THE UNIVERSITY OF CHICAGO

Embedding GitHub Repositories: A Comparative Study of the Python and Java Communities

By

Yutao Chen

December 2021

A paper submitted in partial fulfillment of the requirements for the Master of Arts degree in the Master of Arts in Computational Social Science

Faculty Advisor: James Evans
Preceptor: Pedro Alberto Arroyo

Embedding GitHub Repositories: A Comparative Study of the Python and Java Communities

Abstract

GitHub, the largest platform for open-source software, which allows code contributors to collaboratively develop software in a variety of programming languages, has motivated extensive research on social coding. However, the heterogeneity of GitHub communities in different programming languages has not been explored in previous studies. Inspired by the linguistic relativity hypothesis, I deduce that different programming languages might result in distinct patterns in social coding. This research identifies such discrepancies between the Python and Java communities based on repository representations (embeddings) from a unique, newly constructed dataset. By describing the representation learning process as a pipeline, this thesis first demonstrates how to generate high-quality repository embeddings from content and contextual data, including source code (import), readme text, and co-contributor networks. The evaluation results suggest that models derived from Word2Vec, including Doc2Vec, Import2Vec, and Node2Vec, are the most competitive for representing the GitHub data.

I then used the best performing embeddings to explore language-specific patterns via questions regarding GitHub activities and success. By investigating the consistency between different embedding spaces, I identified how social (contributor) and functional (import, readme) embedding spaces diverge in the Python community but align in the Java community, implying the difference in their socio-functional mapping. Furthermore, the results indicate that functionally similar Python repositories experience more competition and that Python programmers contribute more to functionally diverse repositories when compared with their Java counterparts. Afterward, by analyzing the correlations between the functional diversity and the average popularity among contributed repositories, I found evidence that Python

programmers who commit to dissimilar repositories are more likely to be the contributors of popular repositories, while Java programmers do not exhibit this pattern. Finally, by comparing embeddings with baseline features, I verified their potency to predict repository popularity and discovered that functional embeddings are beneficial for predicting Python repositories, while social embeddings contribute more to Java repositories.

These language-related heterogeneities can be attributed to the inherent difference in philosophy between Python and Java: As a language highlighting flexibility and reusability, Python treasures contributors' ability to produce code for other coders with various functional needs, whereas Java focuses on the independence and thoroughness in programming, prioritizing specialized coding for passive end-users who value only the performance of final products.

1 Introduction

"I think open source is an evolutionary idea for humanity, this idea of transparency. It played out for us in the technology world, but it also played out with the idea of a truth and reconciliation commission and Wikipedia."

–Megan Smith, chief technology officer of the United States (2014-2017)

Since the birth of the first electronic numerical integrator and computer (ENIAC) in 1935, human society has become increasingly inseparable from computing machines, owing to the advancement in computer and information science. Programming languages serve as the translators between humans and machines. Software, functional units developed in different programming languages, now comprise the building blocks of both industrial development and personal life. Therefore, how people design and create software has a profound impact on this era. Unlike manufacturing industries where workers generate products in a closed form (i.e., the process is not shared with others), software engineers embrace a different mode: collaborating in open-source software (OSS) communities (Hippel & Krogh, 2003).

An open-source software community is an Internet-based community that brings people with shared interests together to develop software toolkits, which are shared with anyone inside and outside the community (Ducheneaut, 2005; Lakhani & von Hippel, 2004). GitHub[1], the largest global open-source community, has millions of public open-source projects across different programming languages. Contributors work on their projects in repositories, where source code is stored and can be propagated by forking. The growth of a repository can be achieved by new commits from the owner or by merging pull requests from other developers. This mechanism allows new contributors to join the development, promoting collaborations between previously unconnected programmers. Apart from the openness in code and collaboration, it is

[1] https://github.com/

recommended for each repository to upload a readme file to introduce it to new visitors. Moreover, GitHub also collects summary information for repositories, enabling users to locate reliable repositories of their interests. For instance, topic tags (see Fig.1) usually entail the keywords related to the functionality of a repository, while the number of watchers, stars, and forks reveal how many users attend to, appreciate, and reuse the code from a repository.

The designs mentioned above make GitHub a unique resource for examining the patterns in social coding, i.e., how programmers practice software development through online collaboration. It serves as a window from which scholars can directly observe how human collaboration leads to technical evolution: whether a few lines of code develop into a software empire or fade into a corner of the Internet. With the support of API (Application Programming Interface) wrappers like pygit2[2] and PyGithub[3], the data and activities on GitHub can be tracked for facilitating the academic analysis of social coding (Dabbish et al., 2012; Lima et al., 2014; Thung et al., 2013; Zöller et al., 2020). For instance, Zöller et al. (2020) explored the collaborative topologies among contributors by constructing networks from their pull-request interactions. Borges et al. (2016) probed how the size of the contributor group impacts the popularity growth of a repository. However, most previous studies did not consider language-specific heterogeneity in social coding, i.e., the differences between the repositories or contributors of different programming languages. Theoretically, the hypothesis of linguistic relativity indicates (Kay & Kempton, 1984) that the characteristics of a language influence its users' cognition and behaviors. Programming languages are languages used in a coding environment, so it is reasonable to deduce that their traits affect programmers' coding activities. Practically, such heterogeneity has significant implications regarding the generality of analytical results and may lead to more comprehensive interpretations of the findings.

[2] https://www.pygit2.org/
[3] https://github.com/PyGithub/PyGithub

Therefore, I conducted a comparative analysis between repositories mainly written in Python and those mainly written in Java (at least 50 percentage of the files written in Python or Java) from a newly constructed dataset. Python and Java were chosen for two reasons. First, they are both popular languages on GitHub, thus providing enough samples. Second, Python and Java are inherently dissimilar in aspects such as design philosophy and extensibility (Arnold & Gosling, 2000; Kuhlman, 2011), ensuring that the comparison would be well-founded and explainable.

In particular, this thesis uncovers the discrepancy between the Python and Java communities with vectorized representations (embeddings) of repositories. The specially organized dataset contains the information on repositories from multiple perspectives, including metadata (such as the number of commits and files), functionality content (such as readme text and source code), and the social context composed by their contributors. Despite the variety in data sources, the primary challenge involved how to use them for representation learning. While the repositories can be simply represented with the metadata, this is unlikely to be ideal as metadata usually fails to reflect the functional and social details of repositories. Hence, content and contextual data, including source code, readme text, and project contributors, were utilized to generate the representations in this research. The repositories were converted to data points in a high dimensional space, where their similarity-based relationships were preserved according to the extracted information from a certain data source. Researchers in representation learning have proposed various models for fulfilling this task (Bengio et al., 2013; Hamilton et al., 2017), but whether they fit into the data from GitHub has not yet been confirmed; this leads to the first research question of this project: **how can we effectively represent GitHub repositories with content and contextual data, and what models are desirable?** This research fills the existing knowledge gap by elucidating the representation learning process as a pipeline from data processing to embedding evaluation. The results suggest that pre-trained transformer models

such as SciBERT (Beltagy et al., 2019) and topic models (Blei et al., 2003) do not yield the most informative representations for either language, as compared to their prevalence in the textual analysis of other online communities (Mozafari et al., 2020; Pennacchiotti & Gurumurthy, 2011; Wang et al., 2020). Instead, models akin to the Word2Vec model (Mikolov, Chen, et al., 2013; Mikolov, Sutskever, et al., 2013), such as Doc2Vec (Le & Mikolov, 2014), perform the best in evaluation tasks, indicating their merits in embedding documents with domain-specific semantics.

With high-quality embeddings, I investigated the language-specific heterogeneity concerning GitHub activities and success. My second research question relates to the socio-functional mapping between the contributors and repository functions: **do repositories more alike in functionality share more similarities in contributor composition, and vice versa?** This question was answered by measuring the consistency between functional embedding spaces (import, readme) and social (contributor) embedding spaces. The findings suggest that while social and functional spaces align in the Java community, they diverge considerably in the Python community. Furthermore, the results imply more competition between functionally similar Python repositories and more diversity in the functions of repositories Python programmers contribute to. In addition, these discoveries can also help improve repository recommendation systems by indicating the pitfalls in system evaluation and generality.

Inspired by the results from the second question, I aimed to explain Python programmers' preference for functional diversity among contributed repositories. One hypothesis is **that programmers contributing to functionally diversified repositories are more likely to be the contributors of popular repositories**. The third question focuses on testing this. The result shows a significant positive correlation between the functional diversity and the average stars of the contributed repositories for Python coders, but not for Java programmers, supporting the hypothesis and indicating that such diversity is more valued in the Python community.

My final question involves the success of repositories: **how can repository embeddings help predict the popularity of repositories?** Different combinations of embeddings were compared to a baseline model with only metadata features; the results suggest that embedding-based features contribute to the accuracy of popularity prediction to a non-trivial extent. More importantly, the prediction performance also indicates the difference between the two language communities: while functional embeddings are more helpful to predict the Python repositories, social embeddings are more efficacious for Java.

The remaining parts of the paper are structured as follows: Section 2 reviews the literature on social coding studies in GitHub and embedding-based research on online communities; Section 3 introduces the GitHub data used for this research; Section 4 illustrates the details about the representation pipeline; Section 5–7 demonstrates the analyses of the second, third, and fourth research questions, where the difference between the Python and Java communities are scrutinized; Section 8 discusses the indications of the findings, explains language-specific heterogeneity, and proposes directions for future work; and Section 9 presents the conclusion.

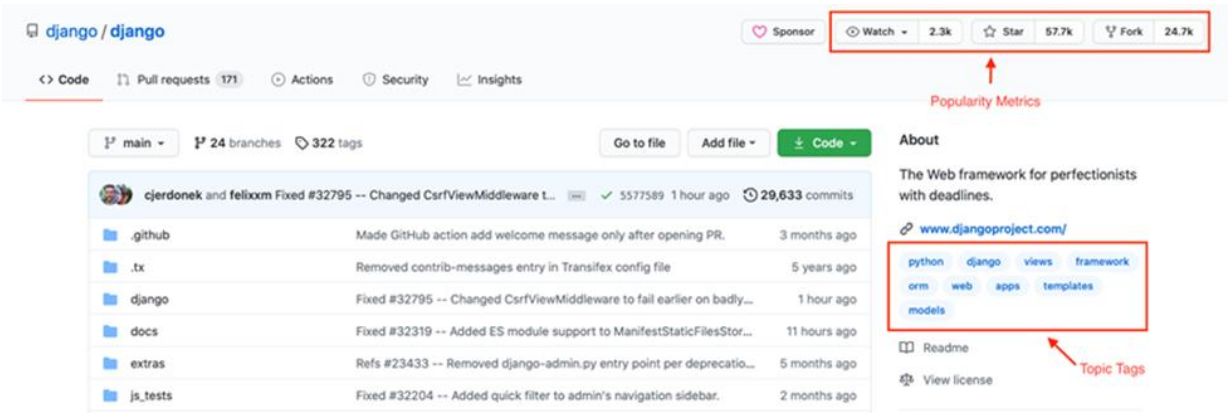

Figure 1. Topic tags and popularity metrics

2 Literature Review

2.1 Social Coding in GitHub

Treated as a prominent digital camp for social coding, GitHub has inspired researchers to look

into the collaborative process and outcomes with various questions and methodologies. For this review, I mainly focused on studies related to four areas in GitHub: collaboration patterns, repository popularity and productivity, coder expertise, and coder influence.

Most research on **collaborative patterns** relies on the social networks of contributors, such as pull-request networks (El Mezouar et al., 2019; Fu et al., 2021; Zöller et al., 2020) and contributor-collaboration networks (Alves et al., 2016; Batista et al., 2017, 2018; Moradi-Jamei et al., 2021; Thung et al., 2013). A pull-request network is built by linking the coders who made pull requests to the reviewers of the requests and is suitable for probing micro-level collaborations within a GitHub repository. For example, Zöller et al. (2020) characterized five particular structures from repositories' pull-request networks and identified that the repositories of a centralized topology obtain more popularity on average. A contributor-collaboration network is formed by connecting coders who have contributed to the same repositories, thus describing the macro-level collaborations of programmers across multiple repositories. Batista et al. (2018) used such networks to develop new metrics to measure the collaboration strength between two developers. Moradi-Jamei et al. (2021) treated this network as a testbed for their community detection method.

Despite the extensive use of networks, most prior studies portrayed collaborations from a contributor-to-contributor perspective. Collaborations can also be reflected with a repository-based network, in which the edges represent the co-contributors between two repositories. This network is valuable in shaping the social relationships between repositories (Thung et al., 2013). Moreover, it helps in embedding social context for repositories. Therefore, I adopted co-contributor networks for the embedding process.

Papers exploring **repository popularity and productivity** contained more diverse methods, and the questions could be either explanatory (i.e., what factors affect them) or predictive (i.e.,

how to predict them). For example**,** Borges & Tulio Valente (2018) and Zhou et al. (2019) launched surveys to learn the reasons users fork or star a repository and what factors impact their decisions. Han et al. (2019) strived to construct feature sets to predict the popularity of repositories, with the number of stars being the metric for popularity measurement. Motivated by these studies, I also incorporated the popularity prediction task in this paper to elucidate the predictive power of embedding-based features and to examine the differences in predictive performance between Python and Java. Regarding repository productivity, researchers have primarily defined productivity as the volume of work (such as commits, merging pull requests, or solving issues) finished within a certain period (Choudhary et al., 2020; Saadat et al., 2020; Sheoran et al., 2014). For instance, Saadat et al. (2020) assessed how group size affects the efficiency of contributors in dealing with tasks such as merging pull requests and commenting on issues. Furthermore, a few scholars (Fang et al., 2020; Vasilescu et al., 2013) analyzed this issue by crosslinking contributors' activities on multiple platforms (StackOverflow, Twitter).

Papers on **coder expertise** utilized contributors' committing history as the most common data source (Bhattacharya et al., 2014; Constantinou & Kapitsaki, 2016; Dey et al., 2021; Montandon et al., 2019). For instance, Constantinou & Kapitsaki (2016) measured contributors' language-specific expertise by extracting language-related statistics from their committing history. One related research by Dey et al. (2021) represented contributors' coding skills by training Doc2Vec models with the APIs from their modified code files. Moreover, the authors validated the embeddings by examining the consistency between the embedding space and the self-reported expertise from their profiles. The aptness of this research inspired me to deploy an embedding-based approach in this paper.

Studies on **coder influence** frequently analyzed follower networks (Ma et al., 2017; Yu et al., 2014). A follower network is a directed graph with edges from the following users to the followed users. Following a developer on GitHub is similar to following a friend on social

media platforms, such as Facebook and Instagram. Thus, researchers have explored structural similarities and differences between follower networks on different sites (Ma et al., 2017). While some studies used single networks, others adopted a combination of multiple networks (Hu et al., 2018; Lima et al., 2014). For instance, Lima et al. (2014) compared the follower network with the collaboration network, revealing their similarity in the power-law distribution of node degrees.

Although some of these studies cover repositories or contributors associated with different programming languages (Borges & Tulio Valente, 2018; El Mezouar et al., 2019; Lima et al., 2014; Yu et al., 2014), the heterogeneity of programming languages has not been explored in their analyses or findings. Therefore, to the best of my knowledge, this is the first research emphasizing the ways in which the GitHub communities of different programming languages diverge.

2.2 Representation Learning Applications in Online Community

Representation learning, also known as embedding learning, is a concept that depicts the process of extracting useful information from data as features for predictors (Bengio et al., 2013). This notion is widespread in natural language processing, computer vision, and network science (Bengio et al., 2013; Hamilton et al., 2017), where researchers invent methods for representing unstructured data. Along with the speedy evolution of artificial intelligence, representation learning methods have become increasingly prevalent in fields outside computer science. Online communities, replete with both structured data (such as user profile, metadata) and unstructured data (such as textual comments, image posts, video posts), provide a natural environment for embedding-based studies.

For example, researchers have successfully applied textual embedding models such as BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018) and topic

models (Blei et al., 2003) to the sentiment analysis of user comments in social media during the COVID-19 pandemic (Wang et al., 2020); the identification of hate speech and racial bias in Twitter (Mozafari et al., 2020); and the mining of user interests for content recommendation (Pennacchiotti & Gurumurthy, 2011). Moreover, network embeddings are also widespread in studies of online communities. For instance, Li et al. (2019) proposed the possibility of generating customized answers in online question-answering communities using network embedding. Zhang et al. (2019) used multi-view network embeddings to detect key players in underground forums.

