Context Survey

Author: 220032952

Supervisor: Rosa Filgueira and Lei Fang

June 2023

1 Introduction

Determining repository similarity has been a significant area of interest within the fields of computer science, software engineering, and data analysis. This has become even more pertinent in the era of distributed version control systems like Git and online platforms such as GitHub, and GitLab, which host millions of code repositories. In fact, this could be useful for a variety of other related tasks, such as identifying potential collaborations, analyzing code plagiarism, recommending libraries or frameworks, improving code reuse, and even informing strategies for project management. This survey aims to identify and review related works or methods that have been used to solve or address this problem.

2 Background

2.1 Similarity definition

For investigating the previous approaches to determining code similarity, it is extremely important to determine the scope of code similarity. As shown in the article [21], they divided code-fragment similarity into two categories:

- **Textual similarity**: 1) identical code fragments, 2) structural similarity (structurally identical, other features may be different, e.g. identifiers, types, etc.), and 3) approximate structural similarity (there may be operations such as add, delete statements, etc.)
- **Functional similarity**: As known as **semantic similarity**, it was defined that the functionalities of two code fragments are identical or similar, where the same operation or computation can be implemented by two or more code fragments.

This survey aims to focus on the methods used to determine semantic similarity.

2.2 Repository code

For determining the code similarity, most of the previous work is based on the comparison of the repository's code itself. Then it is really necessary to divide the repository's code more finely. As mentioned in article [15], dividing the code into three parts:

- **Code content**: The logical code content and structure.
- **Textual descriptions**: Associated methods, file names, and comments in the code, as known as *docstrings*.
- **Library dependencies**: The dependencies as a graph of functions calls within and across scripts or libraries.

2.3 Repository metadata

Aside from code, repositories also contain metadata. Based on the article [20], we will give a more detailed description of the metadata.

- **Descriptive information**: This includes the title of the repository, a brief description, the programming language used, the license type, tags or topics associated with the repository, and the *README* file content.
- **Version history**: This includes the commit history and the issue history of the repository. Each commit is an iteration to the repository and contains a description for describing the changes updated, the author of the commit, etc. Each issue means a bug tested by users or a discussion about the details of implementation and a process of integration with other projects, etc.
- **Contributor information**: This includes data about the contributors to the repository, such as their usernames, the number and nature of their contributions, and their roles within the project.
- **Activity information**: This includes data on the repository's activity, such as the number of issues opened and closed, pull requests made and merged, and the discussions in the comment sections of these issues and pull requests.
- **Social information**: On platforms like *GitHub*, this can also include data about stars, forks, watches, and followers of the repository, which can provide an indication of the repository's popularity and influence within the developer community.

2.4 Repository structure

Repository structure as mentioned in the paper [20] is also called project directory structure. Usually includes the following elements:

- **Root directory**: This is the top-level directory in the repository, which often contains *README* files, *LICENSE* files, and version-controlled files.
- **Source directory**: This directory usually contains the source code of the application.
- **Test directory**: This is where test codes are stored. It often mirrors the structure of the source directory, with each source file having a corresponding test file.
- **Documentation directory**: This directory contains additional documentation beyond what's in the *README*, such as user guides, API references, and design documents.
- **Auxiliary directories**: These directories are often covered by configuration files, deployment environment files, and dependency files.

3 Related works

3.1 Text-based techniques

Early works in the area of repository similarity focused primarily on text analysis techniques. For instance, using *Latent Semantic Analysis (LSA)* is a common technique for helping us to determine the repository similarity.

LSA, as known as *Latent Semantic Indexing (LSI)*, uses unsupervised machine learning to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text [7]. More specifically, *LSA* is based on the principle that words that are close in meaning will occur in similar pieces of text. A simplified explanation of how *LSA* works will be shown in the following:

- Create a *Term-Document Matrix (TDM)*: This matrix represents the frequency of terms that occur in a document, with rows representing unique words and columns representing each document [4].

