database for digital siblings based on the dependency structures of projects.

- **What metrics, derived from analyzing dependency structures, most accurately quantify the similarity between GitHub repositories?**: Our findings highlighted the effectiveness of the AND Similarity metric (4.1). This metric proved superior in distinguishing repositories belonging to the same reference software category from the repositories belonging to other categories, using only their dependency information.

- **Which clustering methods are most effective in grouping GitHub repositories into clusters mirroring similar problem domains?**: The Density-Based Clustering (DBSCAN) method emerged as the most effective, accurately grouping repositories into clusters that mimic the reference set of clusters (4.2).

- **How can dependency structures as a similarity metric be composed with the similarity metrics investigated by the other RP group members?**: We built upon our investigation of repository similarity metrics and clustering techniques to propose a novel composable similarity metric (3.7). We evaluated this metric and found it to be highly effective at identifying digital siblings, as well as, being composable with the metrics proposed by the other RP group members (4.3).

Our research has the potential to greatly improve the way Open Source developers collaborate, share knowledge and co-create ideas. The presented findings can serve as a foundation for practical approaches such as automated optimization of CI/CD pipelines, or collaborative dependency management systems. This work also serves as a starting point for further research into dependency structure evolution, more efficient dependency extraction approaches, or investigating dependency structures of more software ecosystems.

7 Responsible Research

In this work, we have taken careful considerations to ensure the validity and reproducibility of our findings. Honesty, scrupulousness, transparency, independence, and responsibility are the core principles of the *Netherlands Code of Conduct for Research Integrity* [KNAW et al., 2018], that we strictly adhere to. This section addresses the potential risks to validity and reproducibility, and the measures taken to ensure that standards of responsible research are upheld in our study.

7.1 Ethical Data Collection

Ethical data collection concerns the way our dataset is constructed and the impact of it on the privacy and well-being of affected groups. The following steps were taken to ensure ethical data collection:

Use of Open Data All the data collected for this research is Open Source and publicly available. The repositories have been gathered using the public GitHub API, using the criteria outlined in 3.1. The licenses of the repositories in our dataset have been checked to ensure their fair use in our research.

No Personally Identifiable Information The repository data collected does not contain any personally identifiable information that could compromise privacy of any groups or individuals.

7.2 External Validity

External validity concerns the extent to which our results can be generalized to other contexts and datasets. The following threats to validity can be acknowledged:

Dataset Size Our study is based on a rather small dataset of 150 projects, representing three distinct software categories. This is due to the limited time and resources available for this study. To improve the generalizability of our findings, a larger dataset should be considered with a more varied set of labeled software categories.

Data Collection Criteria In our data collection step (3.1), we highlight that only repositories using the Java language and either the Maven or Gradle build systems will be included in our dataset. While this reduces the bias of the data stemming from differences in build systems and dependency ecosystems, it does harm the generalizability of our approach.

7.3 Reproducibility

Reproducibility concerns the extent to which other researchers can obtain the same results as the ones presented in this paper when using our dataset and evaluation code. The following steps were taken to ensure reproducibility:

Openly Accessible Dataset and Code We store our dataset, the dependency extraction code, and the code used to evaluate repository similarity metrics/clustering techniques, in a publicly accessible Zenodo repository under the DOI 10.5281/zenodo.10576708 [Rębacz, 2024]. Zenodo is a safe, trusted, and citeable platform for archiving research repositories, funded by the European Union and CERN [Zenodo, 2024]. This approach guarantees that the dataset, as well as, the code used in this research stays openly available to other researchers and allows for easier reproducibility. We encourage a collaborative environment and the growth of collective knowledge by inviting others to validate, reuse, and expand on our work.

Integrity of Results We present all the results of our investigation accurately with no outside modifications. Every table is an accurate representation of the data collected during the evaluation. If any data has been excluded, we have made note of it explicitly and explained our choice. This policy protects the integrity of our research and guarantees that the results accurately reflect our work.