In contrast to the pervasive adoption of embedding-based approaches in the analyses of other online communities, such as Twitter and Weibo, their applications in OSS communities remain limited. Except for sporadic cases like those mentioned in 2.1, recommendation systems form the domain in OSS where this methodology is most practiced. Previous studies have used metadata (Liu et al., 2018), APIs (McMillan, Grechanik, & Poshyvanyk, 2012), readme text (Zhang et al., 2017), or user behaviors (Jiang et al., 2017) as the source of representation learning, then made recommendations based on the similarities of embeddings. Additionally, a few studies (Alon et al., 2019; Theeten et al., 2019) benefited from the code available on GitHub, creating representation learning methods for code or code elements. Nevertheless, these studies did not delve deeper into the activities or patterns in social coding, leaving the application of embeddings at a purely technical level. Therefore, this research is novel as it unveils how embeddings conduce to pattern identification and interpretation regarding activities on GitHub.

3 Data Description

The GitHub data used in this research has two sources. GitHub content data, such as the source code, readme files, and contributors, were fetched from pre-downloaded repositories, whereas

other data, including topic tags and popularity metrics, were obtained via web-scraping and API searching:

- Pre-downloaded repositories are collected by the Knowledge Lab at the University of Chicago from all public GitHub repositories that were active until 2019 with the library pygit2. They are stored as zip files at the university's research computing system (RCC), and each file contains the commit history of a GitHub repository or a collection of repositories. The content data and metadata for this study were extracted from the latest version of the repositories. The metadata[4] includes the number of unique import packages, the number of programming languages, the number of files, the number of contributors, the number of commits, the average number of commits per contributor, and the length of the cleaned readme text.
- Topic tags serve as naturally annotated labels indicating the functionality of a repository. They are scraped as the labels for the prediction task assessing the performance of readme embeddings because they can be viewed as keywords of the readme text. However, since GitHub allows developers to use self-defined labels, some of the topic tags are too customized for a classification task (for instance, a label that only occurs once would not have a valid train-test split). Thus, we only reserved the tags appearing over 50 times and removed non-informative ones, such as *python*, *python3*, *java*, and *java8*.
- Popularity metrics include the number of stars, watches, and forks available on the repository pages and were obtained with PyGithub, a python interface of the GitHub API. In addition, I queried whether a repository was forked from others or not. Forked repositories would have the same number of forks and identical content as the original

[4] The degrees of repositories in the co-contributor network are added to metadata after preprocessing.

repositories but would not have any stars or watches when being forked. Thus, they would cause discrepancies in the data generation process and should be excluded from popularity-related analyses.

4 Representation Learning Pipeline

This section describes how repository embeddings are generated from different data sources and evaluated using different prediction tasks. The entire process is summarized in a representation learning pipeline (see Fig. 2) of three phases: data preprocessing, embedding generation, and embedding evaluation. Each step in the pipeline and its corresponding outcomes are detailed in the subsections.

4.1 Data Preprocessing

Contributor List

The contributor list contains all the contributors in a repository. The embedding method first generates a co-contributor network from the lists; then, it yields repository representations by embedding that network. In this network, the nodes are the repositories that are not isolated (i.e., having at least one co-contributor with others), and the edges represent the existence of co-contributors between two repositories weighted by the number of co-contributors. The summary statistics of the networks for Python and Java are displayed in Table 1. Though the number of nodes (repositories) in the Python network is fewer than the Java network, the number of edges and the average node degree in the Python network is larger than the Java network. This suggests that Python repositories are socially more connected.

Table 1. Contributor network degree summary

Language	node	edge	mean	std	min	25%	50%	75%	max
Python	410869	116838952	569	2074	1	2	10	141	40089

Java	632675	23684474	75	363	1	1	3	9	9467

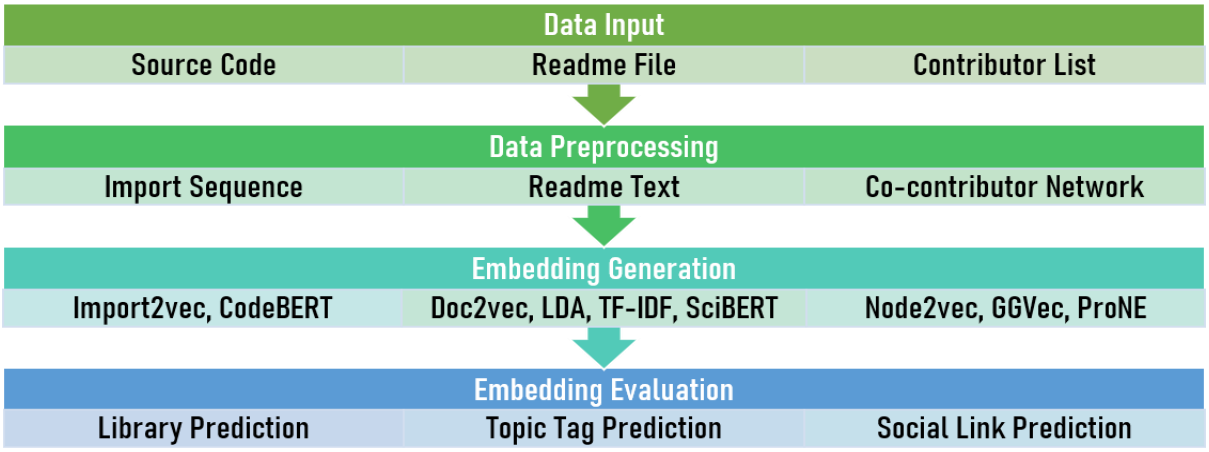

Figure 2. Representation learning pipeline

Readme File

The raw readme files are usually structured in Markdown or RST files. They contain characters used for formatting, example codes, and links, all of which need to be removed to retain only the informative text. To clean the raw text, I used the *pypandoc*[5] to convert it to HTML and then parsed the HTML with *beautifulSoup* to preserve useful tags (see Fig. 3). The contents under the paragraph tags (<p>) are usually the most informative, and the list tags (<li>) sometimes contain valuable information. Therefore, I conducted a pre-test on two versions of cleaned readme text (with and without text from lists). The pre-test was performed on the readme files of the top 1000 starred Python repositories with Doc2vec, and the result (see Fig. 4) implies that the inclusion of list contents helps increase the distinction between two clusters: TensorFlow-based repositories (AI) and Django-based repositories (Web). Therefore, I incorporated the content under list tags into the corpora for readme embedding generation. Furthermore, I also removed special characters such as "#" and "&", digits, and email addresses

[5] https://pypi.org/project/pypandoc/

from the cleaned text. Finally, since some of the readme files were not written in English, I filtered them based on the language detection results using *langdetect*[6].

The statistics of the repositories with cleaned readme files are displayed in Table 2, in which we can see that most readme files have relatively fewer words compared to a regular article. Unlike node degrees in co-contributor networks, the distribution of readme length for Python and Java repositories is similar.

Table 2. Readme length summary

Language	Repo-count	mean	std	min	25%	50%	75%	max
Python	370,198	195	478	1	17	66	203	65,271
Java	331,815	170	631	1	13	50	173	173,866

Figure 3. Readme text conversion

[6] https://pypi.org/project/langdetect/

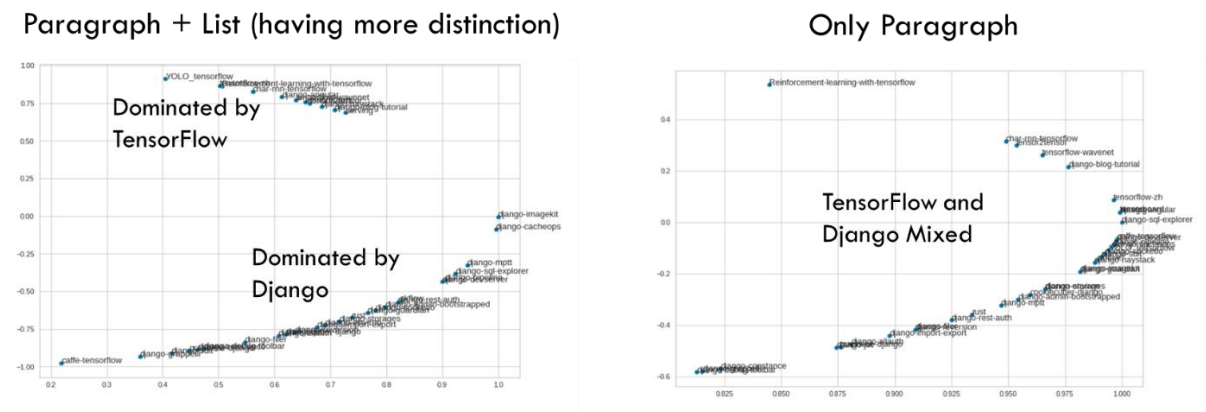

Figure 4. Readme pre-test result

Source Code

Regarding the techniques for implementing repository functions, the import packages are the most informative part. Hence, the representations from the source code were built by embedding the imports. I parsed the code through abstract syntax trees and fetched the import objects from the corresponding syntax (see Fig. 5). Coders can import packages at multiple levels, such as “scipy” and “scipy.stats.” To narrow down the number of imports, I only included the top-level libraries (“scipy.stats” to “scipy”). I also filtered the imports based on whether they were “good” imports; a good import is a package or module that has been imported over ten times and imported in repositories with at least five stars. Moreover, only statically declared imports were included, and dynamic imports were not considered. It was found that, on average, Python repositories have 15.73 unique imports after filtering, which is considerably more than that of Java. In addition, the number of imports varies more for Python repositories (see Table 3).

Table 3. Unique imports summary

Language	Repo-count	mean	std	min	25%	50%	75%	max
Python	604,047	15.73	38	0	4	7	14	1724

Java	939,957	3.33	2	0	2	3	4	107

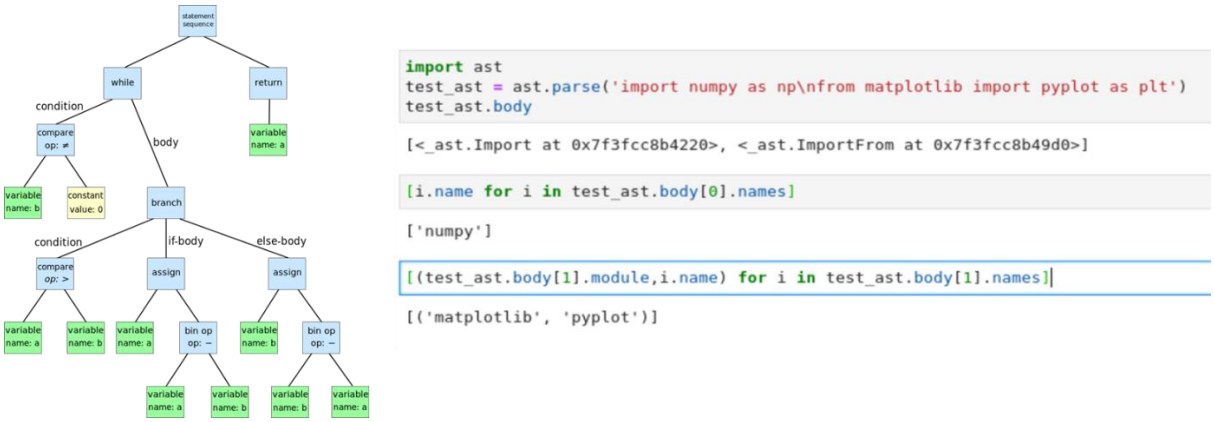

Figure 5. Imports extraction using abstract syntax tree

After collecting the imports, I performed a pre-test to decide how to structure the import data for embedding. I structured the extracted imports in two formats with different granularity: (1) file-level, in which each document in the import corpus is the sequence of libraries used in a code file, and (2) repo-level, in which each document in the import corpus is the sequence of libraries used in a repository. The pre-test was also conducted on the top 1000 starred Python repositories with Doc2vec. To visualize the result, I randomly chose repositories whose URL included TensorFlow, Django, or SQL as samples (ten for each). The result (Fig. 6) suggests that the file-level corpora can catch the heterogeneity between the repository clusters, while the repo-level one cannot. This is because within-cluster similarities are higher than between-cluster similarities in the file-level space. Hence, the file-level corpus was employed for import embedding generation.

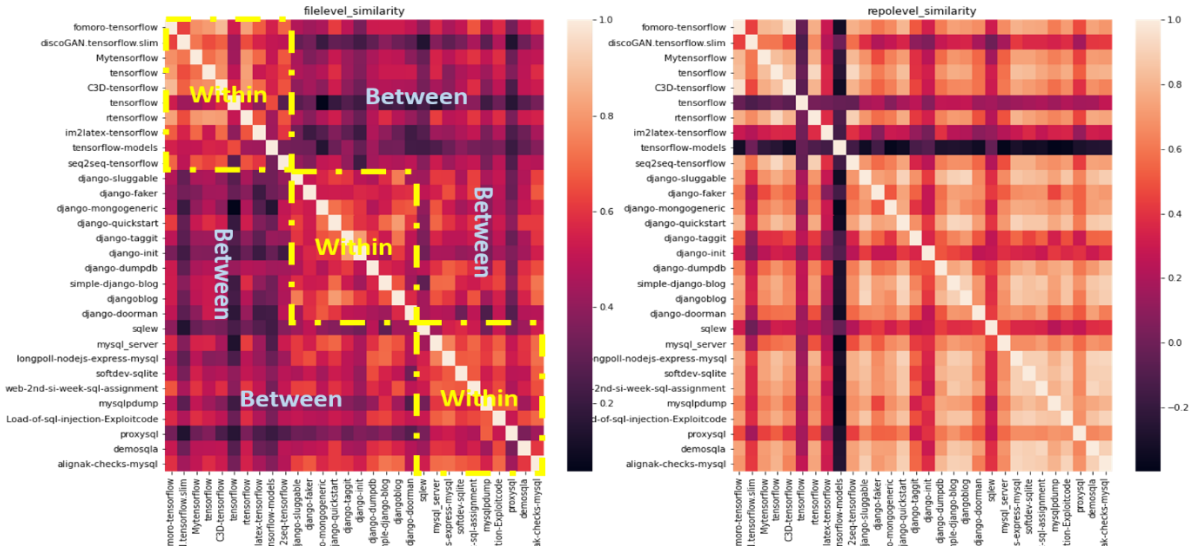

Figure 6. Imports' pre-test results

4.2 Embedding Generation

Co-contributor Network

GGVec: This method is provided by the graph embedding generation package *nodevectors*[7], designed to visualize large networks. For GGVec and all other network-based embeddings, I set the dimension size of the vectors to 50[8].

ProNE: This method acts as a fast and scalable approach for embedding large-scale networks (Zhang et al., 2019). It involves two steps: (1) sparse matrix factorization for fast embedding initialization and (2) spectral propagation in the modulated networks for embedding enhancement.

Node2Vec: This approach (Grover & Leskovec, 2016) deploys random walks through a network to generate "corpora" composed of the nodes (repositories) in the network. Then it uses the corpora as the input for a Word2Vec model. The trained word vectors are the

[7] https://github.com/VHRanger/nodevectors
[8] Dimension sizes of 10,30,50, and 100 were tested, with the results indicating a dimension size of 50 leads to an interpretable embedding space with the least computational cost.

embeddings of the nodes in the network.

Readme Text

Doc2Vec: This model (Le & Mikolov, 2014) extends the Word2Vec model by adding a paragraph (document) vector. In particular, the training algorithm is the same as the Word2Vec model; the difference lies in the concatenation of a vector representing a document before word vectors, i.e., the document vectors are trained jointly with the word vectors. Researchers have proved that this method outperforms the simple method of averaging word vectors as the document vectors in downstream tasks (Le & Mikolov, 2014). Similar to Word2Vec, there are two training algorithms for Doc2Vec: (1) PV-DM, which corresponds to the continuous bag of word (CBOW) approach in Word2Vec, predicts the center word from the contextual words, and (2) PV-DBOW, which corresponds to the skip-gram method in Word2Vec, reverses the task in PV-DM. I utilized the PV-DBOW model because it can be trained fast and well for short documents.

Topic Attention: This embedding generation method builds on the LDA topic model (Blei et al., 2003). The documents are viewed as a mixture of topics, and each topic is a Dirichlet distribution over words. An important hyperparameter is the number of topics, *k*. Following (Newman et al., 2010), I referred to the average topic coherence score to choose the optimal number of *k*. The coherence score for a given topic measures the degree of semantic similarity between the high-scoring words under this topic. (Newman et al., 2010) showed that this measure could select topics closer to human intuition than likelihood-based (perplexity) ones. The test result implied an optimal topic number of 20. Therefore, it was determined that each repository would have a 20-dimensional representation of topic loadings.

