Mapping the Repository Landscape: Harnessing Similarity with `RepoSim` and `RepoSnipy`

Zihao Li
School of Computer Science
University of St Andrews
St Andrews, UK
zl81@st-andrews.ac.uk

Rosa Filgueira
School of Computer Science
University of St Andrews
St Andrews, UK
rf208@st-andrews.ac.uk

***Abstract*—The rapid growth of scientific software development has led to the emergence of large and complex codebases, making it challenging to search, find, and compare software repositories within the scientific research community. In this paper, we propose a solution by leveraging deep learning techniques to learn embeddings that capture semantic similarities among repositories. Our approach focuses on identifying repositories with similar semantics, even when their code fragments and documentation exhibit different syntax. To address this challenge, we introduce two complementary open-source tools: `RepoSim` and `RepoSnipy`. `RepoSim` is a command-line toolbox designed to represent repositories at both the source code and documentation levels. It utilizes the *UniXcoder* pre-trained language model, which has demonstrated remarkable performance in code-related understanding tasks. `RepoSnipy` is a web-based neural semantic search engine that utilizes the powerful capabilities of `RepoSim` and offers a user-friendly search interface, allowing researchers and practitioners to query public repositories hosted on GitHub and discover semantically similar repositories. `RepoSim` and `RepoSnipy` empower researchers, developers, and practitioners by facilitating the comparison and analysis of software repositories. They not only enable efficient collaboration and code reuse but also accelerate the development of scientific software.**

***Index Terms*—semantic similarity, code search, code understanding, embeddings, pre-trained language models, GitHub.**

I. Introduction

The advent of increasingly powerful and affordable computing has revolutionized scientific and scholarly discovery across diverse fields [1]. This transformation is closely tied to the rapid growth of software, which has given rise to expansive and intricate codebases comprising thousands of source code files [2]. Within the scientific research community, researchers are increasingly sharing repositories of code related to their research papers, aiming to amplify the reach and influence of their work [3].

The ability to detect similar repositories is of paramount importance in various aspects of scientific research. Firstly, it facilitates code reuse, allowing researchers to leverage existing solutions and build upon them, leading to increased productivity and efficiency [4]. Additionally, identifying alternative implementations of algorithms and methods can aid in validation and comparison of results, promoting robustness and transparency in research practices [5]. Exploring related research projects through repository comparison offers significant advantages to researchers, Research Software Engineers (RSEs), and the scientific community. By engaging in repository comparison, researchers can obtain valuable insights, discover innovative approaches, promote knowledge exchange, and foster collaborations within the scientific community [6] . This practice aligns with the 'collaboration' pillar of Research Software Engineers, facilitating the sharing of best code practices [7]. As well, repository comparison also plays a crucial role in enhancing software citation [8]. Researchers have the opportunity to identify relevant work and projects in their field, leading to a more comprehensive and interconnected software ecosystem. By recognizing and citing these related repositories, researchers contribute to the acknowledgment and visibility of the underlying software projects that support their research endeavors.

Moreover, the detection of code theft and plagiarism is a critical concern in the scientific research landscape. Comparing repositories can help identify instances of inappropriate code reuse and ensure the integrity of scientific contributions [9]. Furthermore, rapid prototyping and finding projects for collaboration are facilitated by the ability to locate repositories with similar functionality and research objectives. However, searching for, finding,and comparing software from scientific communities can be a challenging task [10].

To address the challenge of effectively representing and comparing software repositories, this study explores the application of deep learning techniques to learn embeddings that capture semantic similarities among repositories. Our focus is on identifying repositories with similar semantics, even when their code fragments and code documentation exhibit different syntax. While our current investigation is focused on Python repositories, our approach is extensible to other languages.

In this paper, we propose two complementary open-source solutions. The first solution, `RepoSim`, is a command-line toolbox that leverages embeddings to represent Python repositories at two levels: source code and documentation (docstrings). The second solution, `RepoSnipy`, is a web-based neural semantic search engine built upon the capabilities of `RepoSim`. By utilizing `inspect4py` [11], a static code analysis framework, `RepoSim` automatically extracts key features from Python software, including code and docstrings. To generate our embeddings, we have selected the pre-trained language model *UnixCoder* [12], which has shown superior

performance in various code understanding tasks such as clone detection and code search.

By leveraging `RepoSim` and `RepoSnipy`, researchers and developers can benefit from accelerated software development in several ways. Firstly, these tools facilitate efficient collaboration by allowing users to discover and explore existing repositories, which can serve as valuable resources and references for new projects. Secondly, by building upon existing solutions and leveraging shared code, developers can significantly reduce development time and effort, accelerating the overall software development process. Reusing pre-existing code also ensures the reuse of reliable and tested implementations, leading to faster and more robust software development.

The remainder of the paper is structured as follows. Section II presents background on technologies relevant for this work. Section III details the features of the `RepoSim` command-line tool. Section IV gives an overview of the main features of the `RepoSnipy` neural search engine. Finally, section V summarises related work, and section VI concludes with a summary of achievements and future work.

II. Background

In the following sections, we look into the background work that is inherently established in this paper.

A. `inspect4py`

`inspect4py` [11] is a robust static code analysis framework specifically designed to facilitate comprehension of Python software repositories. By analyzing and extracting crucial information such as functions, classes, methods, documentation, dependencies, call graphs, and control flow graphs, inspect4py empowers developers to gain a deeper understanding of the repository's structure. It enables us to identify the software type (library, package, service) and its usage (software invocation), thus enhancing repository comprehension. `inspect4py` parses repositories into Abstract Syntax Trees (ASTs), and enhaces machine readability by extracting:

- **Documentation for functions and methods**
- **Source Code** and AST of functions & methods
- Call list
- **Licence(s)**, Readme, **Starts, Topics**
- File hierarchy
- Dependencies and requirements
- Analyze the previous data with heuristics for test detection, software type, and invocation

Using `inspect4py`, `RepoSim` leverages its capabilities to extract essential components from a given repository. This includes code and docstrings of functions and methods, as well as other pertinent details such as licenses, topics, and stars. In Section III we delve into the detailed process of how `RepoSim` uses `inspect4py`. Note that topics are labels that create subject-based connections between GitHub repositories and let developers explore projects by type, technology, and more. While a star in GitHub signifies the external recognition and interest of a repository from other users.