- Apply a method called *Singular Value Decomposition (SVD)* to this matrix. *SVD* is a factorization technique in linear algebra that reduces the number of rows while preserving the similarity structure among columns. This results in a set of vectors that represent the documents and terms in a 'concept' space.
- By using this reduced matrix, documents are compared by taking the cosine of the angle between the two vectors (or the dot product between the normalisations of the vectors) formed by any two documents. Other correlations between the document vectors can also be used to ascertain similarity.

There are two previous works based on *LSA*:

- **MUDABlue** [13]: It is a project to determine repository similarity only researched on the repository code. Firstly, it extracts identifiers (such as variable names and function names) from source code and removes unrelated content (such as comments). Secondly, it considers identifiers as terms and software as documents in the *TDM*. Thirdly, it generates a reduced matrix by *SVD* and calculates the similarity.
- **CLAN** [16]: Similar to **MUDABlue**, it only uses the repository code as the base for researching the repository similarity. By contrast, it only uses API calls as semantic anchors to compute application similarity since API calls contain precisely defined semantics. Therefore, in the *TDM*, a row contains a unique class or package and a column corresponds to an application. By the way, the overall process is similar to **MUDABlue**.

3.2 Topic modeling techniques

Topic modeling is a form of statistical modeling for discovering the abstract "topics" that occur in a collection of documents. In the context of determining the similarity of repositories, each repository will be deemed as a document. A topic modeling algorithm like *Latent Dirichlet Allocation (LDA)*, assumes that each document (repository) is a mix of various topics, and each word is attributable to one of the document's topics [2]. Applying the *LDA* algorithm can output a set of topics (a group of words) and a weight distribution over these topics for each document [2]. In other words, each repository gets a vector representing the proportion of each topic in the repository. Finally, by computing the cosine similarity between these vectors, we can determine the similarity between repositories.

3.3 Network-based techniques

With the social coding aspects of platforms like GitHub, network-based techniques have been used to capture the interaction between repositories and developers. There are two previous works using network-based techniques:

- **RepoPal** [26]: In contrast to many previous works that are generally based on repository code, it takes only repository metadata as its input. Within this approach, there are three types of metadata to be considered: 1) *README* files, 2) starred information such as starred by users of similar interests, and 3) starred together by the same users within a short time gap. Thus, it integrates three ways of similarity computation into a final similarity computation: using the *cosine similarity* to calculate readme-based similarity, utilising the *Jaccard index*[12] to calculate stargazer-based similarity, and exploiting the time gap to calculate time-based similarity.
- **CROSSSIM** [18] [19]: It utilises both repository metadata and code information to measure the similarity between different software projects. It leverages a novel combination of graph kernels and text processing techniques to measure the similarity between software projects [18]. Graph kernels are used to capture the structural information of the software, such as the relationships between different methods and classes, whilst text processing techniques are used to capture the semantic information of the software.

3.4 Deep learning techniques

3.4.1 Embedding

An embedding, as known as *distribution representation*, is an unsupervised approach for conversing discrete categorical variables, such as words or images, into relatively low-dimensional continuous vectors by using a large training corpus [14] [17]. An efficient embedding can give a fixed vector representation and ensure that semantically similar entities have a closer distance between their vector spaces. There are different types of embedding depending on the data they represent, for instance:

- **Word embedding**: These models represent individual words as high-dimensional vectors, such that words with similar meanings are positioned close to each other in the embedding space, such as *word2vec* [17].
- **Document embedding**: These models represent entire documents or sentences as vectors. This is the extension of word embedding [17]. For example, *doc2vec* takes a document as input and maps it into a multiple-dimensional embedding vector while doing an agent task [14]. Noticeably, *doc2vec* has two types: 1) *Distributed Memory Model of Paragraph Vectors (PV-DM)*, which is similar to the *Continuous Bag of Words (CBOW)* model in the *word2vec*; 2) *Distributed Bag of Words version of Paragraph Vector (PV-DBOW)*, which is similar to the skip-gram model of *word2vec*.
- **Code embedding**: These models are the embedding approach applied to the code. For example, *code2vec* aims to capture the semantic meaning of code snippets, such as a method, similar to how *word2vec* captures the semantic meaning of words in the context of natural language processing [1]. In the *code2vec* model, code snippets are represented as *abstract syntax trees (ASTs)* [5]. The paths between nodes in these trees are extracted and encoded as path contexts. These path contexts are then processed by a neural network, typically a combination of an embedding layer and an attention mechanism, to generate a vector representation for the entire code snippet [5].
- **Graph embedding**: These represent nodes, edges, and entire graphs in graph-structured data as vectors. For example, at first, *node2vec* [9] converts the network structure into a set of paths using a sampling strategy (*Depth-First Sampling and Breadth-First Sampling*) for each node and then it trains a skip-gram model presented in *word2vec* according to these paths to obtain the vector representations for each node.