TF-IDF: This method makes a sparse vector for each document, highlighting unique words within. The TF-IDF (Ramos & Others, 2003) score for a particular word in a document is

calculated as the product of its term frequency (TF) and the inverse document frequency (IDF) of the word. The TF is defined as the frequency with which a word occurs in a document. The IDF measures how common a word is across the entire corpus. The TF-IDF vectors for Python and Java readme texts have over 100,000 sparse dimensions. To make the evaluation more feasible, I further reduced the TF-IDF vectors to 50 dimensions by capturing the first 50 principal components (Jolliffe, 2013) from the raw vectors.

SciBERT: This method leverages the power of the deep bidirectional transformer model, BERT (Devlin et al., 2018), to generate text representations. BERT generates contextual embeddings for tokens (usually words) based on multiple attention mechanisms, allowing the model to selectively focus on the most informative segments from the input. The model will first generate token embeddings in the last hidden layer given a tokenized document. Then, the token embeddings would be pooled to aggregate the information into document embeddings. The pooling method I performed involved taking the vector of the special token (CLS), which the BERT author recommended as an aggregated sequence (document) representation. These pooled embeddings formed the final representation for the readme files, which were vectors of 768 dimensions. SciBERT is a specialized BERT, trained on scientific papers from different disciplines, where computer science accounts for 12% (Beltagy et al., 2019). It is reported to have better performance in tasks on computer science corpora than the general BERT.

Import Sequence

Import2Vec: This model (Theeten et al., 2019) derives from the Doc2Vec model (PV-DBOW). Unlike the Doc2vec model for readme data, the Import2Vec model needs to capture the co-occurrence of all import libraries in a given source file. In this model, the input tokens within a pre-defined window are converted into token pairs before training (see Fig. 7). To make all imports in the same file form combinations, I set the window size as the longest sequence length in the corpora minus one. The model's output was also specified to be a 50-dimensional

vector for each imported library (word embedding) and each repository (document embedding).

CodeBERT: This BERT-based model published by Microsoft for code representation (Feng et al., 2020) was trained on pairs of social language and code elements in six programming languages, including Python and Java. In contrast to the file-level data used for Import2Vec, I used the repo-level import sequences for this model to take advantage of the automatic pooling by the token CLS. Otherwise, I would have had to average the file embeddings as repository embeddings, which might have been less effective. The learned vectors also contained 768 dimensions, like those from other transformer models.

Source Text	Training Samples generated from source text			
I will have orange juice and eggs for breakfast	(will, I)	(will, have)	(will, orange)	
I will have orange juice and eggs for breakfast	(have, I)	(have, will)	(have, orange)	(have, juice)
I will have orange juice and eggs for breakfast	(orange, will)	(orange, have)	(orange, juice)	(orange, and)
I will have orange juice and eggs for breakfast	(juice, have)	(juice, orange)	(juice, and)	(juice, eggs)
I will have orange juice and eggs for breakfast	(and, orange)	(and, juice)	(and, eggs)	(and, for)
I will have orange juice and eggs for breakfast	(eggs, juice)	(eggs, and)	(eggs, for)	(eggs, breakfast)
I will have orange juice and eggs for breakfast	(for, and)	(for, eggs)	(for, breakfast)	

Figure 7. Word pairs generation in skip-gram model

4.3 Embedding Evaluation

Examining the quality of embeddings before making further inferences guarantees the reliability of findings. In this research, I assessed embedding qualities by comparing their performances in downstream tasks, which is viewed as extrinsic evaluation (Shi et al., 2018). In particular, I designed separate tasks for embeddings from different data sources instead of using one task to test all the embeddings. This was intended to ensure the prediction targets directly related to the data source of the embeddings. Otherwise, the mismatch between the

task targets and the training data might have invalidated the evaluation results. For example, if the co-contributor embeddings were used to predict the topic tags as an evaluation, the result might not be instructive because the repository relationship reflected from the topic tags and the co-contributor network could be inherently different. Thus, this task may not signify how much the model learned from the training data. Therefore, I highlighted the correspondence between the prediction targets and embedding sources. The tasks and results are presented as follows:

Social Link Prediction

This task evaluates the quality of co-contributor network embeddings. For each repository, I randomly chose one of its neighbors from the network and other *k* repositories that were not neighbors (negative samples) to form a candidate pool. Then I computed and ranked the cosine distances between the repository and each candidate from the pool. This process was replicated ten times[9] for each repository. The links of repositories with more neighbors were usually more challenging to predict because their embeddings were impacted by more neighbor embeddings during steps such as the random walks (Node2Vec, GGVec) or spectral propagation (ProNE). Therefore, I only selected the top 1000 repositories (sorted by the number of neighbors) to predict[10], which made this task more efficient. A high-quality embedding space is expected to rank actual neighbors higher than negative samples. For this task, the following measures, which are standard metrics for link prediction (Costabello et al., 2019; Goyal et al., 2019), were used to measure embedding performance:

- Mean rank (MR): The average rank of the actual neighbors, where a smaller value indicates better performance.

[9] The results were robust for other replication times, including 5, 20 and 50.
[10] The negative samples were picked from all the repositories, not just the top 1000 repositories.

$$\frac{1}{N * S}\sum_{i=1}^{N}\sum_{j=1}^{S} Rank(neighbor)_{ij},$$

Where N is the number of repositories, and S is the number of replications.

- Mean reciprocal rank (MRR): The average reciprocal rank of the actual neighbors, where a larger value indicates better performance.

$$\frac{1}{N}\sum_{i=1}^{N}\frac{S}{\sum_{j=1}^{S} Rank(neighbor)_{ij}}$$

- Hit(N): The ratio when the ranks of the true neighbors are higher than N. I assigned values of 3 and 10 to N, where a larger value means better performance.

The results when k equals 1000 are presented in Table 4. The Node2Vec model performs much better than the other two models on all the metrics for both languages. I also tested the impact of the negative sampling ratio k on prediction performance and found that Node2Vec shows a robust advantage as k increases, while GGVec has the worst performance (see Fig. 8). Therefore, I utilized the Node2Vec embeddings to build a social space for the repositories. Furthermore, the overall performance of the models is better in Java repositories than in Python repositories. This may be pertinent as Python repositories tend to have more co-contributor connections on average. Even non-neighboring repositories can be indirectly connected through their common neighbors, thereby shortening their distances in the embedding space and increasing the difficulty of identifying actual neighbors.

Table 4. Social link prediction results ($k = 1000$)

Language	Model	MR	MRR	Hit-3	Hit-10
Java	GGVec	373.9802	0.0070	0.0010	0.0050
	ProNE	49.4195	0.2335	0.2780	0.3540
	Node2Vec	**4.7498**	**0.4816**	**0.5470**	**0.8780**
Python	GGVec	300.9023	0.0050	0.0	0.0

ProNE	171.2221	0.0082	0.0	0.0010
Node2Vec	**14.9610**	**0.1505**	**0.1140**	**0.4840**

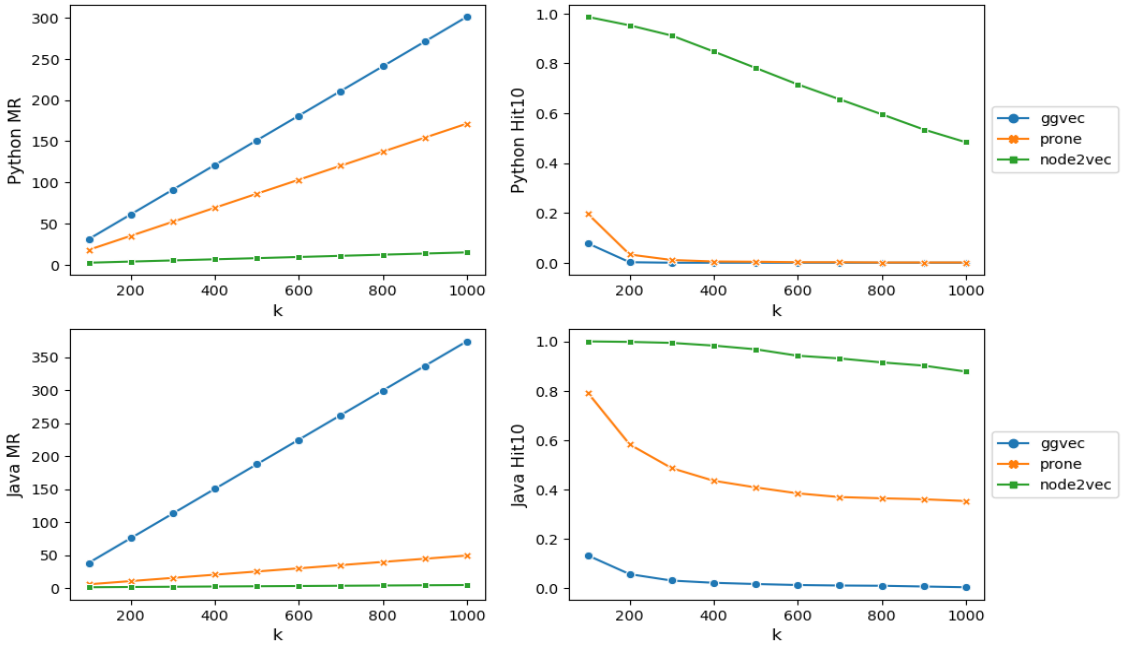

Figure 8. Mean Rank and Hit10 for different negative sampling ratios (*k*)

Topic Tag Prediction

This task evaluates the quality of embeddings from readme text. Each repository may be labeled with multiple topic tags, among which many are interchangeable—for instance, "deep learning" and "neural network." To make the task more feasible, I reduced the size of the label space by detecting the communities in the co-occurrence network of the topic tags. The Louvain algorithm, a community detection method based on modularity optimization (Blondel et al., 2008), was applied to this step. I fine-tuned the detection results to obtain eight reasonable clusters for the Python and Java communities. The topic tag communities are visualized in Fig. 9 and Fig. 10, and the repositories are then labeled with one or more cluster labels below.

- Python: *AI, DS, Web, Platform, Bot, Game, GUI, Finance*
- Java: *Full-stack, Android, Minecraft, DS, Web-Page, Testing, Intellij, GUI*

Following this, the prediction task was formed as a multi-label classification. The classifier's output for each sample was an N-dimensional vector of 0 and 1, with N being the total number of topic tag communities. The classification was performed by a multi-layer neural network (MLP) and optimized with Keras tuner (O'Malley et al., 2019), a toolkit for seeking the optimal neural network structure for a given task in terms of the number of layers, neuron numbers in each layer, dropout rates, and learning rates.

To evaluate the models, I created a test set containing 20% of the repositories. In addition, I considered the following metrics to quantify the model performance:

- Hamming loss (H-Loss):

$$\frac{\sum_{i \in C}(False\ Positives)_i + \sum_{i \in C}(False\ Negatives)_i}{\sum_{i \in C}(Positives)_i + \sum_{i \in C}(Negatives)_i}, C \text{ is the set of labels}$$

1. Micro-precision:

$$\frac{\sum_{i \in C}(True\ Positives)_i}{\sum_{i \in C}(True\ Positives)_i + \sum_{i \in C}(False\ Positives)_i}, C \text{ is the set of labels}$$

2. Micro-recall:

$$\frac{\sum_{i \in C}(True\ Positives)_i}{\sum_{i \in C}(True\ Positives)_i + \sum_{i \in C}(False\ Negatives)_i}, C \text{ is the set of labels}$$

3. Micro F1-Score:

$$\frac{2 * (Micro\ Recall) * (Micro\ Precision)}{Micro\ Recall + Micro\ Precision}$$

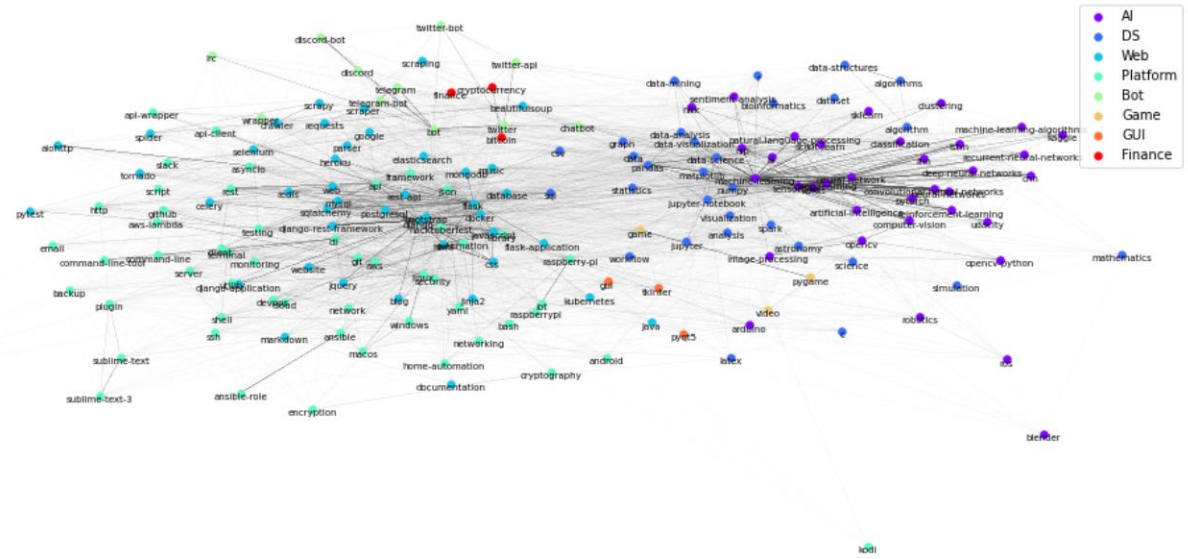

Figure 9. Python topic tag communities

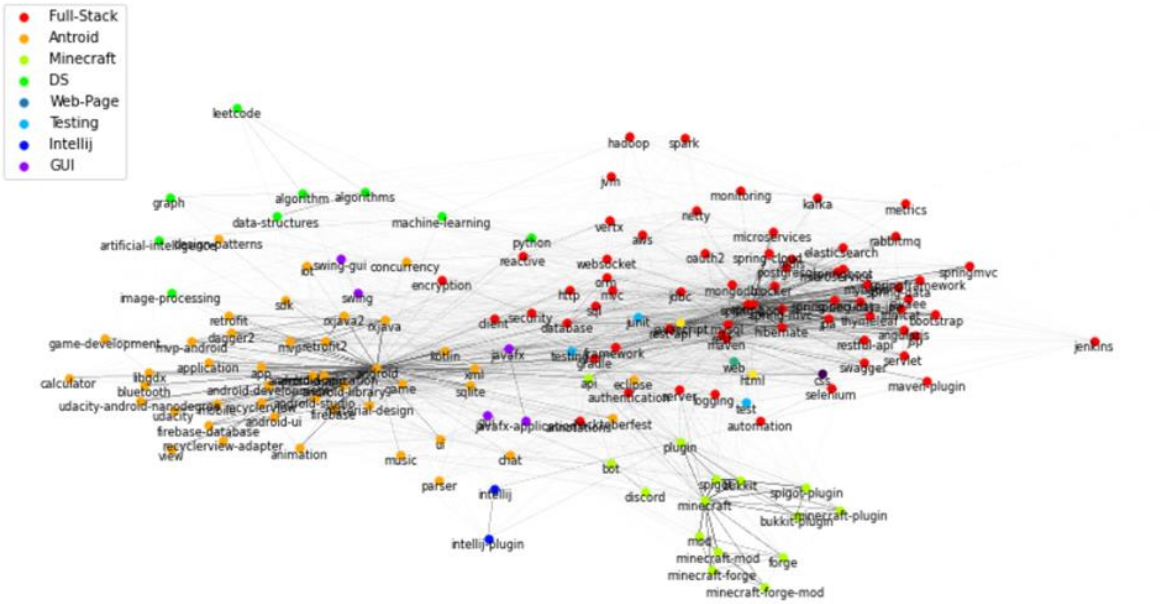

Figure 10. Java topic tag communities

The results are displayed in Table 5; it can be seen that the Doc2vec model outperforms other models in most of the metrics for both languages, while the SciBERT model has the worst performance. The Micro F1-scores for the best models are between 0.7 and 0.8, which can be thought of as a remarkable performance for a multi-label classification task.