Instructions for Reproducibility The methodology section (3) of our work contains detailed explanations of the steps we took to collect our data and produce the presented results. This makes it easy for the reader to follow our approach and arrive at the same results, which enhances the reproducibility of our research. The code published alongside this research contains a *README.md* file with additional explanations on how to compile the dependency extraction and evaluation modules [Rębacz, 2024].

References

[Anderberg, 1973] Anderberg, M. R. (1973). Chapter 6 - hierarchical clustering methods. In Anderberg, M. R., editor, *Cluster Analysis for Applications*, Probability and Mathematical Statistics: A Series of Monographs and Textbooks, pages 131–155. Academic Press.

[Ester et al., 1996] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press.

[Ikotun et al., 2023] Ikotun, A. M., Ezugwu, A. E., Abualigah, L., Abuhaija, B., and Heming, J. (2023). K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622:178–210.

[Kawaguchi et al., 2006] Kawaguchi, S., Garg, P. K., Matsushita, M., and Inoue, K. (2006). Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953.

[KNAW et al., 2018] KNAW, NFU, TO2-federatie, Hogescholen, V., and VSNU (2018). Nederlandse gedragscode wetenschappelijke integriteit. *DANS*.

[MacQueen, 1967] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations.

[McMillan et al., 2012] McMillan, C., Grechanik, M., and Poshyvanyk, D. (2012). Detecting similar software applications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE.

[Nguyen et al., 2018] Nguyen, P. T., Di Rocco, J., Rubei, R., and Di Ruscio, D. (2018). Crosssim: Exploiting mutual relationships to detect similar oss projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE.

[Rand, 1971] Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850.

[Rębacz, 2024] Rębacz, M. (2024). itsmatoosh/tudelft-bachelor-research-project: 1.0.

[Spinellis, 2012] Spinellis, D. (2012). Git. *IEEE Software*, 29(3):100–101.

[StackOverflow, 2023] StackOverflow (2023). Stack Overflow developer survey 2023.

[Vinh et al., 2010] Vinh, N. X., Epps, J., and Bailey, J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *J. Mach. Learn. Res.*, 11:2837–2854.

[Warner, 2018] Warner, J. (2018). Thank you for 100 million repositories.

[Zenodo, 2024] Zenodo (2024).

[Zhang et al., 2017] Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., and Sun, J. (2017). Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