3.4.2 Recurrent Neural Network

A *Recurrent Neural Network (RNN)* is a type of artificial neural network well-suited to time series data or any sort of data where the previous input (or sequence of previous inputs) influences the future ones. Unlike feed-forward neural networks, *RNNs* have "memory", as known as *latent variables*, in the sense that the output for a given input is determined not just by the current input, but also by a series of previous inputs or a series of *latent variables* [25]. The idea behind *RNNs* is to make use of sequential information by performing the same task for every element in a sequence, with the output depending on the previous computations.

One of the biggest issues with simple *RNNs* is that they're not very good at capturing long-term dependencies in a sequence due to the so-called "vanishing gradient" problem, which makes it hard for *RNNs* to learn and tune the parameters of the earlier layers in the network. This is where two variants come into play, as they're able to keep the information in "memory" over longer periods of time.

- **Long Short Term Memory (LSTM)**: *LSTMs* are explicitly designed to avoid the long-term dependency problem [24]. They contain a cell state and three gates to control the flow of information: 1) Forget gate: Decides what information should be thrown away or kept, 2) Input Gate: Updates the cell state with new information, 3) Output Gate: Determines what the next hidden state should be [24].
- **Gated Recurrent Unit (GRU)**: Similar to *LSTMs*, *GRUs* are designed to have more persistent memory, thereby solving the vanishing gradient problem that traditional *RNNs* face. In contrast to *LSTMs*, *GRUs* have two gates to control the flow of data [3]: 1) Update Gate: This gate

determines how much of the previous state should be carried over to the current state. It is a combination of the input gate and the forget gate in an *LSTM*; 2) Reset Gate: This gate decides how much of the past information to forget.

3.4.3 Transformer

The *Transformer* model, proposed in the paper [23], offers a new approach to handling sequential data processing tasks, such as machine translation and text summarisation, by replacing recurrence with a self-attention mechanism. The *Transformer* model consists of an encoder and a decoder, each made up of several identical layers. The architecture of the transformer is shown in the figure 1.

- **Encoder**: The encoder takes a list of token embedding as inputs, and for each token, it computes a set of key, query, and value vectors (K, Q, V), using separate linear projections. These are used in a scaled dot-product attention mechanism. The output is a single vector that represents all the input tokens for that position. This operation is done for all positions in the sequence simultaneously, which allows for parallel computation.
- **Decoder**: The decoder operates similarly but with an additional 'masked' self-attention layer, which prevents positions from attending to subsequent positions, preserving the autoregressive property. The output from each step is then used as the query for the next step of the encoder-decoder attention.
- **Self-Attention**: The key innovation in the *Transformer* model is the self-attention mechanism, which weighs the importance of different positions in the input sequence when producing an output sequence.

3.4.4 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a groundbreaking model, which is introduced by researchers at Google AI Language in 2018 [6], and brings a revolution in the way of *NLP* tasks. The key components are the following:

- **Pre-training and Fine-tuning**: *BERT*'s training process consists of two steps. First, the model is pre-trained on a large corpus of text in an unsupervised manner. During this pre-training, the model learns internal representations of language by predicting masked words in a sentence *(Masked Language Model)*, and by predicting if a sentence comes after another one *(Next Sentence Prediction)*. Second, the pre-trained *BERT* model is fine-tuned on a specific task with labeled data. Fine-tuning only requires a few additional output layers to reshape *BERT* for the task, maintaining the pre-learned weights from the original model.
- **Transformer Architecture**: *BERT* relies on the *Transformer* model architecture, proposed in the paper [23]. This architecture uses self-attention mechanisms, allowing for more efficient and flexible handling of context.
- **Bidirectionality**: The fundamental breakthrough in *BERT* is its bidirectional nature. Unlike previous models that read text sequences in one direction, *BERT*, through the *Masked Language Model* objective, enables each word to directly influence the final embeddings on both sides, thus capturing the word's full context.