Table 5. Topic tag prediction results

Language	Model	H-loss	Micro-f1	Micro-precision	Micro-recall
Java	Doc2vec	**0.0580**	**0.7875**	0.8233	**0.7547**
	Topic	0.0764	0.7177	0.7612	0.6789
	TF-IDF	0.0618	0.7701	**0.8235**	0.7232
	SciBERT	0.0875	0.6744	0.7186	0.6354
Python	Doc2vec	**0.0808**	**0.7188**	**0.7462**	**0.6933**
	Topic	0.1038	0.6365	0.6702	0.6060
	TF-IDF	0.0844	0.7041	0.7433	0.6688
	SciBERT	0.1158	0.6031	0.6167	0.5901

To examine the embedding space more intuitively, I picked out seven topic tags for the Python repositories (*algorithm, bot, django, git, opencv, scraper, tensorflow*) and six topic tags for the Java repositories (*android-application, javascript, json, kotlin, minecraft, mysql*). Repositories with these topic tags were selected as samples for visualization. Ideally, repositories with the same topic tags should be closer in the embedding space and shape visible clusters. The visualization process was performed via t-SNE (Van der Maaten & Hinton, 2008), a nonlinear method for reducing high-dimensional data to two or three dimensions, which is effective for visualizing relative proximities in data. Fig. 11 presents the embedding spaces of the selected repositories. The visualization outcomes support the extrinsic evaluation results above. The embedding space generated by SciBERT appears entirely random, which is in line with its worst prediction performance. This can be explained based on the following: (1) some words used frequently in GitHub readme files, such as library names and framework names, might not appear in general academic works, preventing SciBERT from capturing their semantics in embeddings and (2) previous research has suggested that automatic pooling may not always be effective (Reimers & Gurevych, 2019); thus, it may result in non-informative embeddings for some readme text.

Meanwhile, the superiority of the Doc2vec models may arise from the fewer assumptions they make about the distribution of the corpus compared to topic models and from their lack of reliance on pre-knowledge compared to transformer models. These differences indicate the importance of domain-specific information in representation learning. Choosing models with presumptions or pre-knowledge that the target data does not comply with can result in low-quality embeddings. Researchers who use textual embeddings in their analysis should provide sufficient evidence to show the match between the model and the target text; otherwise, evaluating the embeddings are necessary to verify their methodologies.

Library Prediction

This task evaluates the quality of embeddings from import sequences. One repository can import multiple libraries, but not all libraries are informative in reflecting the functional focus of the repository; therefore, I used the import embeddings to predict only the meaningful libraries instead of all libraries. The importance of the libraries is ranked by their TF-IDF scores; thus, libraries that occur uniquely and frequently in the code of one repository tend to be ranked higher than libraries shared by many repositories. The task is processed with three sub-tests as follows:

- Predict the most important library for the top 10000 repositories that have the largest number of unique imports;
- Predict the most important *n* libraries for the top 10000 repositories, where the libraries are restricted to the top 1000 libraries (ranked by the number of repositories importing). This filtration limits the label space as *n* increases and ensures that each library label has positive samples in training and test sets.
- Predict the most important library for the top 10000 repositories conditioned by different limits on the number of unique imported libraries. The filtration in Test 2 is also applied here for the same reason.

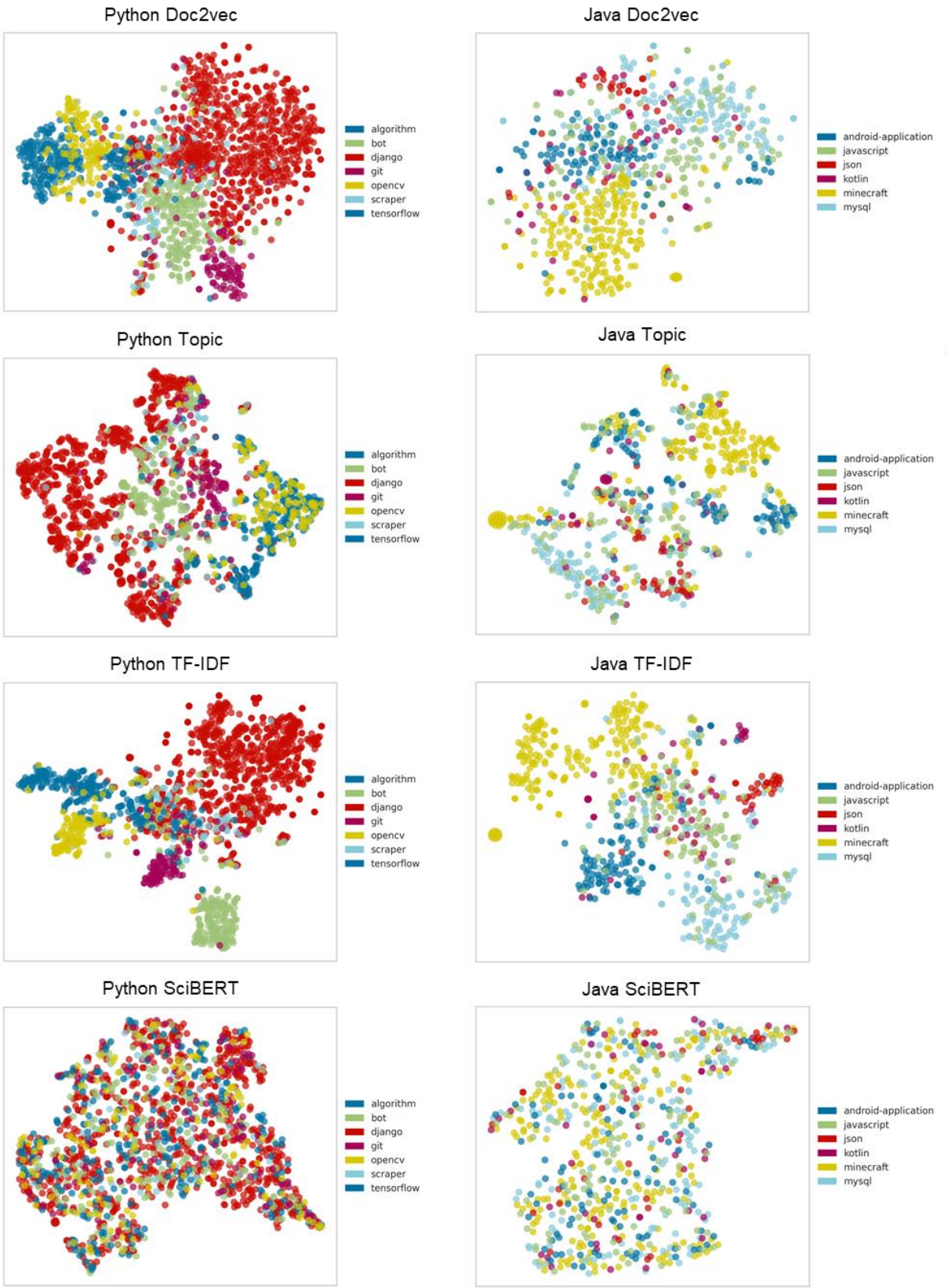

Figure 11. Readme embedding space for the selected topic tags

I performed the sub-tests using a *k*-nearest neighbor (*k*-NN) estimator (Altman, 1992), which assigns the label(s) to a sample with the most common label(s) among its nearest neighbors from the training set. Tests 1 and 3 were multi-class prediction tasks, while Test 2 was a multi-label task. The metrics for the topic tag (multi-label) prediction are also applicable for multi-class predictions, so they were applied to the sub-tests above. 20% of the selected repositories served as the test set for each sub-test, and the final results were cross-validated with 5-folds.

The results[11] of Test 1 are described in Table 6, and it can be seen that Import2Vec outperforms CodeBERT for both languages and all metrics. Fig. 12 and Fig. 13 display the results of Tests 2 and 3, where the Micro F1-scores were plotted. The performance of both models improves as *n* increases because a relatively larger *n* gives the estimator more chances to predict the most important libraries regardless of their relative rankings. The performance of CodeBERT is still inferior to that of Import2Vec in both languages for different values of *n*. Similarly, Fig. 13 indicates that Import2Vec performs better for most repositories regardless of their number of unique imports. The only exceptional case occurs when repositories only have one or two unique libraries; then, CodeBERT exhibits a slight advantage over Import2Vec. In this case, the embeddings of these repositories are similar to the embeddings of imported libraries, suggesting that the token embeddings generated by CodeBERT may be useful in representing libraries. However, automatic pooling may not aggregate the token-level information as accurately as Import2Vec to represent the repositories. Specifically, CodeBERT considers the order of the input tokens (bidirectional), which does not suit the case of import sequences and may result in worse performance for repositories that import more libraries.

In addition, I also examined the import embedding space by visualizing the cosine distance matrices of the repositories importing specific libraries. For Java, 20 repositories importing

[11] The results are robust for number of neighbors 5, 10, and 20.

library *android* (an essential library for app development) or *weka* (a library for machine learning) were randomly chosen. For Python, 20 repositories with library *tensorflow* (a library for deep learning) or *django* (a library for web development) were randomly selected. These repositories were double-checked to avoid overlaps (such as importing both *tensorflow* and *django*). The results (see Fig. 14) reveal that the spaces generated by Import2Vec discriminate between the repository groups, whereas the spaces produced by CodeBERT do not. As all available evidence indicates the advantages of Import2Vec embeddings, I employed them to build the import space for the repositories.

Table 6. Library prediction subtest-1 results (k-neighbors = 10)

Language	Model	H-loss	Micro-f1	Micro-precision	Micro-recall
Java	Import2Vec	**0.0009**	**0.5557**	**0.7787**	**0.4321**
	CodeBERT	0.0011	0.3254	0.7160	0.2106
Python	Import2Vec	**0.0006**	**0.7993**	**0.8789**	**0.7330**
	CodeBERT	0.0010	0.6499	0.8089	0.5432

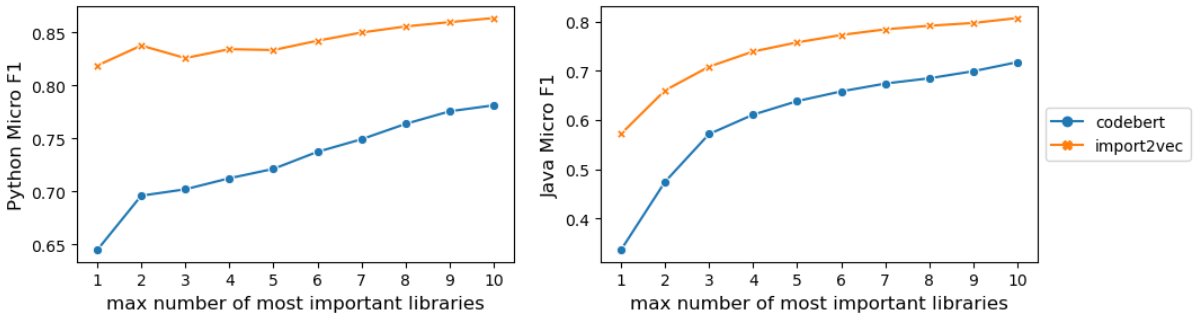

Figure 12. Library prediction subtest-2: Micro-F1 Score

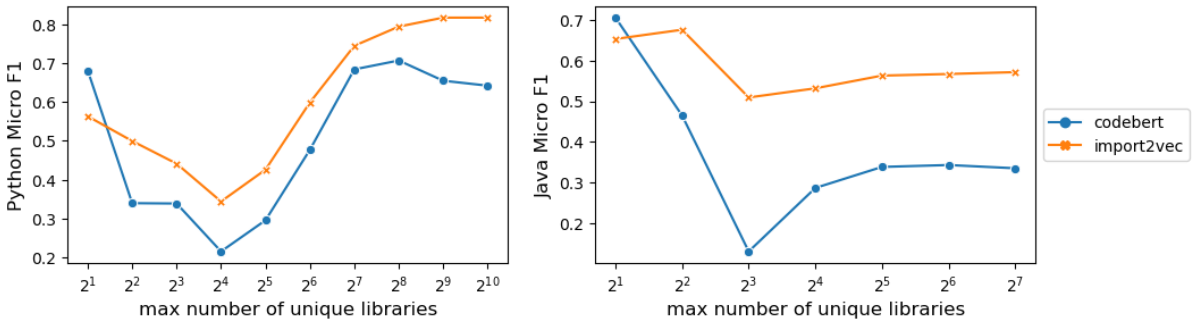

Figure 13. Library prediction subtest-3: Micro-F1 Score

After the evaluations, I obtained the best embedding for each data source, and the results are consistent for both Python and Java repositories:

- Co-contributor network: Node2Vec;
- Readme text: Doc2Vec;
- Import sequence: Import2Vec.

All the best models are derivatives of Word2Vec, which suggests that their embedding strategies are suitable for GitHub Data. When large-scale data are available, these models are competitive in generating paragraph-level or document-level embeddings for domain-specific corpora. Pretrained transformers such as BERT and its derivatives, although powerful, might not always be ideal for domain-specific data and for representation learning of documents or paragraphs. Therefore, scholars need to carefully consider the source, volume, and granularity of the data to be represented in their model selection process.

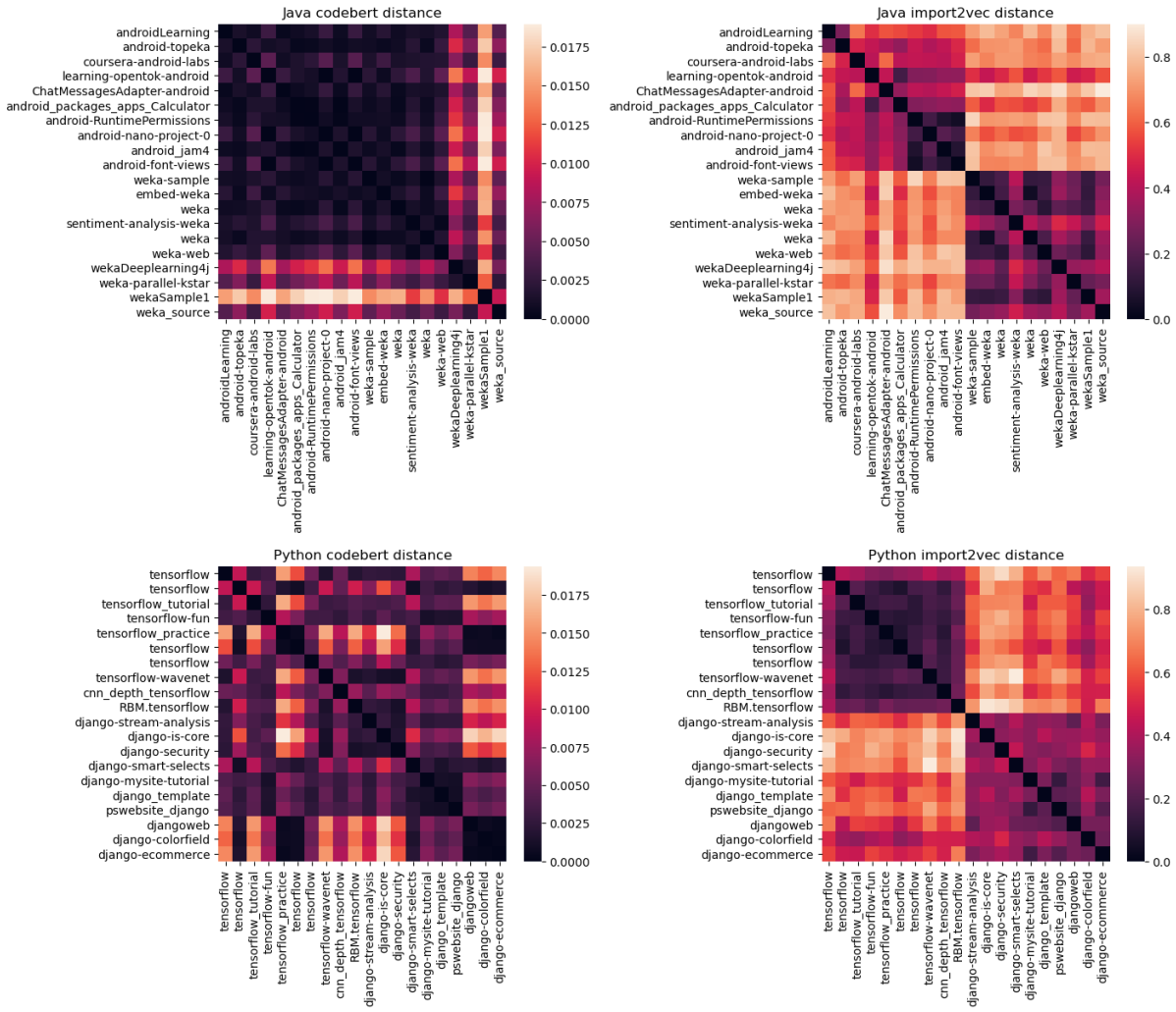

Figure 14. Cosine distance matrices for selected import embeddings

5 Socio-functional Mapping

This section examines the mapping between contributor groups and repository functions and pinpoints the difference in the socio-functional structures between the Python and Java communities. Besides, it also provides insights into practical areas such as recommendation systems in OSS. Specifically, the mappings are quantified according to the consistency between the best embedding spaces through canonical correlation analysis (CCA) (Wegelin, 2000) and neighbor distance comparison. The analysis here also indicates the potential of embedding-based approaches in uncovering behavior patterns in social coding.