B. Language Models and Transformers

Natural language processing models have revolutionized computers' capabilities to read, speak, and comprehend human languages. Among the state-of-the-art models, the Transformer architecture has demonstrated remarkable advancements [13]. However, the scope of these models extends beyond human language and can be expanded to incorporate abstract syntax trees, enabling them to understand and compare code.

C. Code Search: Cross-Encoder and Bi-Encoder Paradigms

Code search encompasses the process of querying a corpus of code samples to retrieve relevant codes based on either *code-to-code* or *text-to-code similarity*. In *code-to-code* search, the query itself is a code sample, while in *text-to-code* search, the query is a natural language sentence. The objective is to identify matching codes in the search corpus that correspond to the query. This process is commonly categorized into two tasks: the code-search task for *text-to-code* search and the clone-detection task for *code-to-code search*. The code-search task aims to locate codes in the search corpus that are relevant to the given natural language query, while the clone-detection task focuses on identifying code clones within the search corpus that resemble the provided code sample query

There are mainly two paradigms of model architecture in existing deep learning code search methods: *cross-encoder* and bi-encoder [14]. As described in Figure 1, *cross-encoders* perform full- attention over the input pairs of query and code, while *bi-encoders* map each input (query or code) independently into a dense vector space. Note that in Figure 1, the query can be both, a code sample (clone-detection task) or a natural language sentence (code-search task).

Bi-enconders calculates the embeddings for both, query and code, which can be stored. While *cross-enconders* do not generate embeddings for those, they produce an output value between 0 and 1 indicating their similarity.

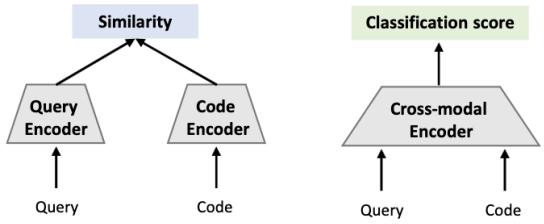


Fig. 1: The concept diagram of *bi-encoder* and *cross- encoder* code search architecture. *Bi-encoder models* are fast as the code embeddings can be pre-calculated offline. While *cross-encoder* models perform full-attention over the input pair of query and code, which could gain more information.

The trade-off of efficiency and effectiveness also exist between these two paradigms: *cross-encoders* achieves better accuracy than *bi-encoders*. As the code embeddings can be pre-calculated and stored, *bi-encoders* are more efficient than *cross-encoders*. However, for many applications, including

repository similarity, *cross-encoders* are not practical as they cannot produce separate embeddings for effective comparing.

In summary, *bi-encoder* architecture is fast, but less accurate, while *cross-Encoder* is more accurate, but slow. Given, that we are interested to compare the similarity or repositories based in their functions code and docstrings, which might contain large number of functions we have selected the *bi-encoder* architecture for this work. In the future, we plan to combine both strategies.

D. UnixCoder Language Model

UnixCoder [12] is a transformer-based model specifically designed to convert Abstract Syntax Trees (ASTs) into sequence text representations [15]. In order to enhance the semantic representation of code fragments through embeddings, the authors introduced two pre-training tasks: multi-modal contrastive learning (MCL), which leverages ASTs to acquire comprehensive code semantic representations, and cross-modal generation (CMG), which aligns embeddings across different programming languages using code comments.

Through rigorous experimentation presented in the paper [12], *UnixCoder* surpasses previous state-of-the-art models, including CodeBERT [16], GraphCodeBERT [17], SYNCOBERT [18], and CodeT5 [19], in various code-related understanding tasks such as code search and clone detection. Given *UnixCoder's* exceptional performance in extracting code semantic information and its versatility across multiple programming languages, we have chosen it as the model of choice for generating code and docstring embeddings within `RepoSim` (see Sections III-B and III-C).

III. RepoSim Toolbox

This work focuses on addressing the challenge of comparing software repositories that share a similar objective but may have different implementations. While their source code and documentation (docstrings) may exhibit different syntax, they possess similar semantics. Thus, we propose comparing repositories based on their source code and docstrings.

We present in this paper `RepoSim`[1], a novel deep-learning command-line tool. It is designed to assess the semantic similarity of repositories by analyzing their code and docstrings. `RepoSim` operates through the following steps:

1) Utilizing `inspect4py`, it extracts the source code and docstrings of all functions and methods within a repository. Additionally, it gathers relevant metadata information such as topics, stars, and licenses
2) By employing a fine-tuned *UniXCoder* language model (see Section III-B) it obtains code embeddings for each function and method (*code-level*). These embeddings are then averaged to compute the *code-mean* embedding representing a repository at source code level.
3) Leveraging a fine-tuned *UniXCoder* language model (see Section III-C) it generates embeddings for each docstring (*docstring-level*). Then it computes the *docstring-mean* embedding, which represents a repository at documentation level.
4) To determine the similarity scores between two repositories, `RepoSim` computes the cosine similarities of their code and docstrings.

[1]https://github.com/RepoAnalysis/RepoSim

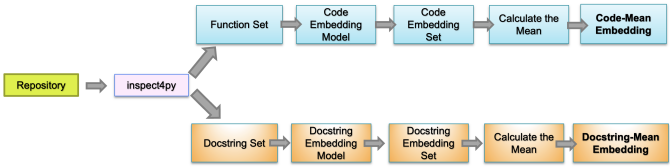


Fig. 2: `RepoSim` general model diagram including both embeddings representations: based on code and docstrings.

The diagram illustrating the `RepoSim` model can be seen in Figure 2. It is important to note that `RepoSim` only encodes each function and docstring once, enabling swift comparisons between repositories.

Figure 3 illustrates an example usage of `RepoSim` for comparing two or more repositories. The tool accepts GitHub repositories as input in the format of <owner>/<repo> (e.g., keon/algorithms), allowing the comparison of multiple repositories simultaneously. `RepoSim` generates a pickle file, <output dir>/output.pkl, which contains essential information such as the repositories' names, topics, licenses, and stars. Additionally, it includes the code and docstrings embeddings for each repository. Moreover, when the '–eval' flag is specified, RepoSim saves a csv file, <output dir>/eval_res.csv, which presents the similarity scores between pairs of repositories at both the code and docstrings levels. This file provides a comprehensive overview of the similarities across the repositories.

```
python repo_sim.py --input keon/algorithms
prabhupant/python-ds --output ./outputdir --eval
```

Fig. 3: `RepoSim` usage example to compare two repositories: `keon/algorithms` [2] and `prabhupant/python-ds`[3].