A Dataset

https://github.com/filoghost/HolographicDisplays~mc-plugin
https://github.com/NoCheatPlus/NoCheatPlus~mc-plugin
https://github.com/ViaVersion/ViaBackwards~mc-plugin
https://github.com/ViaVersion/ViaVersion~mc-plugin
https://github.com/games647/FastLogin~mc-plugin
https://github.com/TownyAdvanced/Towny~mc-plugin
https://github.com/SkinsRestorer/SkinsRestorerX~mc-plugin
https://github.com/IntellectualSites/PlotSquared~mc-plugin
https://github.com/AuthMe/AuthMeReloaded~mc-plugin
https://github.com/Slimefun/Slimefun4~mc-plugin
https://github.com/SkriptLang/Skript~mc-plugin
https://github.com/DiscordSRV/DiscordSRV~mc-plugin
https://github.com/EngineHub/WorldGuard~mc-plugin
https://github.com/SaberLLC/Saber-Factions~mc-plugin
https://github.com/noahbclarkson/Auto-Tune~mc-plugin
https://github.com/Multiverse/Multiverse-Core~mc-plugin
https://github.com/MilkBowl/Vault~mc-plugin
https://github.com/GeorgH93/MarriageMaster~mc-plugin
https://github.com/MyPetORG/MyPet~mc-plugin
https://github.com/rlf/uSkyBlock~mc-plugin
https://github.com/elBukkit/MagicPlugin~mc-plugin
https://github.com/dmulloy2/ProtocolLib~mc-plugin
https://github.com/DRE2N/DungeonsXL~mc-plugin
https://github.com/GeorgH93/Minepacks~mc-plugin
https://github.com/lenis0012/LoginSecurity~mc-plugin
https://github.com/APDevTeam/Movecraft~mc-plugin
https://github.com/ChestShop-authors/ChestShop-3~mc-plugin
https://github.com/goncalomb/NBTEditor~mc-plugin
https://github.com/TheJeterLP/ChatEx~mc-plugin
https://github.com/garbagemule/MobArena~mc-plugin
https://github.com/PEXPlugins/PermissionsEx~mc-plugin
https://github.com/WooMinecraft/WooMinecraft~mc-plugin
https://github.com/CitizensDev/Citizens2~mc-plugin
https://github.com/PlayPro/CoreProtect~mc-plugin
https://github.com/nsporillo/GlobalWarming~mc-plugin
https://github.com/BentoBoxWorld/BentoBox~mc-plugin
https://github.com/cijaaimee/Slime-World-Manager~mc-plugin
https://github.com/jpenilla/MiniMOTD~mc-plugin
https://github.com/andrei1058/BedWars1058~mc-plugin
https://github.com/libraryaddict/LibsDisguises~mc-plugin
https://github.com/DownThePark/SetHome~mc-plugin
https://github.com/IkeVoodoo/LSSMP~mc-plugin
https://github.com/sgtcaze/NametagEdit~mc-plugin
https://github.com/Gecolay/GSit~mc-plugin
https://github.com/PlaceholderAPI/PlaceholderAPI~mc-plugin
https://github.com/Puharesource/TitleManager~mc-plugin
https://github.com/gonalez/znpcs~mc-plugin
https://github.com/yL3oft/zHomes~mc-plugin
https://github.com/crusopaul/OreRandomizer~mc-plugin
https://github.com/EnttbotX/ClansX~mc-plugin
https://github.com/cabaletta/baritone~mc-mod
https://github.com/Creators-of-Create/Create~mc-mod
https://github.com/xCollateral/VulkanMod~mc-mod
https://github.com/Wurst-Imperium/Wurst7~mc-mod
https://github.com/ReplayMod/ReplayMod~mc-mod
https://github.com/BluSunrize/ImmersiveEngineering~mc-mod
https://github.com/mezz/JustEnoughItems~mc-mod