5.1 Method

The consistency between the different embeddings of the same entities (repositories) can be measured globally or locally. If examined globally, measuring the consistency can be transformed into a correlation analysis between two matrices in which each row refers to the embedding vector of a repository. Beyer et al. (2020) proposed the application of CCA to measure domain similarities and verified its effectiveness by comparing the detected correlations with human intuition. In canonical correlation analysis, the compared embedding spaces are projected into a shared space (usually with a lower dimensionality) in which their correlations are maximally extracted via linear combination. Each dimension in the projected space could be considered a latent variable responsible for data generation in the original spaces. Mathematically, the projection process can be expressed as finding the transformation matrices V and W for the original embedding spaces X and Y such that the transformed matrices $X' = XV$ and $Y' = YW$ have maximal dimension-wise correlations. The first dimension (component) in the projected space accounts for the highest covariance between the two original spaces, and the lower the dimension ranks, the less the covariance it covers. In previous applications (Beyer et al., 2020), researchers have used the maximal or the mean dimension-wise correlation between the transformed spaces to measure the correlation or similarity between the original domains. In my exploration, I projected the embedding spaces into a ten-dimension space and then visualized the distribution of dimension-wise correlations to examine the space consistency. Moreover, concerning the scenario of recommendation systems in which the popularity of repositories is considered, I also explored how the space consistency changes with repositories of different popularity levels. The repositories are sorted by the number of stars in a descending order and then equally divided into ten folds (i.e., the first fold contains repositories with the highest number of stars). The space consistency in each fold is captured with the maximal canonical correlation (first-dimension correlation).

When examined locally, the consistency between the embedding spaces can be represented through the consistency of the repository neighborhood. Previous studies have used the ratio of shared neighbors in two embedding spaces or the Jaccard index to quantify local similarity (Boggust et al., 2019). In my study, I used an ideologically similar but distance-based measurement to check the neighborhood consistency:

1. Given two embedding spaces *A* and *B* and a neighbor threshold *d*, select a repository *r* and find its neighbors in space *A* whose cosine distance to *r* is less than *d*.
2. Compute the average cosine distance between the neighbors and *r* in space *B*.
3. Replicate steps 1 and 2 for all the repositories and obtain the mean average cosine distance.
4. Replicate step 3 for a sequence of *d*, and record the change of mean average cosine distance.

If the two spaces are consistent, the mean average cosine distance in space *B* is expected to increase steadily as the threshold *d* for neighbor identification expands.

Another merit of this is strategy is that it deconstructs socio-functional mapping patterns into the social dispersion of functionally similar repositories and the functional diversity of socially connected repositories. This further helps illustrate the potential discrepancy between Python and Java. For instance, suppose the neighbors of the import/readme embedding space have a larger average distance in the social space for Python repositories. This implies that the dispersion among Python repositories with similar libraries/functions is more intensive than in Java repositories. On the other hand, if neighbors in the social space are more distant in the import and readme space for Java repositories, it suggests that the Java repositories with similar contributor compositions are more diverse in functionality.

Since the consistency investigation requires different embeddings of the same repositories, I

have only included Python and Java repositories having all three embeddings for this analysis. Approximately 190k Python repositories and 176k Java repositories were included in the dataset.

5.2 Results

CCA

The results of the canonical correlation analysis on all selected repositories are shown in Fig. 15. Before discussing the socio-functional mapping, I will briefly comment on the functional embeddings. We see that the import and readme spaces have the highest consistency compared with the other space pairs for both languages. The consistency between import and readme spaces also provides an alternative data source for recommending repositories based on functional similarities. Previous research has reported that API calls (imports) in code snippets can serve as an excellent source to recommend functionally resemblant repositories (McMillan, Grechanik, & Poshyvanyk, 2012; McMillan, Grechanik, Poshyvanyk, et al., 2012). However, as the number of repositories and the code volume accumulate, this design becomes increasingly expensive since it requires the iteration of code files to update the system. Compared to the API calls, readme seems to be a more economical yet reliable source because each repository only has one readme file and is less frequently modified; therefore, the system's renewal cycle could be longer. Recent research has attempted to incorporate readme embeddings to recommend relevant repositories from one thousand most popular Java repositories; this has achieved comparable performance as the code-based approach (Yun Zhang et al., 2017). My results agree with this idea and verify its feasibility for application to more Python and Java repositories.

In contrast, the mapping between social and functional spaces is quite different for the two communities. While Java repositories still pertain to a certain level of consistency between the

social space and the functional spaces, Python repositories do not imply such correlations. This highlights two pitfalls behind the current research on OSS recommendation systems. First, researchers might overlook the multifacetedness of the definition of relevance and continue to use a single and over-general criterion to evaluate these systems. In previous studies, a widely used method to compare different systems involves human annotators to score the relevance of the recommendation results (McMillan, Grechanik, & Poshyvanyk, 2012; McMillan, Grechanik, Poshyvanyk, et al., 2012; Yun Zhang et al., 2017). For instance, Zhang et al. (2017) asked college students with programming experience to rank how each recommendation result related to a given repository from "Highly Irrelevant" to "Highly Relevant". However, an arbitrary or omniscient relevance between the repositories is hard to measure and may not exist, as my result indicates. In the actual evaluation, annotators tend to possess bias such as weighing functional relevance more than social relevance, which benefits an import-based or readme-based system but belittles a contributor-based one. Accordingly, the authors may conclude that an import-based system is better than a contributor-based system, thereby neglecting the potential merit of the contributor-based system in making social recommendations. This situation is similar to using a single but biased task for embedding the evaluation described in Section 4.3. To better illustrate this, I plotted the Import2Vec and the Node2Vec spaces for the Python repositories selected for visualizing topic tag clusters (see Fig. 16). Based on the graph, we admit that the Node2Vec embedding performs worse in representing topic tag clusters. However, we cannot make a general conclusion that it is a poor representation of the repositories. The topic tag prediction reflects the models' ability to capture the repository functionality, but the Node2Vec embedding is trained on co-contributor links. Thus, using topic tag prediction to evaluate Node2Vec overlooks its capacity in mimicking social relationships. The "bias" in topic tag prediction is designed solely for readme embedding evaluation. However, in a recommendation system evaluation, the bias is usually obscure. To reduce such

bias in human annotation, researchers need to notify the annotators about the specific relevance they expect from the recommendations (e.g., whether the recommended repositories should be relevant in functions or contributors). In this manner, we may present the results by saying that the import-based system is better for recommending functionally related repositories than saying that it is better or worse in general.

The other pitfall relates to the generality of recommendation methods. The difference between Python and Java repositories implies that a contributor-based system might predict functionally similar repositories in Java with a certain accuracy but is unlikely to achieve the same for Python repositories. Such language-specific efficacy tends to occur when the data source of the recommendation system does not directly match the recommendation task. Matched situations would entail making a functionality-based recommendation with an import-based system or making a social recommendation with a contributor-based system. For unmatched cases, designers should be more careful when stating the generality of their methods, especially when the methods are only being tested for projects in a single programming language.

The consistency change results shown in Fig. 17 suggest that the consistency between spaces is higher for repositories with higher popularity, providing additional advice for recommendation systems. Previous research has often tested recommendation systems on the most popular repositories (Matek & Zebec, 2016; Yun Zhang et al., 2017). Though this filtration is beneficial regarding computational feasibility and data completeness (mediocre repositories may lack the data that some recommendation systems need, e.g., readme text), it also makes the generality of the system to developing repositories, which are temporarily less popular, unknown. According to my results, using social embedding to recommend functionally similar repositories might work for top Java repositories (batch index 1, correlation close to 0.6) but is unlikely to operate well for the less popular ones (e.g., batch index 3 and bigger, correlation around 0.3). However, developing repositories usually demand more

recommendations than well-established ones for seeking collaboration opportunities and learning better implementations. Therefore, future studies should pay attention to whether their methods function well on start-up repositories or projects.

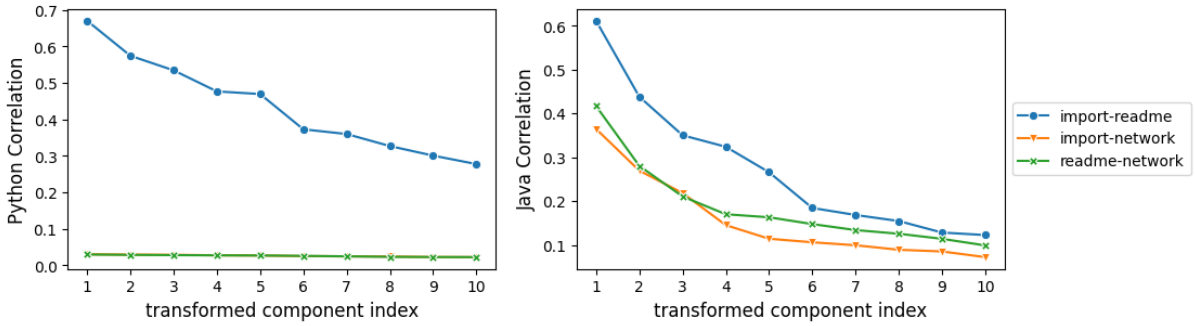

Figure 15. CCA result: dimension-wise correlations (component index means dimension index)

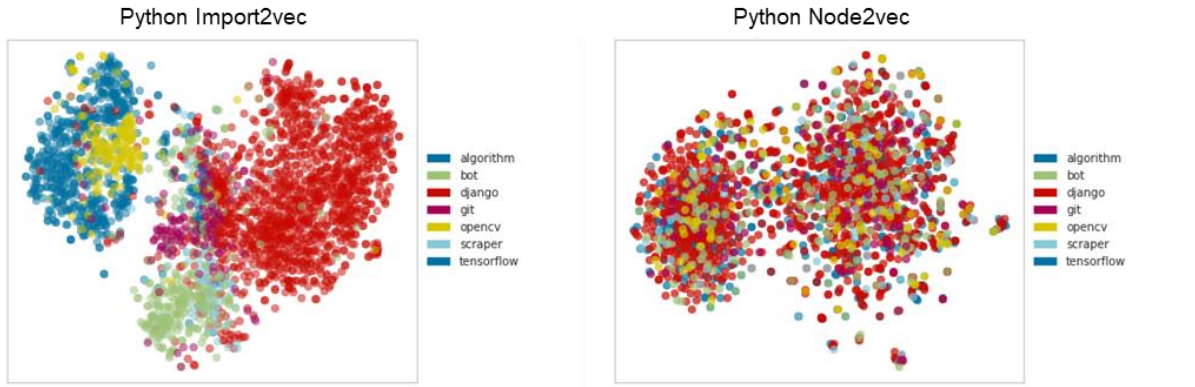

Figure 16. Import and network embedding space for selected topic tags

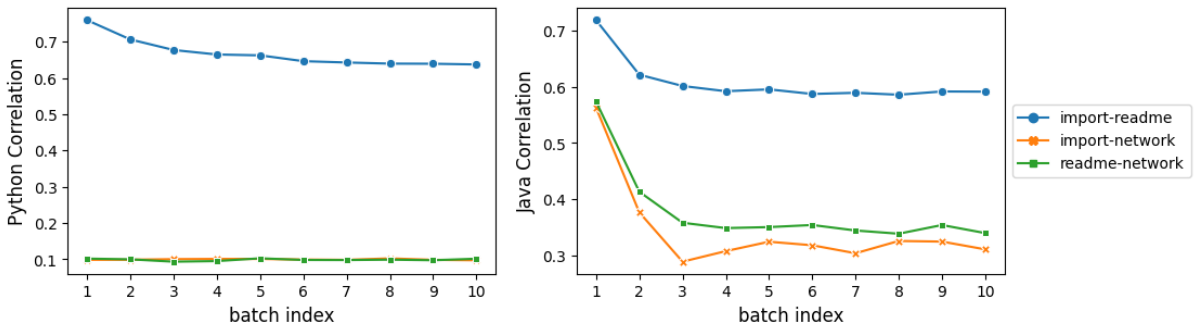

Figure 17. Consistency change for repositories conditioned by stars (lower index means more stars)

Neighbor Distance

The results of neighbor distance comparison provide insights into the observed language-specific heterogeneity in socio-functional mapping. Fig. 18 presents the comparison of dispersion in functionally similar repositories between the two languages. The trend is consistent with both import or readme to anchor the neighbors but is more evident for readme neighbors. As the threshold for neighbors loosens (more remote repositories are included), the mean distance between the neighbors and the central repository in the social space also increases for Java but remains stable for Python; this supports the CCA results. More importantly, the mean network distance for neighboring repositories in Python is substantially larger than in Java, especially when the neighbor threshold is small. This divergence suggests that the social dispersion for functionally similar repositories is more intensive in the Python community. Fig. 19 compares functional diversity in socially close repositories between Java and Python. Python neighbors have larger mean distances in functional spaces compared with Java neighbors. This indicates that socially connected Python repositories have more dissimilar functions than their Java counterparts.

When the two perspectives are viewed as modes in socio-functional mapping, more social dispersion in functionally similar repositories reflects a many-to-one mode, and higher diversity in socially connected repositories corresponds to a one-to-many mode. When a community is intensive in both directions, as the Python community reveals, it can be depicted as a many-to-many state (see Fig. 20). On the other hand, the Java community would be closer to a one-to-one mode where the consistency between functional and social spaces is more explicit. Apart from characterizing the mapping mode between contributors and repository functionality, the two perspectives also lead to valuable social interpretations. First, the dispersion perspective relates to the competition between functionally similar repositories, especially for the non-trivial repositories. A higher social dispersion for Python indicates more

competition among the repositories because they are more likely to be developed by sparsely connected contributor groups, the members between which have little or no collaboration. Second, the diversity perspective can hint at contributors' committing preference in terms of exploitation or exploration (March, 1991). Compared to Java contributors, Python contributors tend to commit to repositories less similar in functionality, which corresponds to the favor of exploration. In social sciences, exploration is associated with knowledge creation and innovation diffusion (Y. Li et al., 2008; Uzzi et al., 2013). In this regard, the favor of exploration implies that the Python community is more active in these activities, and summary statistics, including the average number of unique imports (see Section 4.1) and the total number of "good" imports (Python: 36485, Java: 3681), also advocate this inference.