After generating the *code-mean* and *docstring-mean* embeddings of a repository using RepoSim, these embeddings can be visualized using the 2D t-SNE [20] algorithm. t-SNE is a dimension reduction technique that maps high-dimensional embeddings into two dimensions, enabling visualization.

Figure 4 illustrates the projection of *code-mean* embeddings for a collection of Python repositories, created with `RepoSim`. By examining the visualization, we can observe that repositories belonging to the same topic tend to cluster together, indicating their proximity in the embedding space. This visualization provides insights into the semantic relationships and similarities among repositories based on their code semantics.

A. Fine-tuning UnixCoder Models

In this work, our focus lies on the fine-tuned *Unixcoder* models for code-related understanding tasks such as code-search and clone-detection, as presented in [12]. However, the

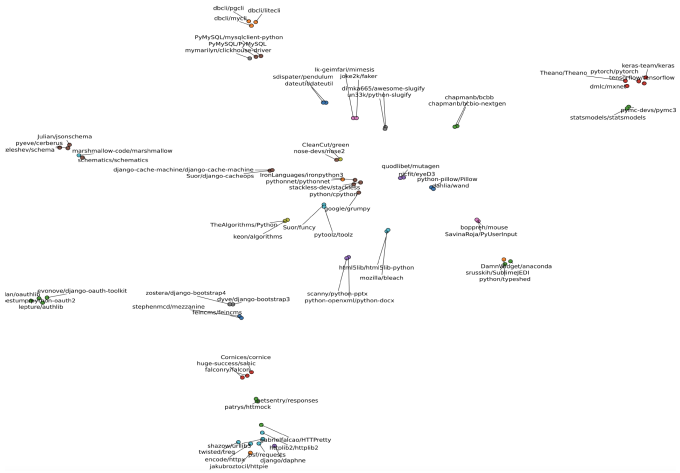


Fig. 4: 2D TSNE projection of repositories `RepoSim` *code-mean* embeddings. Markers represent repositories. Colors represent topics (e.g. algorithms, security, cli tools, etc.).

base model provided by the authors through Hugging Face [4] lacks any fine-tuning tasks. To address this, we conducted our own fine-tuning process following the instructions outlined in [5], utilizing the AdvTest dataset [21]. The AdvTest dataset comprises 280,634 pairs of (documentation, function) sourced from CodeSearchNet [22]. Notably, the dataset normalizes Python function and variable names to enhance the testing of model understanding and generalization capabilities.

Our fine-tuning efforts yielded two models: *unixcoder-code-search* and *unixcoder-clone-detection*. Each model required approximately 6 hours to complete the fine-tuning process on an NVIDIA A40 GPU server.

B. Code-level Embeddings

To determine the most suitable model for generating *code-level* embeddings in `RepoSim`, we conducted a performance comparison (see Section III-E)) among our *UnixCoder* fine-tuned models. Based on the results of this evaluation, we selected the fine-tuned *UnixCoder* model specifically designed for code search (*unixcoder-code-search*). `RepoSim` utilizes the mean of these embeddings to generate a *code-mean* embedding. The resulting embedding has a dimensionality of 768, capturing the essence of the repository's code representation.

C. Docstring-level Embeddings

In the process of selecting the appropriate model for generating *docstring-level* embeddings in `RepoSim`, we explored a variety of well-known and high-performing pre-trained sentence transformer models (refer to Table I). Through performance evaluation (see Section III-E), we found that the *UnixCoder* fine-tuned model, specifically developed for the code search task, demonstrated the best performance for *docstring-level* embeddings. Given its effectiveness, we have selected the *unixcoder-code-search* model to generate *docstring-level* embeddings in `RepoSim`. Following a similar approach as with *code-level* embeddings, `RepoSim` utilizes this model to create docstring embeddings for each repository. By computing the mean of these embeddings, a single docstring embedding (*docstring-mean* embedding) is generated to represent the repository at the documentation level, with a dimensionality of 768.

D. `RepoSim` Hugging Face Pipeline

We have also created a new `RepoSim` Hugging Face pipeline [6], that implements the steps outlined in Figure 2.

```python
from transformers import pipeline

model = pipeline(model="Lazyhope/RepoSim", trust_remote_code=True)
```

(a) Initialisation of the pipeline

```python
repo_infos = model("lazyhope/python-hello-world")
print(repo_infos)
```

(b) Specification of one (or multiple) repositories. Here we want to generate the embeddings for `lazyhope/python-hello-world` repository.

```
[{'name': 'lazyhope/python-hello-world',
  'topics': [],
  'license': 'MIT',
  'stars': 0,
  'code_embeddings': [["def main():\n    print('Hello World!')",
    [-2.0755109786987305,
     2.813878297805786,
     2.352170467376709, ...]]],
  'mean_code_embedding': [-2.0755109786987305,
   2.813878297805786,
   2.352170467376709, ...],
  'doc_embeddings': [['Prints hello world',
    [-2.3749449253082275,
     0.5409570336341858,
     2.2958014011383057, ...]]],
  'mean_doc_embedding': [-2.3749449253082275,
   0.5409570336341858,
   2.2958014011383057, ...]}]
```

(c) Output generated by our pipeline

.

Fig. 5: `RepoSim` *Hugging Face pipeline.*

Hugging Face pipelines provide a user-friendly approach to utilize models hosted on the *Hugging Face Model Hub*, encapsulating the necessary components for text inference within a simple interface. Specifically, this pipeline allows for the automatic generation of code and docstring embeddings for any Python repository on GitHub. It can be employed independently and is currently utilized by the `RepoSim` command-line tool. Figure 5 showcases an example of how to effortlessly generate embeddings using this pipeline for GitHub Python repositories. The process involves an initialization step (as depicted in 5a) to load our model `LazyHope/RepoSim`,

[4]unixcoder-base

[5]https://github.com/microsoft/CodeBERT/tree/master/UniXcoder/downstream-tasks/code-search#1-advtest-dataset-2

[6]https://huggingface.co/Lazyhope/RepoSim

and subsequently use it (as shown in Figure 5b) to generate repository embeddings, along with other pertinent information, as illustrated in Figure 5c.

