2016 IEEE 40th Annual Computer Software and Applications Conference

GRETA: Graph-Based Tag Assignment for GitHub Repositories

Xuyang Cai, Jiangang Zhu, Beijun Shen†, Yuting Chen†
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University, Shanghai, China
Email: bakercxy@sjtu.edu.cn, †bjshen@sjtu.edu.cn, †chenyt@cs.sjtu.edu.cn

***Abstract*—GitHub is a well-known software community where a large number of software repositories are hosted. Since large amounts of documents and code in GitHub repositories are in a mess, users cannot search or understand them efficiently. One solution is to employ a tag system, which annotates each repository with several tags. Thus, the GitHub repositories can be more efficiently accessed and understood.**

However, GitHub does not provide any automated tools of tagging repositories. In order to tackle this problem, we propose *GRETA*, a novel graph-based approach to assigning tags for repositories on GitHub. The core insight of GRETA is (1) to construct an *Entity-Tag Graph (ETG)* for GitHub using the domain knowledge from StackOverflow, and (2) to assign tags for repositories by taking a random walk algorithm. We have implemented GRETA and also developed a repository search engine for GitHub using the tag assignment results of GRETA.

We have evaluated GRETA against several baseline methods to investigate its effectiveness of tagging GitHub repositories. The results show GRETA achieves up to 35% of *F-Measure*, outperforming the baseline methods. Besides, the GRETA-based search engine gains a higher *NDCG* value than the search engine provided by GitHub, indicating that it significantly enhances the search ability on GitHub with the tagged repositories.

***Keywords*—Tag assignment, GitHub, cross-community.**

I. Introduction

GitHub is one of the most popular software communities for software developers. It facilitates software engineers to access and manage source code everywhere, control software versions conveniently, and collaborate with each other easily. Up to 2015, GitHub has over 9 million users and hosts over 21.1 million software repositories, becoming the largest software community in the world [1]. Figure 1 shows an example of a GitHub repository with its textual descriptions.

However, large amounts of documents and code existing in repositories are usually in a mess. Thus, users cannot efficiently search and access many repositories. In particular, developers must spend much time on understanding a repository when it is accompanied with a mess of information. They have to read textual documents and sometimes even code to understand a repository and decide whether the repository meets their requirements. Besides, the existing search engine only retrieves repositories based on their explicit texts, rather than their actual functionalities and characteristics.

One solution is to employ a tag system such that the GitHub repositories can be more efficiently searched and understood. A tag system annotates each repository with one or more tags on GitHub. The benefits of tagging software repositories are in two folds. First, tag is a kind of metadata used widely in websites and communities. It is also abstract and is easier to understand than the concrete object (as well as its documents). For example, "*css*" and "*font*" can be used as tags for the text "*how do float font-size values on CSS render among browsers*". Second, a few latent relations among the software repositories can be uncovered by tags. These relations facilitate engineers to search for related software systems or trace software evolving histories. However, GitHub does not provide any automated tools of tagging repositories.

Up to now, the related work of tagging objects is mainly divided into two categories: tag assignment and tag extraction. Tag assignment relies on the human intuition to assign tags to objects. The method needs a predefined dictionary, which may be indeterminate for tagging a practical system [2]–[5]. Besides, many existing tag assignment methods omit the shared knowledge among objects; each object is tagged independently from the others. Tag extraction advocates the idea of extracting keyphrases as tags directly from object documents [6]–[15]. However, since a repository description on GitHub is usually short, and its readme document sometimes only concerns the usage of the project (such as how to install the system and how to setup configurations), the repository documents may be inadequate for tag extraction. Thus, it weakens the representativeness of the tags used for tagging a GitHub repository.

We observe that many software developers who have contributed to the GitHub repositories are also involved in other communities for software development. For example, StackOverflow is one of the most popular Q&A communities. It allows software developers to raise questions, share their development experiences, and search for solutions to problems. More importantly, StackOverflow allows developers to tag questions. Figure 2 shows a question posted in StackOverflow. The question is marked with some tags assigned by the questioner. So far, there are more than 45,000 distinct tags and millions of questions in the community. These labeled tags in StackOverflow are mainly concerned with four aspects: languages, frameworks, API usages and technical supports. As a large number of software developers who contribute to repositories on GitHub also raise and answer questions on StackOverflow, we believe these tags can represent some

† Corresponding Author

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.124

IEEE computer society

domain knowledge of software development, and they can also be used to generate a tag dictionary and be assigned to the repositories on GitHub. But how to bridge the gap between GitHub and StackOverflow is still a challenging problem.