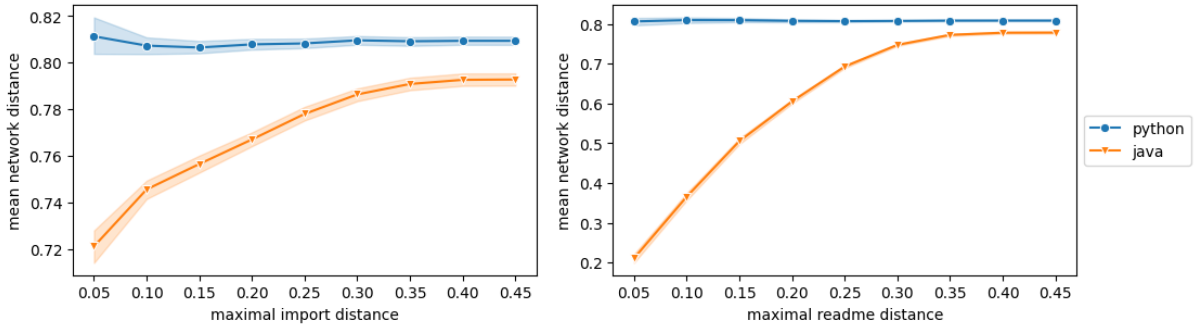

Figure 18. Comparison of dispersion in functionally similar repositories

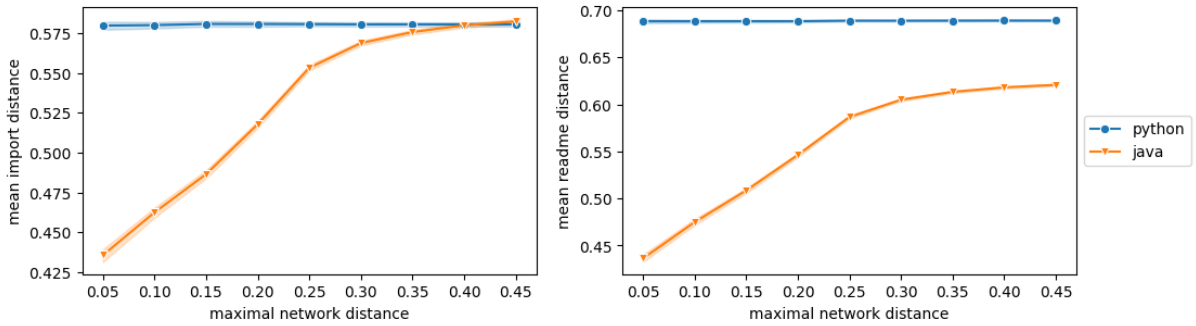

Figure 19. Comparison of diversity in socially close repositories

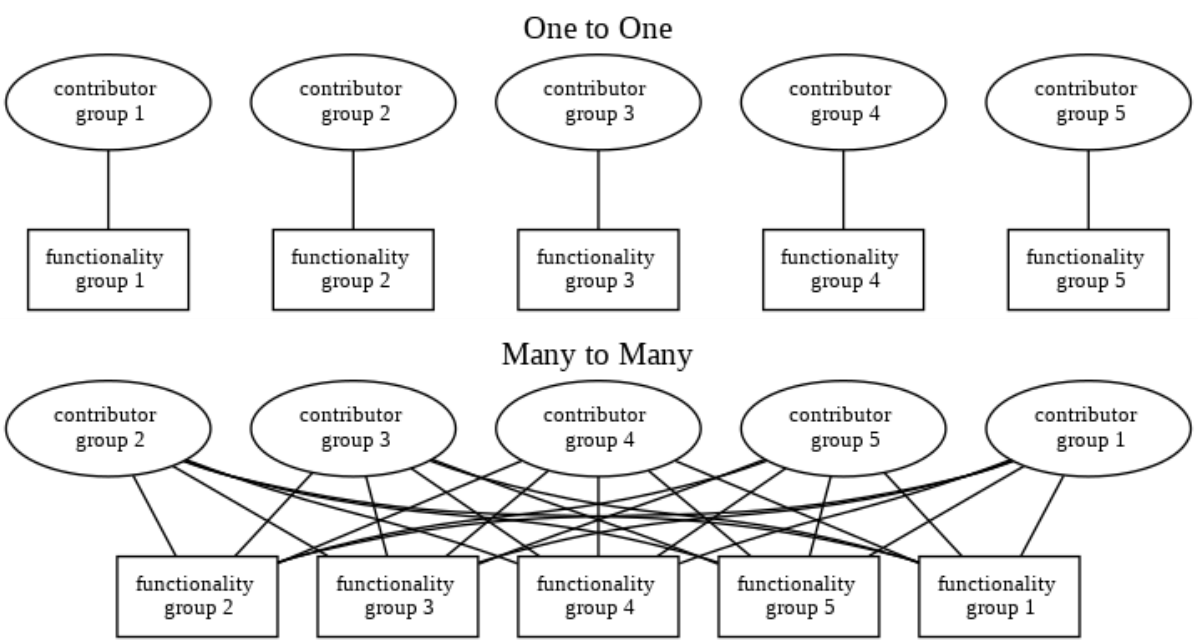

Figure 20. Ecosystem mapping mode

6 Contributing Diversity Analysis

Motivated by the results in Section 5, I took a step further to identify the possible reasons for the exploration preference of Python contributors. I propose that programmers who can contribute to functionally diversified repositories are more likely to be the contributors of popular repositories in Python, which drives them to explore functionally different repositories more than Java contributors. This hypothesis is tested and discussed in this section.

6.1 Method

The hypothesis is examined by calculating the correlation between coders' contributing diversity and the average popularity of the contributed repositories. I defined the contributing diversity of a contributor *c* as the mean value of the cosine distances between the functional embeddings of the repositories contributed by *c*. This can be denoted as follows (R_c is the set of embeddings for repositories contributed by *c*) :

$$Diversity(c) = \frac{2}{|R_c|(|R_c|-1)} \sum_{\{i,j|i,j \in R_c, i \neq j\}} \left(1 - \frac{i \cdot j}{\|i\| \|j\|}\right)$$

To get rid of the repositories built only for demo testing or code storage and to capture the repositories that the contributors actively contributed to, I only included repositories with at least five[12] stars and committed at least five[13] times by a given contributor. Contributors with only one valid repository were treated as zero-diversity contributors, and contributors with no valid repositories were removed from the dataset. Table 7 contains the number of contributors in the final dataset.

Table 7. Contributor count summary

Language/Embedding	Import	Readme
Python	80836	74678
Java	59775	53325

The popularity is quantified by the average[14] number of stars of the repositories, and the final correlation is measured with the Pearson correlation coefficient.

6.2 Results

Fig. 21 presents the mean contributing diversity of the programmers conditioned by the number of contributed repositories. The average diversity level of Python coders is higher than that of Java coders in both functional spaces regardless of the thresholds of repository counts; this supports the findings from the previous section.

[12] Star thresholds between 2 and 5 are tried, where the correlation trends are consistent.
[13] Commit thresholds between 2 and 5 are tried, where the correlation trends are consistent.
[14] Total and median number of stars are also tried, where the correlation trends are consistent.

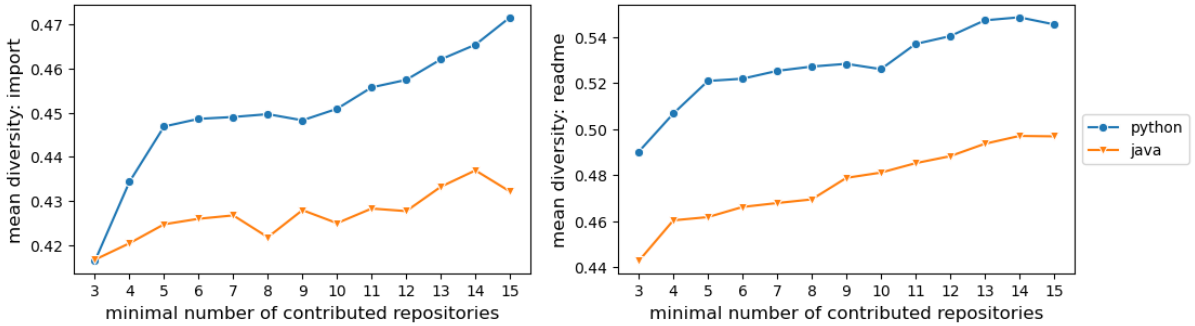

Figure 21. Comparison of mean contributor diversity

The results of the correlation test are presented in Fig. 22. The correlation between their contributing diversity and the average repository popularity for Python contributors is positive and more potent than that for Java programmers in terms of significance and magnitude. Furthermore, the discrepancy between the two communities is more evident for contributors who committed to a larger number of repositories where the correlation coefficient increases for Python but becomes non-significant for Java. Such tendencies are consistent for both import and readme spaces, which validates my hypothesis regarding contributors in Python and Java communities. Additionally, to preclude the possibility that the correlations are simply rooted in the number of contributed repositories instead of the diversity, I also examined the correlation between the number of contributed repositories and their average popularity. The result revealed no significant correlations ($p > 0.1$).

The validity of my hypothesis indicates that exploring functionally different projects is more valued in Python than in Java. In other words, competitive Python contributors usually participate in the developments of repositories from multiple areas. To achieve this, coders might choose to increase the variety of their coding skills (e.g., becoming proficient in both data mining and web design) or increase the expertise in one coding area demanded by different fields (e.g., database management is needed for both data analysis and web development). These choices could lead to new research questions regarding the “exploration vs. exploitation”

tradeoff in coding skills, discussed as future work in Section 8. Irrespective of the choice, the compatibility between expertise and various functional needs is the premise for exploring diversified repositories.

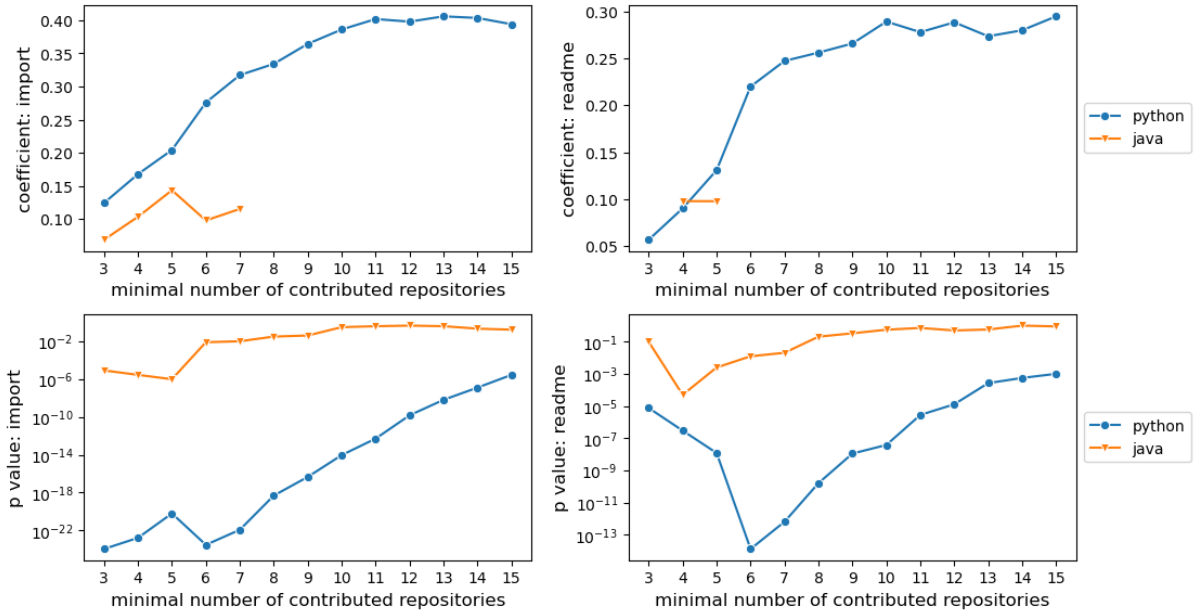

Figure 22. Correlation between contributor diversity and popularity; only significant coefficients ($p < 0.01$) are plotted

7 Repository Popularity Prediction

In this section, I investigate whether the embeddings contribute to predicting the popularity of repositories and which parts in the embeddings are the most helpful. These questions illustrate whether an embedding-based approach can benefit practical tasks such as popularity prediction. For non-forked repositories, popularity metrics are usually highly correlated (see Fig. 23). Borges & Tulio Valente (2018) suggest that GitHub users think of stars as the most reliable metric for popularity measurement. This encouraged me to use stars as the raw popularity indicator. Different combinations of embeddings are tested to obtain the best performance, and the employed dataset is the same as the one in Section 5.

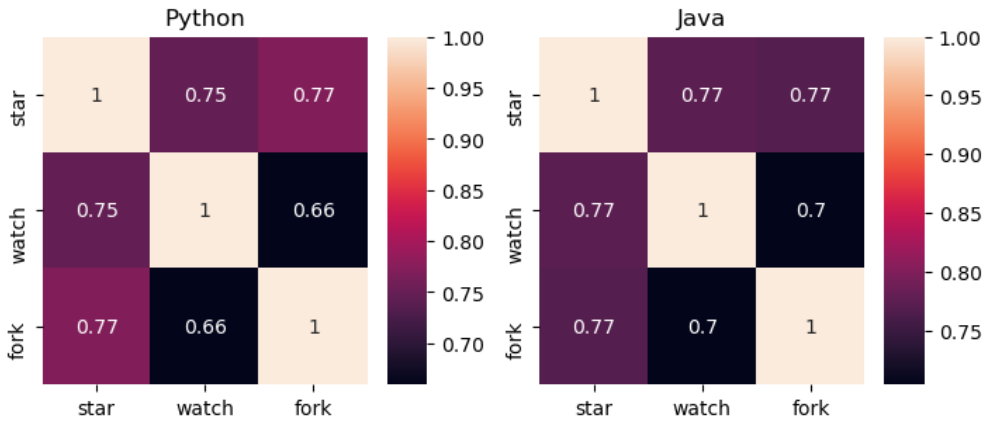

Figure 23. Correlations between popularity metrics

7.1 Method

In GitHub, the repository stars are found to obey a power-law distribution (Lima et al., 2014), which also suits my dataset (see Fig. 24). This distribution suggests that only a few repositories have a non-trivial number of stars. Based on this property, I designed two sub-tests for popularity prediction:

- Predict whether a repository has valid stars (classification) and valid stars implies that the number of stars is larger than one (the owner may star the repository). The prediction targets are binary, and 1 means the repository has valid stars. In this sub-test, the predictor would suffer from label imbalance because of the power-law distribution. To ensure that the samples in the minority class (label 1) are not less than half of the majority class, I deployed the SMOTE algorithm (Chawla et al., 2002) to implement over-sampling on repositories with label 1. This technique synthesizes new samples by creating data points between the nearest neighbors in the minority class. This process was performed after train-test splitting, and only the train set was over-sampled. Model performance is evaluated by accuracy, precision, recall, and ROC-AUC score (Bradley, 1997), where a higher value indicates better performance for all the metrics.

- Predict the popularity level of a repository (regression), and the popularity level here equals the logarithmic value of the number of stars. Only repositories with valid stars, covering about 56k Python repositories and 36k Java repositories, are included for this sub-test. Model performance is evaluated by explained variance score (EV) and mean squared error (MSE). A higher EV or a lower MSE implies better performance.

For both sub-tests, a random forest estimator with 500 trees was trained on 80% of the dataset and tested on the remaining data. For each model, the prediction task was replicated ten times with different train-test splits, and the final performance was determined by the average values of the results. The random forest estimator in *scikit-learn* has built-in feature importance scores after fitting, which are defined as the normalized total reductions on loss brought by each feature. I generated the scores by fitting the best model of the regression task with the entire dataset and utilized these scores to identify the most informative dimensions.

Previous research in Github popularity prediction indicates that metadata such as the number of commits, number of contributors, and length of readmes effectively predicted the popularity of repositories (Han et al., 2019; Weber & Luo, 2014). Therefore, I built a baseline model with only the metadata described in Section 3 (see Table 8). The embeddings can be compared to the baseline features both independently and jointly. Independent models used embeddings from one data source, and joint models concatenated multiple embeddings or added embeddings to baseline features.

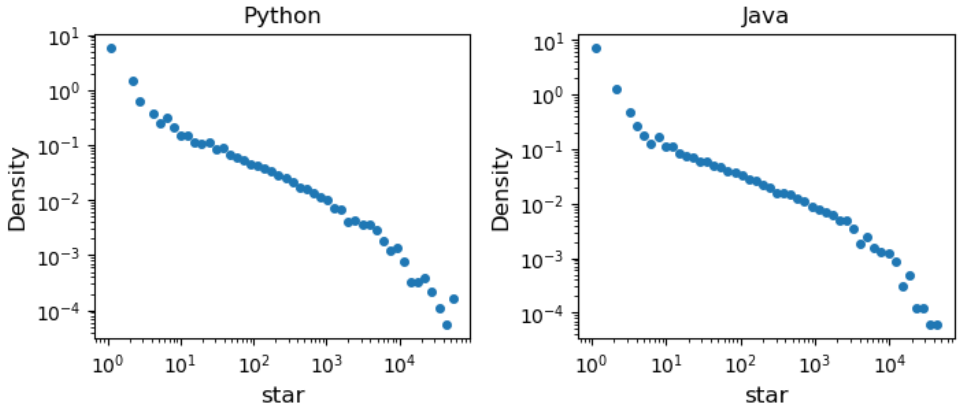

Figure 23. Repository star distribution

Table 8. Baseline features

Source	Metadata
Source Code	the number of unique import packages
	the number of programming languages
	the number of files
Contributor List	the number of contributors
	the number of commits
	the average number of commits per contributor
	the degree in the co-contributor network
Readme File	the length of cleaned readme text

7.2 Results

Tables 9 and 10 present the prediction results. For both languages, the addition of certain embeddings promotes model performance for the two sub-tests. For Python repositories, the combination of import embedding, readme embedding, and baseline features outperforms other models in most metrics for both sub-tests. For Java repositories, the introduction of network embedding benefits the classification task the most, and the combination of network embedding, readme embedding, and baseline features stands out in the regression task. Especially, the

precision (classification) and EV (regression) of the best model increased over 10% compared to the baseline model for both languages. Moreover, even within models with a single feature source (baseline, import, readme, or network), the baseline model is overshadowed by purely embedding-based models except for the regression task in Python.