E. `RepoSim` *Evaluation*

We conducted an extensive evaluation of various models to determine their suitability for integration into `RepoSim`. For *code-level* embeddings, we selected the two *UnixCoder* models fine-tuned in this study, namely *unixcoder-code-search* and *unixcoder-clone-detection* (see Section III-B). For *docstring-level* embeddings, we not only chose these two models but also included the following ones: *all-mpnet-base-v2* [7], *all-distilroberta-v1* [8], *paraphrase-multilingual-mpnet-base-v2* [9], *allenai-specter* [10], *gsarti/scibert-nli* [11], and *pritamdeka/S-Scibert-snli-multinli-stsb* [12]. These are pre-trained language models designed to generate document-level embeddings.

To evaluate the performance of these models, we utilized the *awesome-python* [23], which comprises over 500 Python repositories classified into different topics such as Algorithms, Audio, Authentication, Job Scheduler, Natural Language Processing, and Machine Learning. It is important to note that each repository in this dataset has been assigned a single topic for categorization purposes.

	repo1	repo2	topic1	topic2
0	boto/boto3	gorakhargosh/watchdog	Third-party APIs	Files
1	boto/boto3	zoofIO/flexx	Third-party APIs	GUI Development
2	boto/boto3	paramiko/paramiko	Third-party APIs	Cryptography
3	boto/boto3	antocuni/pdb	Third-party APIs	pdb-like Debugger
4	boto/boto3	martinrusev/imbox	Third-party APIs	Mail Clients
...	...	...	...	...
96575	mindsdb/mindsdb	knipknap/SpiffWorkflow	Machine Learning	Job Scheduler
96576	mindsdb/mindsdb	idan/oauthlib	Machine Learning	OAuth
96577	jonathanslenders/python-prompt-toolkit	knipknap/SpiffWorkflow	Interactive Interpreter	Job Scheduler
96578	jonathanslenders/python-prompt-toolkit	idan/oauthlib	Interactive Interpreter	OAuth
96579	knipknap/SpiffWorkflow	idan/oauthlib	Job Scheduler	OAuth

Fig. 6: Dataframe with pairs of *awesome-python* repositories and their topics. *topic1* column represents the topic of *repo1*, while *topic2* column represents the topic of *repo2*.

Figure 6 displays a dataframe containing pairs of repositories from our dataset, along with their respective topics. Specifically, the first five rows showcase the `boto/boto3` repository[13], labeled with the topic *Third-party Apis*, alongside other repositories and their corresponding topics.

To evaluate the effectiveness of different models in generating code and docstring embeddings, we conducted a series of experiments [14] in which we modified `RepoSim` accordingly. For these experiments, we utilized the *awesome-python* dataset as the ground truth for determining semantically similar repositories. If two repositories are categorized under the same topic, their embeddings should exhibit similarity, with their cosine similarity approaching one.

Figure 7 provides an example, illustrating the computed cosine similarities between repository pairs (e.g., `boto/boto3` and `gorakhargos/watchdog` in row zero) using different models. Since both *UnixCoder* fine-tuned models are employed at the code and docstring levels, we distinguish their cosine similarity results as follows: *unixcoder-code-search-funcs* and *unixcoder-clone-detection-funcs* for *code-level* similarity; and *unixcoder-code-search-docs* and *unixcoder-clone-detection-docs* for *docstring-level* similarity.

	repo1	repo2	topic1	topic2	all-mpnet-base-v2	all-distilroberta-v1	paraphrase-multilingual-mpnet-base-v2	allenai-specter	gsarti/scibert-nli	pritamdeka/S-Scibert-snli-multinli-stsb	unixcoder-code-search-funcs	unixcoder-code-search-docs	unixcoder-clone-detection-docs	unixcoder-clone-detection-funcs
0	boto/boto3	gorakhargosh/watchdog	Third-party APIs	Files	0.506485	0.527854	0.734368	0.974212	0.931573	0.781370	0.406343	0.101407	0.808835	0.793002
1	boto/boto3	zoofIO/flexx	Third-party APIs	GUI Development	0.515931	0.581671	0.716118	0.958938	0.926257	0.822463	0.428869	0.183302	0.809811	0.879089

Fig. 7: Dataframe with pairs of *awesome-python* repositories and their cosine similarities using different models.

To ensure the performance of the evaluated models, we employed the Receiver Operating Characteristic (ROC) curve and Area Under Curve (AUC) metrics [24]. The ROC curve provides a graphical representation of the classification model's performance at different classification thresholds. It plots the true positive rate (TPR) on the y-axis against the false positive rate (FPR) on the x-axis. The AUC score, on the other hand, quantifies the degree of separability achieved by the model. A higher AUC score indicates a better ability to predict true positives and true negatives

Figure 8 depicts the ROC curves for the selected models, demonstrating that the *UnixCoder* model fine-tuned on code search (*unixcoder-code-search-funcs*) and docstrings (*unixcoder-code-search-docs*) outperform the others. Table I provides an overview of the corresponding AUC scores for these models at both levels. The evaluation results clearly indicate that our *UnixCoder* model, fine-tuned on the code search task, exhibits superior performance in comparing similarities between repository source code and documentation.

Upon closer analysis, the superiority of the *unixcoder-code-search* model over the *unixcoder-clone-detection* model in detecting code and docstrings repositories similarities can be attributed to their divergent training objectives. The *unixcoder-code-search* model undergoes fine-tuning specifically for code search tasks, which encompass a wider range of similarity. This model is trained to comprehend the semantic nuances of

[7]https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[8]https://huggingface.co/sentence-transformers/all-distilroberta-v1

[9]https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2

[10]https://huggingface.co/allenai/specter

[11]https://huggingface.co/gsarti/scibert-nli

[12]https://huggingface.co/pritamdeka/S-Scibert-snli-multinli-stsb

[13]https://github.com/boto/boto3

[14]https://github.com/RepoAnalysis/RepoSim/blob/main/notebooks/BiEncoder/Embeddings_evaluation.ipynb

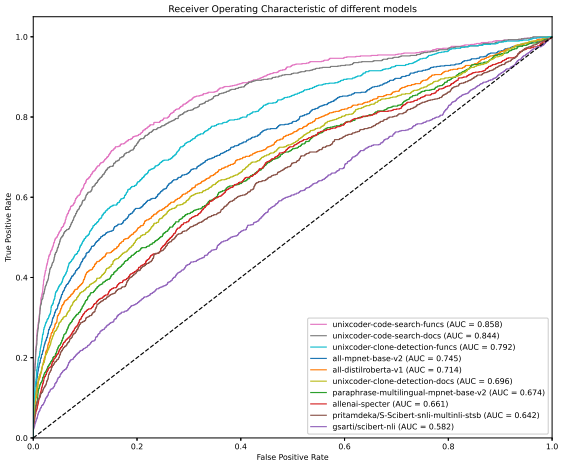


Fig. 8: ROC curves for repository similarity comparison. Each line represent a model from Table I.

code, contributing to its exceptional performance. These factors collectively bolster its capacity to identify resemblances among diverse codebases, rendering it more appropriate for `RepoSim`'s code and docstring similarity detection tasks. As a result, we have selected the *unixcoder-code-search* model for generating *code-level* and *docstring-level* embeddings (steps 2 and 3) in `RepoSim`, including our `RepoSim` Hugging Face pipeline (see Section III-D).