Here are three variants of *BERT*:

- **DistilBERT**: It is a smaller, faster, cheaper, and lighter variant of the *BERT* model, which obeys the *knowledge distillation* process to create a smaller model. *Knowledge distillation* is a type of learning where a smaller (student) model is trained to mimic the behavior of a larger (teacher) model [11]. During the training process of *DistilBERT*, it applied cosine embedding loss to the hidden states, encouraging the student's hidden states to be similar to the teacher's [22].

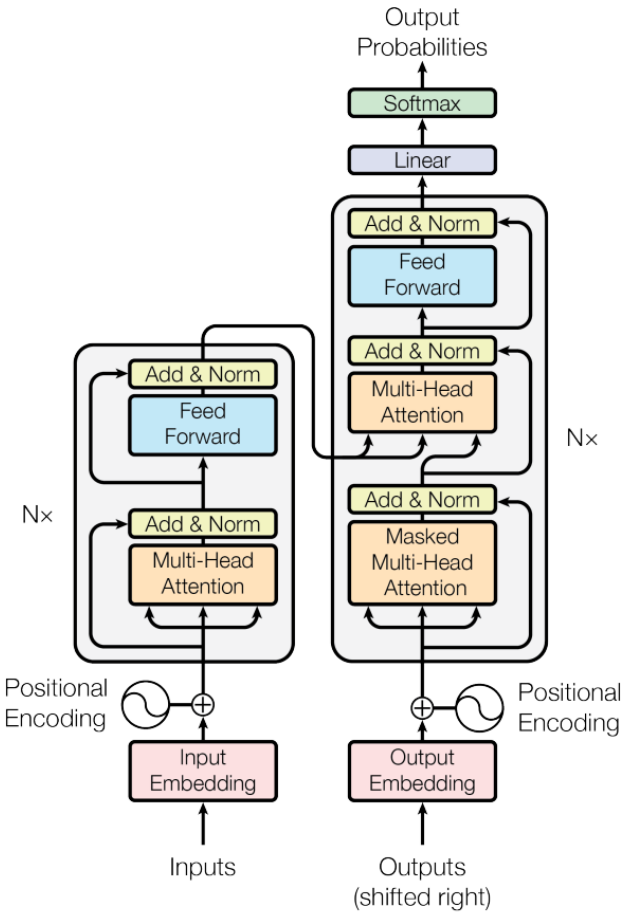

Figure 1: The Transformer architecture.

- **CodeBERT**: *CodeBERT* is a variant of the *BERT* model designed for programming and natural language. It is trained on a large-scale dataset consisting of codes in various programming languages and corresponding natural language annotations [8]. *CodeBERT* is unique in its dual goal of understanding natural language text and programming language, aiming to learn a general representation for bridging natural language and programming language.
- **GraphCodeBERT**: *GraphCodeBERT* [10] is an extension of the *CodeBERT* model, which uses the *Graph Transformer* model to learn the representation directly from the *AST*, rather than learning representations from the linearised *AST* of the code, which lacks the structural and semantic meaning that can be found in the original *AST*.

3.4.5 Example

In previous work, two particularly outstanding projects (researched so far) used deep learning techniques to solve the problem of determining repository code similarity.