Fig. 1: Descriptions of a repository on GitHub.

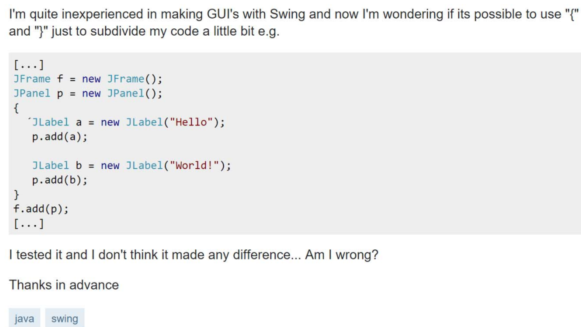


Fig. 2: A question posted on StackOverflow.

In order to tackle this problem, we propose *GRETA*, a novel graph-based approach to assigning tags for repositories on GitHub. The core insight of GRETA is (1) to construct an *Entity-Tag Graph (ETG)* for GitHub using the domain knowledge (*e.g.*, repositories, questions and tags) from cross-communities (*e.g.*, GitHub, StackOverflow), and (2) to propagate tags in ETG by taking an iterative random walk with restart algorithm [16]. An ETG is mainly composed of a set of nodes representing the repositories on GitHub. It also includes domain knowledge collected from StackOverflow. During the propagation, domain knowledge is shared among nodes and tags are assigned to the GitHub repositories.

This paper makes the following contributions:

1) *Approach*. GRETA is a novel, graph-based approach to tag assignment for repositories on GitHub, which allows tags to be assigned by some graph algorithms. GRETA is also a cross-community approach, which utilizes the domain knowledge from StackOverflow for facilitating tag assignment. For this purpose, we design an *Entity-Tag Graph (ETG)* for establishing relations between GitHub repositories and StackOverflow tags, and then use a random walk algorithm to tag software repositories.
2) *Implementation*. We have implemented GRETA and also developed a repository search engine for GitHub using the tag assignment results of GRETA, which is the summarization of characteristics for better software retrieval.
3) *Evaluation*. We have evaluated GRETA against several baseline methods to investigate its effectiveness of tagging GitHub repositories. The results show GRETA outperforms the baseline methods in that it achieves up to 35% of *F-Measure*. Besides, the GRETA-based search engine gains a higher NDCG value than the existing search engine in GitHub, indicating that it significantly enhances the search ability on GitHub with the tagged repositories.

The rest of this paper is organized as follows: We introduce the related work in Section II. We next present the GRETA approach and the GRETA-based engine in Sections III and IV, respectively. Section V presents an evaluation of the GRETA approach and the engine. Section VI concludes.

II. Related Work

Tag extraction. Tag extraction mainly concerns about how to select important and topical phrases from a textual description [17]. Techniques for tag extraction can be supervised or unsupervised.

A series of supervised tag extraction techniques have been proposed [6]–[8]. They mainly take tag extraction as a binary classification task, which trains a binary classifier to determine whether a set of candidate phrases are suitable for acting as tags. Different classification algorithms, such as Naïve Bayes [7] [8], decision tree [6], and multi-layer perceptron [9] [10], have been applied.

However, supervised tag extraction integrating classification formulation has some drawbacks. On the one hand, the training data may be unbalanced. Since only several tags can be selected for a document, the vast majority of candidate phrases are negative examples. Moreover, when multiple candidate phrases for an object are classified by the binary classifier as positive, it is not easy to distinguish which phrases are more representative than the others. These problems lead to a non-ideal performance in tag extraction with traditional classification. Thus, Jiang *et al.* [11] propose a learning-to-rank approach to perform this task. Compared with traditional classification approaches, the ranking-based approach significantly improves the extraction performance. On the other hand, it is time-consuming to use the supervised methods, because they need a training set with manual annotations.