Model	AUC	
	Docstring	Code
gsarti/scibert-nli	58.2	—
pritamdeka/S-Scibert-snli-multinli-stsb	64.2	—
allenai-specter	66.1	—
paraphrase-multilingual-mpnet-base-v2	67.4	—
all-distilroberta-v1	71.4	—
all-mpnet-base-v2	74.5	—
unixcoder-clone-detection	79.2	69.6
unixcoder-code-search	**84.4**	**85.8**

TABLE I: AUC models scores at code and docstrings levels.

In Figure 9, we showcase the outcomes achieved by employing the selected model in `RepoSim`, revealing the top five repositories with the highest similarity scores identified from our dataset at both the code and docstring levels. As depicted in the figure, the repositories `lepture/authlib` and `idan/oauthlib` exhibit the highest similarity score. These repositories share numerous common features, indicating significant similarities:

- Similar Functionality: Both `lepture/authlib` and `idan/oauthlib` are Python libraries that provide functionality related to OAuth implementation. They offer features such as handling OAuth requests, managing OAuth clients and servers, and supporting various OAuth versions and extensions. This similarity in functionality increases the likelihood of similar code and docstrings
- Shared Domain Knowledge: The developers of `lepture/authlib` and `idan/oauthlib` likely have a deep understanding of OAuth and related concepts. They possess domain-specific knowledge, best practices, and standards for implementing OAuth functionality. This shared expertise results in similar code patterns, design decisions, and docstring conventions across both repositories.
- Common Design Patterns: OAuth implementation often involves following established design patterns and specifications. Both repositories may adopt similar design patterns and adhere to OAuth standards, leading to comparable code structures and docstring formats. This shared adherence to industry conventions contributes to the high similarity scores.
- Overlapping Dependencies: `lepture/authlib` and `idan/oauthlib` may rely on similar third-party libraries and dependencies. These shared dependencies can result in similar code patterns, usage of common libraries, and even similar docstring formats. The presence of overlapping dependencies increases the likelihood of code and docstring similarities.

Therefore, given the similarities in functionality, domain knowledge, design patterns, and dependencies between `lepture/authlib` and `idan/oauthlib`, `RepoSim` successfully identifies these commonalities, resulting in high similarity scores for both code and docstrings.

	repo1	repo2	topic1	topic2	code_sim	doc_sim
0	lepture/authlib	idan/oauthlib	OAuth	OAuth	0.936125	0.959024
1	evonove/django-oauth-toolkit	idan/oauthlib	OAuth	OAuth	0.922530	0.932178
2	keon/algorithms	TheAlgorithms/Python	Algorithms	Algorithms	0.895993	0.900286
3	lepture/authlib	evonove/django-oauth-toolkit	OAuth	OAuth	0.891102	0.878009
4	idan/oauthlib	joestump/python-oauth2	OAuth	OAuth	0.879883	0.919387

Fig. 9: The five most similar repositories from the *awesome-python* list. The results are sorted based on code similarity.

IV. RepoSnipy Neural Search Engine

In this paper, we also present `RepoSnipy` [15], an innovative web-based neural search engine designed to improve the exploration and discovery of Python repositories. It is built upon the solid foundation of `RepoSim`, which represents Python repositories at both the source code and documentation levels. Leveraging the fine-tuned *UnixCoder* models, `RepoSnipy` enables users to query a public Python repository hosted on GitHub, initiating a sophisticated search process that utilizes neural network techniques to identify popular repositories with high semantic similarity.

The front end of `RepoSnipy` is meticulously crafted using the Streamlit library [16], providing an intuitive and visually appealing interface for users to interact with. Under the hood, the repository embeddings generated by `RepoSim` are efficiently

[15]https://github.com/RepoAnalysis/RepoSnipy/tree/main
[16]https://streamlit.io

stored and managed using the docarray framework [17], ensuring rapid retrieval and seamless handling of user queries.

A. DataSet

To ensure a comprehensive and diverse dataset for accurate comparison of user queries, `RepoSnipy` relies on a meticulously constructed repository dataset. This dataset serves as the foundation of `RepoSim`, providing the repository data against which user queries are compared. Its creation process involved a systematic approach, leveraging the capabilities of the GitHub API. Specifically, we focused on Python repositories hosted on GitHub, selecting repositories with a substantial following defined by having over 300 stars. To ensure the legitimacy and compliance of the repositories, we filtered out those lacking a license tag, as licensing plays a vital role in software distribution and usage rights.

This dataset comprises 9,702 Python repositories and was carefully selected based on predefined criteria, ensuring a diverse representation of domains, topics, and development practices, thus providing a comprehensive sample for subsequent analysis and experimentation. The process of running all embeddings on an RTX 4090 server took approximately 30 hours. Our dataset follows the following schema:

- repository name (type: str) in the format owner/name
- list of topics (type: List[str]) obtained from metadata using the GitHub API,
- number of stars (type: int) at the time of running RepoSim
- license (type: str) of the repository
- code embedding (type: TorchTensor[768]), which represents the *code-mean* embedding of the repositor
- doc embedding (type: TorchTensor[768]), which represents the *docstring-mean* embedding of the repository.

To facilitate efficient indexing and retrieval of repository information, we developed a custom method that leverages the powerful capabilities of the `RepoSim` Hugging Face pipeline (see Section III-D). This method seamlessly transformed the list of repository names into embeddings, capturing the inherent semantic information encoded within the repositories' source code and documentation. These embeddings were subsequently integrated into a structured docarray index, enabling streamlined storage and indexing of multimodal data, including repository metadata and embeddings. For the convenience of the research community and practitioners, we have provided this facility in the `RepoSnipy`[18] that enables the creation of a custom index with user-specified repositories. This facility would empowers users to tailor the dataset to their specific needs and explore the functionality and performance of `RepoSnipy` on their own curated set of repositories.