- **Repo2vec** [20]: It offers a repository-level embedding, including metadata, structure, and code embedding. In more detail, it combines metadata information at first, then preprocesses this text information, and finally adopts the *doc2vec* approach to compute the metadata embedding vector. In the structure embedding phrase, it represents the directory structure into a tree representation, generates node vectors employing *node2vec*, and aggregates node vectors into a single structure vector. In the code embedding phrase, it computes the method code vectors (snippet code vectors) for each method in the source file, aggregates these method code vectors into a single file code vector, and aggregates all file code vectors into code embedding. In last integration phrase (repository embedding phrase), it uses a way representing three weights of

three embeddings to concatenate these embeddings to a repository-level embedding. By the way, it uses six aggregation methods (average, max, min, mode, sum, and standard deviation) for aggregation embedding, and finally finds mean aggregation function performs better than others. It uses cosine similarity to compute the repository similarity.

- **Topical** [15]: It only considers repository code embedding. However, the core of this project is the way of the integration of embeddings. This project divides repository code into three parts: 1) code content and structure, 2) textual information, and 3) dependency graph of code. Using *GraphCodeBERT* to generate embedding of the first part. Using *DistilBERT* to generate the other embeddings of the second and third parts. Finally, it is a distinguished method compared to the traditional method (concatenation), using a *GRU* layer and a self-attention mechanism layer (like a transformer architecture) to integrate these three embeddings into a new embedding. It also uses cosine similarity to compute the repository similarity.

4 Conclusion

In conclusion, determining repository similarity is a complex task that has been approached using various techniques, each with its own strengths and weaknesses.

Start with text-based techniques, including *Bag of Words*, *TF-IDF*, and *Latent Semantic Analysis*, which are powerful tools for repository similarity analysis, especially when considering documentation, commit messages, and code comments. However, they might fail to capture some structural information present in the code.

Secondly, topic modeling can provide a high-level overview of the themes or 'topics' present in a software repository, which can be a powerful tool for comparing repositories. But, it also requires careful tuning and interpretation.

Besides, graph-based techniques can capture some structural aspects of the code by treating it as a graph, where the nodes represent programming constructs and the edges represent relationships between them. While powerful, these methods can be more complex and computationally intensive.

Finally, deep learning techniques have shown promise in capturing both the semantic and structural aspects of code. Models like *CodeBERT* and *GraphCodeBERT*, use a transformer-based architecture to create meaningful vector representations of code. When applied to the task of repository similarity, they can provide more nuanced and semantic insights.

Overall, the choice of method depends heavily on the specific use case, the available computational resources, and the desired level of granularity. However, it is notable that such approaches, researching based on repository metadata (such as the contributors, the timestamps, the commits records, and the issue records), are relatively simple to implement but may miss out on deeper structural and semantic similarities within the codebase.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[3] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[4] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[5] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 243–253, 2020.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Nicholas E Evangelopoulos. Latent semantic analysis. *Wiley Interdisciplinary Reviews: Cognitive Science*, 4(6):683–692, 2013.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[9] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[11] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[12] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[13] Shinji Kawaguchi, Pankaj K Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. In *11th Asia-Pacific Software Engineering Conference*, pages 184–193. IEEE, 2004.

[14] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[15] Agathe Lherondelle, Yash Satsangi, Fran Silavong, Shaltiel Eloul, and Sean Moran. Topical: Learning repository embeddings from source code using attention. *arXiv preprint arXiv:2208.09495*, 2022.

[16] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 364–374. IEEE, 2012.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[18] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. Crosssim: exploiting mutual relationships to detect similar oss projects. In *2018 44th Euromicro conference on software engineering and advanced applications (SEAA)*, pages 388–395. IEEE, 2018.

[19] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of github repositories. *Software Quality Journal*, 28:595–631, 2020.

[20] Md Omar Faruk Rokon, Pei Yan, Risul Islam, and Michalis Faloutsos. Repo2vec: A comprehensive embedding approach for determining repository similarity. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 355–365. IEEE, 2021.

[21] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[22] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[24] Shudong Yang, Xueying Yu, and Ying Zhou. Lstm and gru neural network performance comparison study: Taking yelp review dataset as an example. In *2020 International workshop on electronic communication and artificial intelligence (IWECAI)*, pages 98–101. IEEE, 2020.

[25] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[26] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23. IEEE, 2017.