https://github.com/PolyhedralDev/Terra~mc-mod
https://github.com/ldtteam/minecolonies~mc-mod
https://github.com/TerraformersMC/ModMenu~mc-mod
https://github.com/VolmitSoftware/Iris~mc-mod
https://github.com/TeamGalacticraft/Galacticraft~mc-mod
https://github.com/TechReborn/TechReborn~mc-mod
https://github.com/LambdAurora/LambDynamicLights~mc-mod
https://github.com/shedaniel/RoughlyEnoughItems~mc-mod
https://github.com/The-Aether-Team/The-Aether~mc-mod
https://github.com/TerraformersMC/Terrestria~mc-mod
https://github.com/LambdAurora/LambdaBetterGrass~mc-mod
https://github.com/TeamLapen/Vampirism~mc-mod
https://github.com/Coffee-Client/Coffee~mc-mod
https://github.com/TerraformersMC/Traverse~mc-mod
https://github.com/ValkyrienSkies/Valkyrien-Skies-2~mc-mod
https://github.com/mekanism/Mekanism~mc-mod
https://github.com/ForestryMC/ForestryMC~mc-mod
https://github.com/CaffeineMC/lithium-fabric~mc-mod
https://github.com/PorkStudios/FarPlaneTwo~mc-mod
https://github.com/PaperMC/Starlight~mc-mod
https://github.com/VazkiiMods/Botania~mc-mod
https://github.com/cc-tweaked/CC-Tweaked~mc-mod
https://github.com/Glitchfiend/BiomesOPlenty~mc-mod
https://github.com/iPortalTeam/ImmersivePortalsMod~mc-mod
https://github.com/TartaricAcid/TouhouLittleMaid~mc-mod
https://github.com/MrCrayfish/MrCrayfishFurnitureMod~mc-mod
https://github.com/Team-EnderIO/EnderIO~mc-mod
https://github.com/Team-RTG/Realistic-Terrain-Generation~mc-mod
https://github.com/AntiqueAtlasTeam/AntiqueAtlas~mc-mod
https://github.com/TwelveIterationMods/Waystones~mc-mod
https://github.com/YaLTeR/MouseTweaks~mc-mod
https://github.com/MattCzyr/NaturesCompass~mc-mod
https://github.com/squeek502/AppleSkin~mc-mod
https://github.com/vectorwing/FarmersDelight~mc-mod
https://github.com/TeamTwilight/twilightforest~mc-mod
https://github.com/AppliedEnergistics/Applied-Energistics-2~mc-mod
https://github.com/Shadows-of-Fire/Toast-Control~mc-mod
https://github.com/AlexModGuy/AlexsMobs~mc-mod
https://github.com/jaquadro/StorageDrawers~mc-mod
https://github.com/TwelveIterationMods/CraftingTweaks~mc-mod
https://github.com/progwml6/ironchest~mc-mod
https://github.com/Shadows-of-Fire/FastWorkbench~mc-mod
https://github.com/KyaniteMods/DeeperAndDarker~mc-mod
https://github.com/RoboBinding/RoboBinding~android-lib
https://github.com/k0shk0sh/PermissionHelper~android-lib
https://github.com/pedrovgs/Renderers~android-lib
https://github.com/jonfinerty/Once~android-lib
https://github.com/maurycyw/StaggeredGridView~android-component
https://github.com/Cutta/GifView~android-component
https://github.com/lyft/scissors~android-component
https://github.com/vekexasia/android-edittext-validator~android-component
https://github.com/YoKeyword/IndexableRecyclerView~android-component
https://github.com/takahirom/PreLollipopTransition~android-component
https://github.com/MrEngineer13/SnackBar~android-component
https://github.com/sharish/ScratchView~android-component
https://github.com/Meituan-Dianping/Shield~android-component
https://github.com/microsoftarchive/android-sliding-layer-lib~android-lib
https://github.com/miguelhincapie/CustomBottomSheetBehavior~android-component
https://github.com/matrixxun/PullToZoomInListView~android-component

https://github.com/florent37/AwesomeBar~android-component
https://github.com/lguipeng/BubbleView~android-component
https://github.com/nirhart/ParallaxScroll~android-component
https://github.com/jjobes/SlideDateTimePicker~android-component
https://github.com/iammert/StatusView~android-component
https://github.com/MasayukiSuda/FPSAnimator~android-lib
https://github.com/bboylin/UniversalToast~android-component
https://github.com/imkarl/CharacterPickerView~android-component
https://github.com/mabbas007/TagsEditText~android-component
https://github.com/romtsn/ArcNavigationView~android-component
https://github.com/Aspsine/FragmentNavigator~android-component
https://github.com/curioustechizen/android-ago~android-component
https://github.com/shehuan/NiceDialog~android-component
https://github.com/LineChen/FlickerProgressBar~android-component
https://github.com/shehabic/Droppy~android-component
https://github.com/wustor/GangedRecyclerview~android-component
https://github.com/nomanr/WeekCalendar~android-component
https://github.com/panpf/spider-web-score-view~android-component
https://github.com/Glamdring/EasyCamera~android-lib
https://github.com/f2prateek/progressbutton~android-component
https://github.com/klinker41/android-chips~android-component
https://github.com/takimafr/androidkickstartr~android-lib
https://github.com/heinrichreimer/material-drawer~android-component
https://github.com/yanzhenjie/Kalle~android-lib
https://github.com/lawloretienne/Trestle~android-lib
https://github.com/SimonVT/MessageBar~android-component
https://github.com/kobakei/Android-RateThisApp~android-lib
https://github.com/bgogetap/StickyHeaders~android-component
https://github.com/liuguangqiang/CookieBar~android-lib
https://github.com/baoyongzhang/Treasure~android-lib
https://github.com/saeedsh92/Banner-Slider~android-component
https://github.com/dbachelder/CreditCardEntry~android-component
https://github.com/feeeei/CircleSeekbar~android-component
https://github.com/Cutta/TagView~android-component