B. Architecture

`RepoSnipy` comprises several interconnected components working together to provide the desired functionality. The architecture can be divided into two main components: User Interface (UI) and Backend.

[17]https://docs.docarray.org

[18]https://github.com/RepoAnalysis/RepoSnipy/blob/main/data/

1) User Interface (UI): The UI component is responsible for providing an intuitive and user-friendly interface for interacting with `RepoSnipy`. It allows users to enter their search queries and displays the search results in a visually appealing manner. In `RepoSnipy`, the Streamlit library is utilized to create the front end of the web-based interface, providing an interactive and responsive experience for users.

2) Backend: The Backend component is responsible for processing user queries, retrieving relevant repositories from the dataset, and computing their similarity scores. It consists of several subcomponents:

- Query Processor: This subcomponent handles the processing of user queries, including parsing and understanding the query's context and intent.
- Embedding Extractor: This subcomponent extracts embeddings from the repositories in the dataset. It utilizes the `RepoSim` pipeline (see Section III-D) to convert repository names into embeddings, enabling efficient comparison and retrieval of similar repositories.
- Similarity Scorer: This subcomponent calculates the similarity scores between user queries and the repositories in the dataset. It employs advanced similarity metrics, such as cosine similarity, to determine the semantic similarity between queries and repositories.
- Ranking Module: The ranking module ranks the retrieved repositories based on their similarity scores, ensuring that the most relevant and similar repositories appear at the top of the search results.

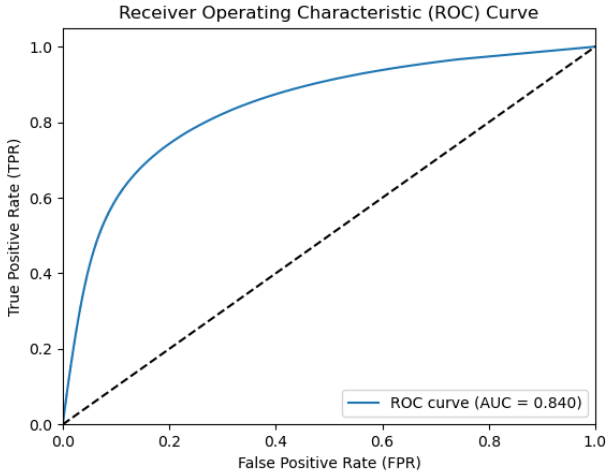
The architecture of `RepoSnipy` emphasizes the seamless integration of these components to deliver an effective and user-centric search experience. The UI component enables users to interact with the system, while the Backend component handles the query processing, similarity computation, and ranking. The dataset introduced in Section IV-A serves as the underlying data source, facilitating the comparison of user queries against a diverse collection of Python repositories.

C. `RepoSnipy` Evaluation

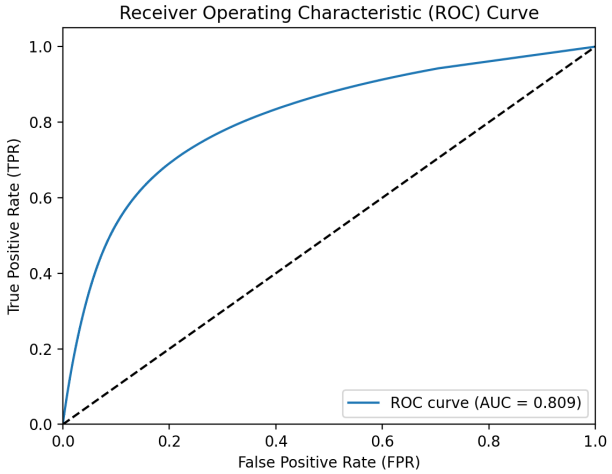
We evaluated `RepoSnipy`'s performance using a similar approach as in the evaluation of `RepoSim`. To assess similarity, we considered shared topics between repositories as a proxy, assuming that repositories with shared topics are more likely to be similar. We conducted evaluations for both code and docstring similarities. Initially, we filtered out repositories (using the dataset introduced in Section IV-A without code or docstring embeddings, resulting in 9,690 repositories for analysis.

For each pair of distinct repositories, we calculated cosine similarity scores based on their code and docstring embeddings, yielding 46,943,205 code similarity scores and 46,943,205 docstring similarity scores. Additionally, we identified shared topics between repositories (excluding 'python' and 'python3') and used this information as binary similarity labels, resulting in 46,943,205 code similarity labels and 46,943,205 docstring similarity labels.

To assess the performance, we derived the ROC and AUC scores using the binary labels and similarity scores. The AUC score provides a single number summarizing the overall performance of the ROC curve. In our evaluations, the ROC-AUC scores for code and docstring embeddings were 0.84 and 0.809, respectively (Figure 10). These results indicate that the embeddings produced by `RepoSnipy` (powered by `RepoSim`) effectively distinguish between similar and dissimilar repositories based on shared topics.



(a) ROC curve evaluating **code embeddings**. AUC is 0.84.



(b) ROC curve evaluating **docstring embeddings**. AUC is 0.809.

Fig. 10: *`RepoSnipy` ROC curves.*

Although shared topics do not guarantee similarity between repositories, this evaluation method serves as a reasonable proxy for similarity in the absence of manually labeled data.

D. `RepoSnipy` in Action

To demonstrate the power and effectiveness of `RepoSnipy`, we invite users to access the tool at [19]. Upon entering a query repository, `RepoSnipy` dynamically evaluates the semantic similarity between the query and a vast array of repositories, presenting users with a curated selection of the most popular repositories that closely align with their search intent.

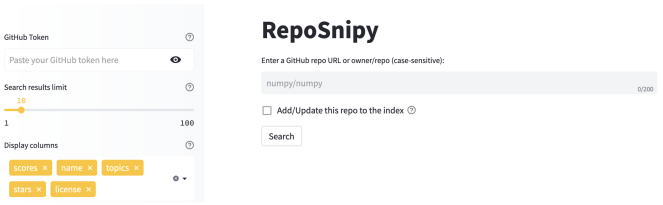


Fig. 11: `RepoSnipy` User Interface.