Among unsupervised approaches, graph-based ranking approaches are most widely used. The basic idea of this approach is to build a graph from the input document and rank its nodes using a graph-based ranking method. The most well-known graph-based method is TextRank [12]: It treats all the phrases as nodes and those phrases in a window with a fixed size are linked by unweighted edges. Then, a traditional PageRank[1] algorithm is applied on this graph. Those phrases with high PageRank values are selected as tags. Rather than focusing on a single document for tag

[1] https://en.wikipedia.org/wiki/PageRank

extraction, Wan *et al.* [13] leverages a small number of nearest neighboring documents for tag extraction. Another unsupervised approach for tag extraction is clustering-based approach [14] [15]. Liu *et al.* [14] leverage several clustering techniques to find some exemplar terms, which semantically cover the object document. They then extract keyphrases from the exemplar terms. Grineva *et al.* [15] model a graph with semantic relationships between documents, and apply graph community detection techniques to partition the graph into thematically cohesive groups of terms. They then use a criterion function to extract tags.

Tag assignment. Tag assignment aims to assign some tags for each repository, even though these tags may not appear in its descriptions. So tag assignment needs to construct a predefined dictionary first. Two widely used dictionaries are the hierarchical concept tree provided by Open Directory Project (ODP)[2] and Wikipedia[3]. If we take the phrases in the dictionary as tags, the tag assignment problem can be casted as a multi-class multi-label classification problem [2] [3]. When the size of the dictionary is large and that of the training data for each tag is small, some semi-supervised learning algorithms are adapted. They can use a little labeled data to build the classification model. Moreover, traditional classification algorithms such as support vector machine (SVM) is hard to tackle with such a large number of tags. Xue *et al.* [4] propose a search heuristic to reduce the number of tags by taking the hierarchical relations between them. Madani *et al.* [5] propose Feature Focus Algorithm (FFA). It is convenient to constrain each feature to connect to a small subset of the classes in the index, which is a mapping from features to classes. They use an index-learning approach that is highly advantageous for space and time efficiency.

Tagging in Software Engineering. Tagging objects in software engineering sites has been studied extensively [18]–[20]. Wang *et al.* propose EnTagRec, a tag recommendation system which combines the Bayesian Inference, Frequentist Inference and spreading activation technique [21] to recommend tags for StackOverflow [18]. Xia *et al.* propose a tag recommendation for software information sites like Freecode and StackOverflow based on multi-label classification [19].

This paper presents an approach to assigning tags for GitHub repositories. Compared with existing tag assignment approaches, GRETA employs the cross-community knowledge when tagging repositories, which saves human efforts on defining a tag dictionary. GRETA also allows tags to be propagated and prioritized for tag assignment.

III. Approach

A. Problem Formulation

In general, the task in this paper is to assign tags for unlabeled repositories on GitHub. The task can also be formulated

[2]http://www.dmoz.org

[3]https://www.wikipedia.org/

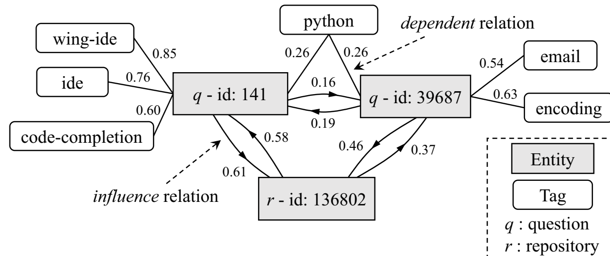


Fig. 3: An example of ETG.

as follows. Let q be a question and Q be a collection of questions. Let r be a repository and R be a collection of repositories. Let U be a collection of users, who participate in one or more repositories and/or questions. Let the set of tags used for questions in Q be T. Since there are plenty of textual materials in both communities, we combine *question* and *answers* in StackOverflow to denote a question document d_q. While in GitHub, we combine a repository's *description*, *language* and *readme* to denote a repository document d_r. The document collection is denoted as D and the word vocabulary V, and the words in d_r and d_q are taken from V.