Despite the improvement in prediction performance, the results suggest two points to consider when using embedding-based features. **First**, simply concatenating all the embeddings and baseline features (import+readme+network+baseline) might not yield the best results. The idea of "the more, the better" does not apply to the prediction task here, and the same may apply in other downstream tasks as well. Adding non-informative embeddings might worsen model performance, so examining different embeddings and embedding combinations is necessary to find the optimal model. **Second**, the efficacy of embeddings from the same source can be significantly different in different language communities. For example, network embedding, which offers minimal boons in Python tasks, contributes the most to Java sub-tests. On the contrary, import embedding, while useful in Python tasks, is the least valuable embedding for Java sub-tests. The aforementioned facts indicate the issue of language heterogeneity in popularity prediction, which has not been emphasized in previous studies. One of the reasons for such heterogeneity being overlooked is associated with data. The data in these studies mostly comprised metadata and summary statistics (Borges et al., 2016; Han et al., 2019; Weber & Luo, 2014), some of which tend to contribute consistently to popularity prediction tasks regardless of programming languages. For instance, a larger number of contributors signals more popularity. This is also confirmed in my dataset since the correlation between the numbers of contributors and popularity level is significantly ($p < 0.001$) positive for both languages (Python coefficient 0.28 and Java coefficient 0.22). However, my results suggest that to make the most of embeddings, researchers need to consider language heterogeneity in feature engineering and may change their goals to searching for the best models in each

language instead of a general one. Besides, from the perspective of model generality, the best model for some repositories of mixed languages might not be ideal for the repositories of a specific language. Likewise, the best model for one language community may be terrible for another. Therefore, running experiments on both general and language-specific datasets can be necessary if high generality is the goal.

Fig. 24 displays the importance scores for the top 10 features in the regression task. For baseline features, only the number of contributors and the length of the cleaned readme ranks high for both languages. The Java regressor also weights the number of commits and the degree in the co-contributor network higher than 90% of features. Embedding-based features occupy more than half of the positions in the top ten for both languages, which confirms their capacity in these prediction tasks. In Python, the most important embedding-based features are dominated by dimensions in readme embedding, where the 50^{th} dimension ranks the highest among all embedding-based features. In Java, network-based dimensions are more valued, and the fourth dimension is the most important among all features. Additionally, although understanding the dimensions in these embeddings can be relatively complex because the embedded information is highly abstract and aggregated, I obtained clues to capture the meaning of several dimensions by querying the repositories with the highest or lowest values for these dimensions. For instance, the repositories receiving the highest values in the 42^{nd} dimension of the Python readme embedding are related to deep learning models such as convolutional neural networks (CNN). Furthermore, repositories with the lowest values in the fourth dimension of the Java network are related to the embedding center on the Netflix[15] developer group (see Table 11).

From the results, it is evident that functionality plays a more decisive role in a Python repository's popularity, while social relationships are more valued for the popularity of Java

[15] Polyglot and Gradle are important tools for Netflix development, see information at https://www.infoq.com/news/2018/08/better-devex-at-netflix/, https://gradle.com/next-developer-productivity-engineering-meetup/

repositories. Such divergence might be caused by the different application scenarios or markets of the two programming languages, which are argued in the next section.

Table 9. Python popularity prediction results

Features	Classification				Regression	
	Accuracy	Precision	Recall	ROC-AUC	EV	MSE
baseline	0.7405	0.5862	0.4104	0.7329	0.2842	1.8001
import	0.7358	0.6428	0.2357	0.7115	0.1461	2.1495
readme	0.7455	0.6369	0.3202	0.7523	0.1798	2.0628
network	0.6835	0.2967	0.0530	0.5006	-0.0357	2.6093
import+readme	0.7544	0.6718	0.3276	0.7668	0.2274	1.9423
import+network	0.7309	0.6475	0.1939	0.7005	0.1274	2.1960
import+baseline	0.7682	0.6796	0.4061	0.7779	0.3508	1.6334
readme+network	0.7377	0.6406	0.2529	0.7411	0.1590	2.1149
readme+baseline	0.7724	0.6837	**0.4257**	0.7879	0.3780	1.5635
network+baseline	0.7434	0.6117	0.3576	0.7389	0.2795	1.8128
import+readme+network	0.7494	0.6737	0.2924	0.7595	0.2145	1.9745
import+readme+baseline	**0.7747**	**0.6971**	0.4185	**0.7908**	**0.3822**	**1.5530**
import+network+baseline	0.7636	0.6780	0.3791	0.7690	0.3331	1.6775
readme+network+baseline	0.7670	0.6782	0.4005	0.7779	0.3592	1.6107
import+readme+network+ baseline	0.7704	0.6948	0.3958	0.7841	0.3692	1.5854

Table 10. Java popularity prediction results

Features	Classification				Regression	
	Accuracy	Precision	Recall	ROC-AUC	EV	MSE
baseline	0.8101	0.5610	0.3713	0.7600	0.2677	2.0408
import	0.7950	0.5143	0.1372	0.6832	0.1045	2.4996

readme	0.8088	0.5727	0.2945	0.7815	0.1989	2.2321
network	0.8136	0.5681	0.4078	0.7719	0.3295	1.8685
import+readme	0.8120	0.6034	0.2630	0.7867	0.2310	2.1419
import+network	0.8132	0.5826	0.3380	0.7685	0.3280	1.8719
import+baseline	0.8169	0.5967	0.3513	0.7800	0.3252	1.8815
readme+network	0.8222	0.6128	0.3783	0.8019	0.3679	1.7589
readme+baseline	0.8303	0.6426	0.4021	0.8193	0.3833	1.7174
network+baseline	**0.8331**	**0.6501**	**0.4161**	0.8130	0.3755	1.7396
import+readme+network	0.8208	0.6091	0.3701	0.7994	0.3719	1.7478
import+readme+baseline	0.8290	0.6434	0.3871	0.8145	0.3795	1.7279
import+network+baseline	0.8271	0.6316	0.3919	0.8053	0.3723	1.7481
readme+network+baseline	0.8318	0.6455	0.4122	**0.8245**	**0.4072**	**1.6495**
import+readme+network+ baseline	0.8292	0.6375	0.4021	0.8197	0.4037	1.6592

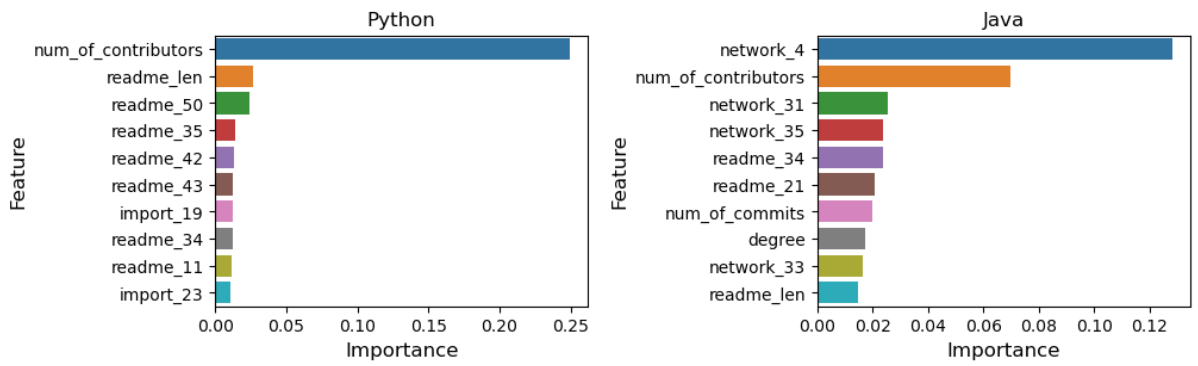

Figure 24. Feature importance for top 10 features in regression sub-test

Table 11. Head/Tail repositories by dimension values

Python readme dimension-42		Java network dimension-4	
Value	URL path	Value	URL path
1.8156	zuowang/deep-fashion	-12.2640	eclipse/vert.x

1.6457	tscohen/gconv_experiments	-12.2992	michaelklishin/quartz-mongodb
1.6025	conansherry/mx-rcnn	-12.3346	Netflix/EVCache
1.5948	gplhegde/theano-xnor-net	-12.7266	Netflix/exhibitor
1.5403	zy97140/omp-benchmark-for-pytorch	-12.9214	gradle/gradle
1.5258	wk910930/py-faster-rcnn	-12.9942	asciidoctor/asciidoctorj
1.5218	ucloud/uai-sdk	-13.0219	Netflix/feign
1.5175	lzx1413/PytorchSSD	-13.1804	detro/ghostdriver
1.4958	BIGBALLON/cifar-10-cnn	-13.5822	Netflix/Hystrix
1.4946	SnippyHolloW/timit_tools	-13.7194	takari/polyglot-maven

8 Discussions and Future Work

This section discusses the technical and social implications of the present thesis. Especially, the language-specific heterogeneity between Python and Java from the findings is explained. It also proposes future research directions based on questions raised in previous sections.

Technically, this study differs from other social coding research by characterizing repositories with embeddings from content and contextual data. This approach allows us to represent repositories from functional and social perspectives and combine the information from both sides quantitatively. Future studies on the analysis of repository competitions may benefit from this. For example, in Section 5, I considered repositories similar in functionality but distant in social space as potential competitors. This idea can be further specified by selecting a distance threshold to better identify the competitors. Based on the identified competitors, we can capture their current competing results with popularity metrics (e.g., if a repository has more stars than the average value of repositories in the same competition group, then it is considered a winner)

and explore whether embedding-based features are efficacious in predicting the results. Apart from competition analysis, high-quality embeddings may also improve the accuracy for other downstream tasks such as predicting the ratio of solved issues, where repositories' functional or social individuality is valued.

Despite the benefits of embeddings, I also acknowledge the obstacles in using this method. Generating high-quality embeddings requires more computational efforts in data collection, preprocessing, and model fitting than simply using metadata. Therefore, the dataset generated from this research is a valuable resource for future research. Moreover, the details provided in the representation learning pipeline (see Section 4) can be referred to as instructions for embedding generation in future work.

Socially, this research makes an in-depth investigation into community differences between two mainstream programming languages on GitHub: Python and Java. To comprehend the causes for such divergence, we may trace the origin of the two languages. Python was initially released in the 1990s with a philosophy that stressed code readability. The famous work Zen of Python[16] by Tim Peters unveils the essence of Python's philosophy:

"Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts."

In contrast, the philosophy of Java highlights the independence of code, stating "Write Once, Run Anywhere" (Arnold & Gosling, 2000), suggesting that Java programs are designed to be executed with as few dependencies as possible.

The discrepancy in philosophy renders a substantial difference in program extensibility. For

[16] https://www.python.org/dev/peps/pep-0020/

Java projects, including external libraries requires coders to manually download source files and add their paths to the compiler. While in Python, this can be achieved with much less effort by entering the "pip install" command. The philosophical difference also results in different grammar rules. For instance, Python allows coders to declare variables without specifying their types (e.g., integer, float, boolean, or customized data type), but Java requires explicitly stating data types for variables. Furthermore, Python enables coders to write for loops quickly with the keyword "in" or with a list comprehension, but Java requires writing it as a function (see Table 12). Besides, Python also omits semicolons at the end of commands and curly braces in the definition of new functions. These differences accentuate the flexibility and conciseness of Python coding and the completeness and concreteness of Java coding; this determines the divergence in their application scenarios.

Python's extensible and succinct style makes the diffusion of code much more effortless, meeting the demand for areas like data science in which researchers highlight the replicability of code. Moreover, the introduction of IPython and Jupyter notebook[17] has empowered Python to become an ideal tool for data analysis and visualization because users can interact with the code instantly and integrate code, annotations, and graphs in one file that can be shared effortlessly. In comparison, the ideology behind Java coding makes it more suitable for programming at the engineering level. Mobile applications and software products are the primary battlefields of Java developers, and relevant code is only shared within developing teams. From the perspective of product markets, the targets of Python contributors are mainly composed of peers who also write Python code. In contrast, Java contributors focus more on end-users who ask for handy software without the need to know the underlying Java programming (see Figure 25).

[17] https://ipython.org/

This difference helps demonstrate why network embedding is essential in the popularity prediction of Java repositories, and functional embeddings are informative for reflecting the popularity of Python repositories. Competitive Java repositories are probably monopolized by large companies or platforms such as Netflix and Gradle, which serve as hubs of proficient Java developers and paradigmatic Java frameworks and libraries. Meanwhile, Python repositories have gained popularity more because they realize functions valuable for other Python users. One way to achieve this is to improve or extend existing functionality, which fosters more competition among functionally similar repositories in Python, as Section 5 implies. Besides, Python's accentuation of functionality also corresponds to its contributors' appreciation of the functional diversity among contributed repositories. Such diversity demonstrates their ability to fulfill different functional needs from others, while Java contributors tend to specialize in one or few fields, as Section 6 indicates.

The difference between Python and Java communities also suggests that future research needs to attribute more importance to language-specific heterogeneity regardless of predictive or explanatory issues. The strategy of using language-specific datasets is fundamental to describe GitHub communities of other programming languages that have typical application scenarios (e.g., JavaScript) and to learn how the philosophy behind a programming language affects its socio-functional structure.

Table 12. Comparison between Python and Java syntax

Syntax	Python	Java
Variable Declaration	x = 1	int x = 1;

Loop Expression	for i in range(5):
	  function(i)
	OR
	[function(i) for i in range(5)]
		for (int i = 0; i < 5; i++) {
		  function(i);
		}

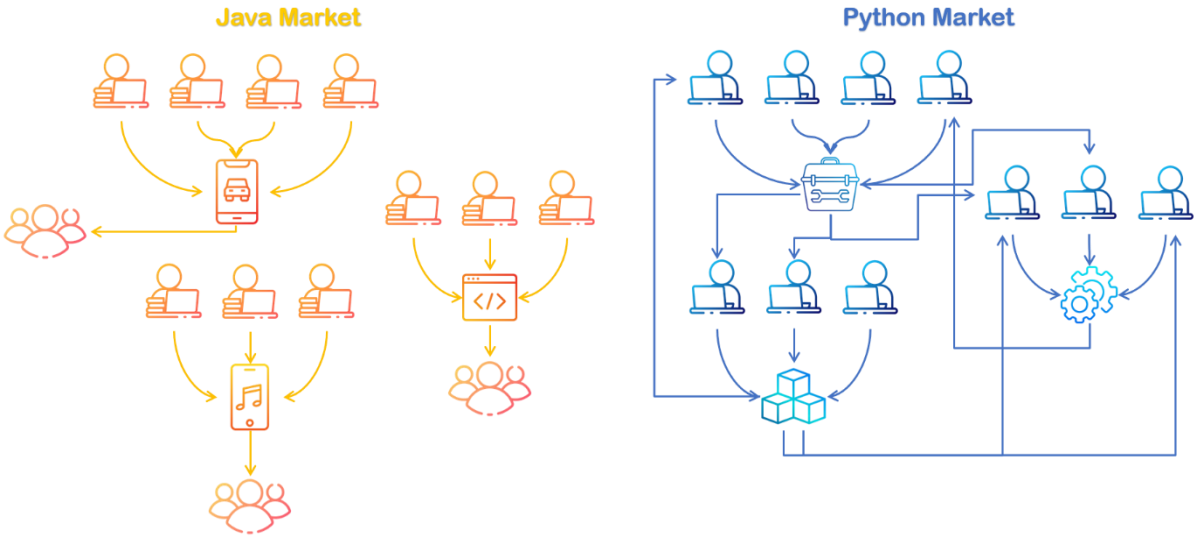

Figure 25. Typical Python and Java market structure on GitHub

Though profited from repository embeddings, this research is also limited to the view of GitHub repositories. However, the answers to some questions require directly organizing data in the unit of contributors, such as exploitation and exploration in contributors' coding skills (see Section 6). This question could be answered by probing the content of contributors' commits to see whether they make similar contributions (e.g., using similar libraries) to the repositories or not. Another contributor-driven question relates to the homophily in the contributor network, i.e., whether contributors tend to collaborate with people sharing similar contribution history or people with different backgrounds. This question may be explored by characterizing contributors with embeddings built from the contributor collaboration network (see Section 2.1) and computing the distance between the embeddings of adjacent contributors.