To initiate a search in `RepoSnipy`, users can simply enter a repository in the format of <owner>/<repo> (e.g., numpy/numpy). After submitting the search, `RepoSnipy` rapidly processes the query and generates a list of search results ranked by their similarity to the user's query. Each search result is presented as a repository card, displaying key information such as repository name, similarity score, topics, stars, and license. Figure 11 illustrates this search process.

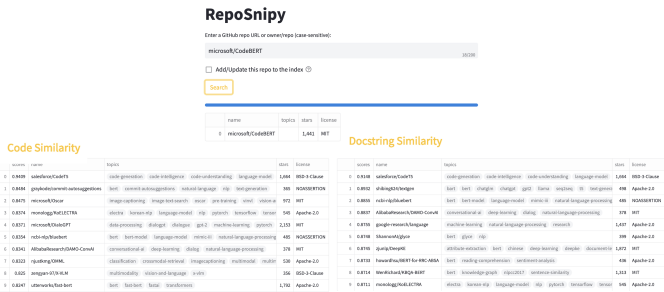


Fig. 12: Obtaining the most similar repositories to `CodeBert`.

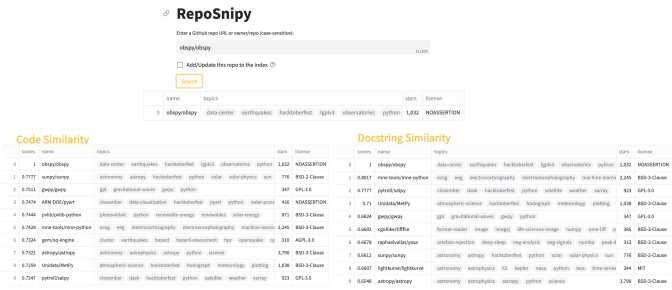


Fig. 13: Obtaining the most similar repositories to `obspy`.

Figures 12 and 13 showcase the search results for `CodeBert` and `obspy` GitHub repositories, respectively, representing different scientific domains. `CodeBert` focuses on natural language processing (NLP) and code analysis, offering a toolkit for training models to understand and process code-related tasks. On the other hand, `obspy` is specifically designed for seismology, providing an open-source Python framework for processing, analyzing, and visualizing seismic data. The similarity between the selected repositories and their respective domains can be observed by inspecting their associated topics.

[19]https://huggingface.co/spaces/Lazyhope/RepoSnipy

Furthermore, RepoSnipy allows users to customize the search results by selecting the desired number of repositories to display in the table and choosing the columns to be shown. This feature provides users with flexibility and control over the information presented to them.

Finally the checkbox in RepoSnipy (*'Add/Update this repo to the index'*), when clicked by a user, triggers the action of encoding the latest version of the selected repository and adding or updating it in the index. This functionality ensures that the search index in RepoSnipy remains up to date with the latest changes and additions to the repository. Therefore, users can access the most recent code and documentation when searching for similar repositories, ensuring that the search results reflect the most current state of the repository. This feature is particularly useful in dynamic and evolving software development environments, where repositories frequently undergo updates and revisions.

V. Related Work

In this section, we discuss the relevant work in the field of Python repository representation and semantic search engines, compare them with our proposed solutions.

A. Repo2vec

Repo2vec [25] is a widely recognized tool for learning distributed representations of Java software repositories using deep learning techniques. It leverages the Skip-gram model to embed source code and documentation into continuous vector representations. While *Repo2vec* primarily focuses on generating embeddings for code and documentation, it lacks a comprehensive command-line interface for repository analysis and lacks a web-based search engine component.

In contrast, RepoSim, our command-line toolbox, represents Python repositories at both the source code and documentation levels. It provides a holistic solution for repository analysis, enabling users to extract embeddings, compute similarities, and perform various tasks through a command-line interface. Moreover, RepoSnipy, built upon RepoSim, adds a web-based neural semantic search engine that facilitates efficient and intuitive repository search and retrieval.

B. Topical

Topical [26] is another tool that addresses the representation of repositories, specifically focusing on topic modeling. It utilizes Latent Dirichlet Allocation (LDA) to extract latent topics from the textual content of repositories.

While *Topical* focuses on learning repository embeddings from source code using attention, RepoSim and RepoSnipy provide a more comprehensive and integrated solution for repository analysis and retrieval. They incorporate both code and docstring-level embeddings, offer a wide range of functionalities, and provide a user-friendly web-based interface for efficient repository search based on semantic similarity.

C. RepoPal

RepoPal [20] is a tool that focuses on repository recommendation by utilizing collaborative filtering techniques. It leverages repository metadata, user activity, and collaborative filtering algorithms to provide personalized recommendations to users. While *RepoPal* is valuable for recommendation purposes, it does not address the fine-grained similarity analysis required for comprehensive repository exploration.

In contrast, RepoSim, our command-line toolbox, provides functionalities beyond recommendation, allowing users to explore and analyze repositories based on their code and docstring similarities. RepoSnipy, built upon RepoSim, extends these capabilities by offering a web-based search engine that enables users to query public Python repositories hosted on GitHub and find semantically similar repositories. The combination of RepoSim and RepoSnipy offers a powerful solution for repository exploration and retrieval.

D. CrossSim

CrossSim [4] aims to identify similar software repositories based on various software engineering artifacts, including source code, issue tracking data, and developer collaboration patterns. It leverages a combination of textual similarity measures, graph-based similarity, and topic modeling techniques to compute similarity scores between repositories

CrossSim, RepoSim, and RepoSnipy address different aspects of repository analysis and retrieval. *CrossSim* focuses on similarity analysis across multiple software engineering artifacts, while RepoSim, and RepoSnipy specifically target Python repositories, providing specialized tools for code and documentation analysis.

VI. Conclusions and Future Work

In this paper, we introduced two novel open source tools, RepoSim and RepoSnipy, that contribute to the scientific software community by enabling the comparison and analysis of software repositories, specifically focusing on Python repositories. RepoSim serves as a command-line toolbox for representing Python repositories at the source code and documentation levels, while RepoSnipy is a web-based neural semantic search engine built upon RepoSim.

The introduction of RepoSim and RepoSnipy brings several significant benefits to the scientific software community. Firstly, RepoSim provides researchers and developers with a comprehensive set of functionalities to extract embeddings, compute similarities, and perform various analysis tasks on Python repositories. This allows for a deeper understanding of code similarities, semantic relationships, and documentation comprehension, which are crucial for code reuse, collaboration, and software maintenance.