9 Conclusions

This research discloses the heterogeneity between the Python and Java communities in social coding with repository embeddings from a newly constructed dataset. By building up a representation learning pipeline, the paper first illustrates how to generate high-quality

repository embeddings with content and contextual data, including import libraries, readme text, and co-contributor networks. The quality of embeddings is verified by comparing model performance on specialized prediction tasks, and the results suggest that models derived from Word2Vec perform the best in representing the repositories for both languages. This refutes the prevalence of transformer and topic models in prior textual analyses.

With the best embeddings, my research then demonstrates the language-specific heterogeneity through the analysis of socio-functional mapping, programmers' contributing diversity, and prediction of repository popularity. For socio-functional mapping, methods such as CCA and neighbor distance comparison are used, leading to the discovery that the Python community presents a "many-to-many" mode, indicating more competition among functionally similar repositories and more diversity among the functions of socially linked repositories. In contrast, the Java community is closer to a "one-to-one" model in which the contributor groups are more specialized in fewer functional areas. The insights from such discrepancy for repository recommendations are also argued. Regarding programmers' contributing diversity, results from the correlation tests illustrate that a higher contributing diversity signifies a larger probability to contribute to more popular repositories in Python, but not in Java. This implies that such diversity is more valued by the Python community. Finally, regarding repository popularity prediction, we see that embedding-based features significantly improve the model performance, and functional embedding is more important for predicting Python repositories' popularity, while social embedding assists more in predicting Java repositories.

The reasons for the language-related divergence are discussed by reviewing the difference in design philosophy, extensibility, grammar, and application scenarios (markets) between the two languages. Python's flexibility and concentration in the market within coders leads to its highlight on functionality, while Java's thoroughness and focus on the end-user market makes it dominated by authoritative organizations. The findings provide evidence for the existence of

linguistic relativity in programming languages, uncovering the latent connections between language properties and user behaviors in the GitHub ecosystem.

Acknowledgments

This research is sponsored by the Alfred P. Sloan Foundation, and I appreciate the contributions made by Songrun He, Luxin Tian, and Yilun Xu in building the representation learning pipeline. I am also grateful for the instructions provided by cohorts in the Knowledge Lab, including Jeremiah Milbauer, Reid McIlroy-Young, and Prof. James Evans.

Reference

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1–29.

Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, *46*(3), 175.

Alves, G. B., Brandão, M. A., Santana, D. M., da Silva, A. P. C., & Moro, M. M. (2016). The Strength of Social Coding Collaboration on GitHub. *SBBD*, 247–252.

Arnold, K., & Gosling, J. (2000). *The java programming language* (3rd ed.). Addison Wesley.

Batista, N. A., Brandão, M. A., Alves, G. B., da Silva, A. P. C., & Moro, M. M. (2017). Collaboration strength metrics and analyses on GitHub. *Proceedings of the International Conference on Web Intelligence - WI '17*. the International Conference, Leipzig, Germany. https://doi.org/10.1145/3106426.3106480

Batista, N. A., Sousa, G. A., Brandão, M. A., da Silva, A. P. C., & Moro, M. M. (2018). Tie strength metrics to rank pairs of developers from github. *Journal of Information and Data Management*, *9*(1), 69–69.

Beltagy, I., Lo, K., & Cohan, A. (2019). SciBERT: A pretrained language model for scientific

text. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China. https://doi.org/10.18653/v1/d19-1371

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: a review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(8), 1798–1828.

Beyer, A., Kauermann, G., & Schütze, H. (2020). Embedding space correlation as a measure of domain similarity. *Proceedings of the 12th Language Resources and Evaluation Conference*, 2431–2439.

Bhattacharya, P., Neamtiu, I., & Faloutsos, M. (2014, September). Determining developers' expertise and role: A graph hierarchy-based approach. *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada. https://doi.org/10.1109/icsme.2014.23

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *The Journal of Machine Learning Research*, *3*, 993–1022.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*, *2008*(10), P10008.

Boggust, A., Carter, B., & Satyanarayan, A. (2019). Embedding Comparator: Visualizing differences in global structure and local neighborhoods via small multiples. In *arXiv [cs.HC]*. arXiv. http://arxiv.org/abs/1912.04853

Borges, H., Hora, A., & Valente, M. T. (2016, September 9). Predicting the popularity of

GitHub repositories. *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2016: The 12th International Conference on Predictive Models and Data Analytics in Software Engineering, Ciudad Real Spain. https://doi.org/10.1145/2972958.2972966

Borges, H., & Tulio Valente, M. (2018). What's in a GitHub star? Understanding repository starring practices in a social coding platform. *The Journal of Systems and Software*, *146*, 112–129.

Bradley, A. P. (1997). The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, *30*(7), 1145–1159.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *The Journal of Artificial Intelligence Research*, *16*, 321–357.

Choudhary, S., Bogart, C., Rose, C., & Herbsleb, J. (2020, February). Using productive collaboration bursts to analyze open source collaboration effectiveness. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London.

Constantinou, E., & Kapitsaki, G. M. (2016). Developers Expertise and Roles on Software Technologies. *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, New Zealand. https://doi.org/10.1109/apsec.2016.061

Costabello, L., Pai, S., Van, C. L., McGrath, R., McCarthy, N., & Tabacof, P. (2019). *AmpliGraph: a Library for Representation Learning on Knowledge Graphs*. https://doi.org/10.5281/zenodo.2595043

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub. *Proceedings*

of the ACM 2012 Conference on Computer Supported Cooperative Work - CSCW '12. the ACM 2012 conference, Seattle, Washington, USA. https://doi.org/10.1145/2145204.2145396

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional Transformers for language understanding. In *arXiv [cs.CL]*. arXiv. http://arxiv.org/abs/1810.04805

Dey, T., Karnauch, A., & Mockus, A. (2021, May). Replication package for representation of developer expertise in open source software. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, ES. https://doi.org/10.1109/icse-companion52605.2021.00109

Ducheneaut, N. (2005). Socialization in an open source software community: A Socio-technical analysis. *Computer Supported Cooperative Work: CSCW: An International Journal*, *14*(4), 323–368.

El Mezouar, M., Zhang, F., & Zou, Y. (2019). An empirical study on the teams structures in social coding using GitHub projects. *Empirical Software Engineer*, *24*(6), 3790–3823.

Fang, H., Klug, D., Lamba, H., Herbsleb, J., & Vasilescu, B. (2020, June 29). Need for tweet. *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20: 17th International Conference on Mining Software Repositories, Seoul Republic of Korea. https://doi.org/10.1145/3379597.3387466

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*. Findings of the Association for Computational Linguistics: EMNLP 2020, Online.

https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fu, E., Zhuang, Y., Zhang, J., Zhang, J., & Chen, Y. (2021, May 5). Understanding the user interactions on GitHub: A social network perspective. *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Dalian, China. https://doi.org/10.1109/cscwd49262.2021.9437744

Goyal, P., Huang, D., Goswami, A., Chhetri, S. R., Canedo, A., & Ferrara, E. (2019). Benchmarks for Graph Embedding Evaluation. In *arXiv [cs.SI]*. arXiv. http://arxiv.org/abs/1908.06543

Grover, A., & Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. *International Conference on Knowledge Discovery & Data Mining. International Conference on Knowledge Discovery & Data Mining*, *2016*, 855–864.

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Representation learning on graphs: Methods and applications. In *arXiv [cs.SI]*. arXiv. http://arxiv.org/abs/1709.05584

Han, J., Deng, S., Xia, X., Wang, D., & Yin, J. (2019, July). Characterization and prediction of popular projects on GitHub. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA. https://doi.org/10.1109/compsac.2019.00013

Hippel, E. von, & Krogh, G. von. (2003). Open source software and the "private-collective" innovation model: Issues for organization science. *Organization Science*, *14*(2), 209–223.

Hu, Y., Wang, S., Ren, Y., & Choo, K.-K. R. (2018). User influence analysis for Github developer social networks. *Expert Systems with Applications*, *108*, 108–118.

Jiang, J.-Y., Cheng, P.-J., & Wang, W. (2017, August 7). Open source repository

recommendation in social coding. *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '17: The 40th International ACM SIGIR conference on research and development in Information Retrieval, Shinjuku Tokyo Japan. https://doi.org/10.1145/3077136.3080753

Jolliffe, I. T. (2013). *Principal component analysis* (1986th ed.) [PDF]. Springer. https://doi.org/10.1007/978-1-4757-1904-8

Kay, P., & Kempton, W. (1984). What is the Sapir-whorf hypothesis? *American Anthropologist*, *86*(1), 65–79.

Kuhlman, D. (2011). *A python book*. Platypus Global Media.

Lakhani, K. R., & von Hippel, E. (2004). How open source software works: "free" user-to-user assistance. In *Produktentwicklung mit virtuellen Communities* (pp. 303–339). Gabler Verlag.

Le, Q. V., & Mikolov, T. (2014). Distributed representations of sentences and documents. In *arXiv [cs.CL]*. arXiv. http://arxiv.org/abs/1405.4053

Li, Y., Vanhaverbeke, W., & Schoenmakers, W. (2008). Exploration and exploitation in innovation: Reframing the interpretation. *Creativity and Innovation Management*, *17*(2), 107–126.

Li, Z., Jiang, J.-Y., Sun, Y., & Wang, W. (2019). Personalized question routing via heterogeneous network embedding. *Proceedings of the ... AAAI Conference on Artificial Intelligence. AAAI Conference on Artificial Intelligence*, *33*, 192–199.

Lima, A., Rossi, L., & Musolesi, M. (2014). Coding together at scale: GitHub as a collaborative social network. In *arXiv [cs.SI]*. arXiv. http://arxiv.org/abs/1407.2535

Liu, C., Yang, D., Zhang, X., Ray, B., & Rahman, M. M. (2018). Recommending GitHub projects for developer onboarding. *IEEE Access: Practical Innovations, Open Solutions*, *6*, 52082–52094.

Ma, Y., Li, H., Hu, J., Xie, R., & Chen, Y. (2017, September 22). Mining the network of the programmers. *Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing*. ChineseCSCW '17: Chinese Conference on Computer Supported Cooperative Work and Social Computing, Chongqing China. https://doi.org/10.1145/3127404.3127431

March, J. G. (1991). Exploration and Exploitation in Organizational Learning. *Organization Science*, *2*(1), 71–87.

Matek, T., & Zebec, S. T. (2016). GitHub open source project recommendation system. In *arXiv [cs.SI]*. arXiv. http://arxiv.org/abs/1602.02594

McMillan, C., Grechanik, M., & Poshyvanyk, D. (2012, June). Detecting similar software applications. *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE 2012), Zurich. https://doi.org/10.1109/icse.2012.6227178

McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., & Xie, Q. (2012). Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, *38*(5), 1069–1087.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In *arXiv [cs.CL]*. arXiv. http://arxiv.org/abs/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. In *arXiv [cs.CL]*. arXiv. http://arxiv.org/abs/1310.4546

Montandon, J. E., Lourdes Silva, L., & Valente, M. T. (2019, May). Identifying experts in software libraries and frameworks among GitHub users. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC,

Canada. https://doi.org/10.1109/msr.2019.00054

Moradi-Jamei, B., Kramer, B. L., Calderon, J. B. S., & Korkmaz, G. (2021). Community formation and detection on GitHub collaboration networks. In *arXiv [cs.SI]*. arXiv. http://arxiv.org/abs/2109.11587

Mozafari, M., Farahbakhsh, R., & Crespi, N. (2020). Hate speech detection and racial bias mitigation in social media based on BERT model. *PloS One*, *15*(8), e0237861.

Newman, D., Lau, J. H., Grieser, K., & Baldwin, T. (2010). Automatic evaluation of topic coherence. *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 100–108.

O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., & Others. (2019). *KerasTuner*. https://github.com/keras-team/keras-tuner

Pennacchiotti, M., & Gurumurthy, S. (2011). Investigating topic models for social media user recommendation. *Proceedings of the 20th International Conference Companion on World Wide Web - WWW '11*. the 20th international conference companion, Hyderabad, India. https://doi.org/10.1145/1963192.1963244

Ramos, J., & Others. (2003). Using tf-idf to determine word relevance in document queries. *Proceedings of the First Instructional Conference on Machine Learning*, *242*, 29–48.

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China. https://doi.org/10.18653/v1/d19-1410

Saadat, S., Newton, O. B., Sukthankar, G., & Fiore, S. M. (2020, December). Analyzing the

productivity of GitHub teams based on formation phase activity. *2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), Melbourne, Australia. https://doi.org/10.1109/wiiat50758.2020.00027

Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., & Ell, J. (2014). Understanding "watchers" on GitHub. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. the 11th Working Conference, Hyderabad, India. https://doi.org/10.1145/2597073.2597114

Shi, Y., Zheng, Y., Guo, K., Zhu, L., & Qu, Y. (2018, November). Intrinsic or extrinsic evaluation: An overview of word embedding evaluation. *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. 2018 IEEE International Conference on Data Mining Workshops (ICDMW), Singapore, Singapore. https://doi.org/10.1109/icdmw.2018.00179

Theeten, B., Vandeputte, F., & Van Cutsem, T. (2019, May). Import2vec: Learning Embeddings for Software Libraries. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada. https://doi.org/10.1109/msr.2019.00014

Thung, F., Bissyande, T. F., Lo, D., & Jiang, L. (2013, March). Network structure of social coding in GitHub. *2013 17th European Conference on Software Maintenance and Reengineering*. 2013 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), Genova. https://doi.org/10.1109/csmr.2013.41

Uzzi, B., Mukherjee, S., Stringer, M., & Jones, B. (2013). Atypical combinations and scientific impact. *Science (New York, N.Y.), 342*(6157), 468–472.

Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, *9*(11).

Vasilescu, B., Filkov, V., & Serebrenik, A. (2013, September). StackOverflow and GitHub: Associations between software development and crowdsourced knowledge. *2013 International Conference on Social Computing*. 2013 International Conference on Social Computing (SocialCom), Alexandria, VA, USA. https://doi.org/10.1109/socialcom.2013.35

Wang, T., Lu, K., Chow, K. P., & Zhu, Q. (2020). COVID-19 sensing: Negative sentiment analysis on social media in China via BERT model. *IEEE Access: Practical Innovations, Open Solutions*, *8*, 138162–138169.

Weber, S., & Luo, J. (2014). What makes an open source code popular on git hub? *2014 IEEE International Conference on Data Mining Workshop*, 851–855.

Wegelin, J. A. (2000). *A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case*.

Yu, Y., Yin, G., Wang, H., & Wang, T. (2014). Exploring the patterns of social behavior in GitHub. *Proceedings of the 1st International Workshop on Crowd-Based Software Development Methods and Technologies - CrowdSoft 2014*. the 1st International Workshop, Hong Kong, China. https://doi.org/10.1145/2666539.2666571

Zhang, J., Dong, Y., Wang, Y., Tang, J., & Ding, M. (2019, August). ProNE: Fast and scalable network representation learning. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Twenty-Eighth International Joint Conference on Artificial Intelligence {IJCAI-19}, Macao, China. https://doi.org/10.24963/ijcai.2019/594

Zhang, Yiming, Fan, Y., Ye, Y., Zhao, L., & Shi, C. (2019, November 3). Key player identification in underground forums over attributed heterogeneous information

network embedding framework. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. CIKM '19: The 28th ACM International Conference on Information and Knowledge Management, Beijing China. https://doi.org/10.1145/3357384.3357876

Zhang, Yun, Lo, D., Kochhar, P. S., Xia, X., Li, Q., & Sun, J. (2017, February). Detecting similar repositories on GitHub. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria. https://doi.org/10.1109/saner.2017.7884605

Zhou, S., Vasilescu, B., & Kästner, C. (2019, August 12). What the fork: a study of inefficient and efficient forking practices in social coding. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '19: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn Estonia. https://doi.org/10.1145/3338906.3338918

Zöller, N., Morgan, J. H., & Schröder, T. (2020). A topology of groups: What GitHub can tell us about online collaboration. *Technological Forecasting and Social Change*, *161*(120291), 120291.