Furthermore, RepoSnipy extends the capabilities of RepoSim by offering a user-friendly web-based interface for querying public Python repositories hosted on GitHub and finding semantically similar repositories. This empowers

[20] https://github.com/Qualia-Li/RepoPal

researchers and practitioners to quickly identify relevant repositories for their specific development needs, saving time and effort in discovering existing solutions and best practices. The integration of neural network models and advanced natural language processing techniques in `RepoSnipy` enhances the search experience by capturing semantic information and enabling more accurate retrieval of repositories based on their similarities. This further facilitates the exploration and discovery of scientific software repositories, encouraging collaboration, knowledge sharing, and promoting the reuse of existing solutions.

Although `RepoSim` and `RepoSnipy` provide valuable tools for the analysis and comparison of software repositories, there are several avenues for future research and development to further enhance their capabilities and expand their impact, such as integration of additional programming languages or enhanced their embedding techniques. While `RepoSim` and `RepoSnipy` utilize state-of-the-art embedding techniques, further exploration of advanced embedding models could be beneficial. Investigating the use of transformer-based models, graph neural networks, or domain-specific embeddings tailored for software repositories could potentially improve the accuracy and semantic understanding of the representations. Also the integration of richer repository metadata can be studied. Currently, `RepoSim` and `RepoSnipy` primarily focus on source code and documentation. Incorporating additional metadata, such as software licenses, development activity, package dependencies, and community metrics, could provide more comprehensive insights into repository characteristics and enable more advanced analysis and recommendation capabilities.

REFERENCES

[1] D. E. Atkins, Revolutionizing science and engineering through cyberinfrastructure: Report of the National Science Foundation blue-ribbon advisory panel on cyberinfrastructure, National Science Foundation, 2003.

[2] M. Garcia, S. Brown, R. Johnson, The growth and complexity of software codebases in scientific research, IEEE Transactions on Software Engineering 46 (9) (2020) 1078–1096. doi:10.1109/TSE.2019.2911347.

[3] G. Avelino, L. Passos, A. Hora, M. T. Valente, Measuring and analyzing code authorship in 1+118 open source projects, Science of Computer Programming 176 (2019) 14–32. doi:https://doi.org/10.1016/j.scico.2019.03.001.

[4] P. T. Nguyen, J. Di Rocco, R. Rubei, D. Di Ruscio, Crosssim: Exploiting mutual relationships to detect similar oss projects, in: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2018, pp. 388–395. doi:10.1109/SEAA.2018.00069.

[5] V. Stodden, J. Seiler, Z. Ma, An empirical analysis of journal policy effectiveness for computational reproducibility, Proceedings of the National Academy of Sciences 115 (11) (2018) 2584–2589. doi:10.1073/pnas.1708290114.

[6] W. Ke, P. Zhang, Effects of empowerment on performance in open-source software projects, IEEE Trans. Engineering Management 58 (2) (2011) 334–346. doi:10.1109/TEM.2010.2096510.
URL https://doi.org/10.1109/TEM.2010.2096510

[7] J. Cohen, D. S. Katz, M. Barker, N. P. C. Hong, R. Haines, C. Jay, The four pillars of research software engineering, IEEE Softw. 38 (1) (2021) 97–105. doi:10.1109/MS.2020.2973362.
URL https://doi.org/10.1109/MS.2020.2973362

[8] D. S. Katz, N. P. C. Hong, T. Clark, M. Fenner, M. E. Martone, Software and data citation, Comput. Sci. Eng. 22 (2) (2020) 4–7. doi:10.1109/MCSE.2020.2969730.
URL https://doi.org/10.1109/MCSE.2020.2969730

[9] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, S. Panichella, An empirical investigation on documentation usage patterns in maintenance tasks, 2013 IEEE International Conference on Software Maintenance (2013) 210–219.

[10] X. Yao, M. H. Yap, Y. Zhang, An empirical study to evaluate structural similarity for source code translation, in: 2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON), 2019, pp. 1–5. doi:10.1109/TIMES-iCON47539.2019.9024512.

[11] R. Filgueira, D. Garijo, Inspect4py: A Knowledge Extraction Framework for Python Code Repositories, in: IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, IEEE, 2022, pp. 232–236.

[12] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, Unixcoder: Unified cross-modal pre-training for code representation, in: S. Muresan, P. Nakov, A. Villavicencio (Eds.), Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, Association for Computational Linguistics, 2022, pp. 7212–7225. doi:10.18653/v1/2022.acl-long.499.

[13] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Brew, Huggingface's transformers: State-of-the-art natural language processing, CoRR abs/1910.03771 (2019). arXiv:1910.03771.

[14] X. Gu, H. Zhang, S. Kim, Deep code search, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 933–944. doi:10.1145/3180155.3180167.
URL https://doi.org/10.1145/3180155.3180167

[15] A comprehensive review of state-of-the-art methods for java code generation from natural language text, Natural Language Processing Journal 3 (2023) 100013. doi:https://doi.org/10.1016/j.nlp.2023.100013.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages (2020). arXiv:2002.08155.

[17] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, M. Zhou, GraphCodeBERT: Pre-training Code Representations with Data Flow (2020). doi:10.48550/ARXIV.2009.08366.

[18] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, X. Jiang, Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation (2021). doi:10.48550/ARXIV.2108.04556.

[19] Y. Wang, W. Wang, S. Joty, S. C. H. Hoi, CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation (2021). doi:10.48550/ARXIV.2109.00859.

[20] L. van der Maaten, G. Hinton, Visualizing data using t-sne, Journal of Machine Learning Research 9 (86) (2008) 2579–2605.

[21] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, S. Liu, CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation, CoRR abs/2102.04664 (2021).

[22] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, CodeSearchNet Challenge: Evaluating the State of Semantic Code Search (2019). doi:10.48550/ARXIV.1909.09436.

[23] Vinta, Awesome python, https://github.com/vinta/awesome-python (2023).

[24] T. Fawcett, An introduction to roc analysis, Pattern Recognition Letters 27 (8) (2006) 861–874. doi:10.1016/j.patrec.2005.10.010.

[25] M. O. F. Rokon, P. Yan, R. Islam, M. Faloutsos, Repo2vec: A comprehensive embedding approach for determining repository similarity (2021). doi:10.48550/ARXIV.2107.05112.

[26] A. Lherondelle, Y. Satsangi, F. Silavong, S. Eloul, S. Moran, Topical: Learning repository embeddings from source code using attention (2022). arXiv:2208.09495.