Thus, a question is defined as a 3-tuple $q = \{U_q, T_q, d_q\}$: $U_q \subseteq U$ is a set of questioner and responders of q; $T_q \subseteq T$ is a set of tags annotated to q. An unlabeled repository is defined as a 2-tuple $r = \{U_r, d_r\}$: $U_r \subseteq U$ is the set of developers in repository r. Given an input set $I = \{Q, R, T, U\}$, we output a labeled repository set R', in which elements are represented as a 3-tuple $r' = \{U_r, d_r, T_r\}$, where $T_r \subseteq T$.

ETG definition. An ETG is defined as a weighted graph. As Figure 3 shows, an element of an ETG can be:

1) A vertex: There are two types of vertices in ETG. One is entity, including both question and repository. The other is tag, which represents a distinct tag in T.

2) An edge: The edges in ETG represent two kinds of relations. One is a directed edge between entities. It shows the knowledge-sharing probability from an entity to the other, denoted as *influence* relation. The other is an undirected edge between tag and entity. It shows the tag assignment probability on entities, denoted as *dependent* relation.

Data set. In our study, the data we use is all before July, 2015. We first select some repositories with many stargazers and forks in GitHub, and some questions with huge amount of votes or answers in StackOverflow. They are collected in R and Q respectively. We then collect the users who participate in at least one repositories in R and/or questions in Q and also identify the same users in the user set from both communities.

In order to ensure the accuracy of the similarity calculated between entities, we only use the simple heuristic rule to make identification: if a user profile url in StackOverflow and that in GitHub link to the same site, the two users are taken as the same person [22]. All these identified same users are collected

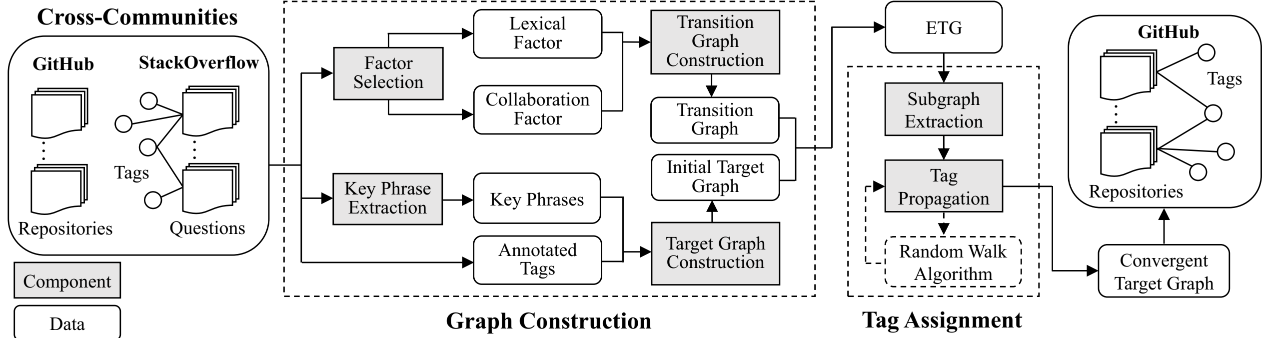


Fig. 4: Workflow of GRETA.

in U. Thus, we totally collected 217,089 questions and 416,666 repositories, with 35,931 users and 38,205 annotated tags.

Unlike some popular social communities such as Twitter [23] and Weibo [24], users in StackOverflow or GitHub sometimes provide code fragments when raising questions or introducing their projects. However, as there is no such an engine that is strong enough for abstracting any program code to their functionalities, code usually cannot be processed as well as textual descriptions. Thus we remove all the code fragments from documents. It is also necessary to remove the stopwords (*e.g.*, the, for, of) and special characters (*e.g.*, :-(, @,&, ...).

B. Approach Overview

We propose a novel, graph-based approach to assigning tags for repositories by using cross-community resources. As Figure 4 shows, the workflow of our approach is mainly divided into two phases.

1) Graph Construction: We collect a set of entities including questions on StackOverflow and repositories on GitHub, in which we identify a few same users who participate in the both communities. After that, we build an ETG from two views. One view is a *transition graph*, which explains the influence relation between entities. The other view is a *target graph*, which indicates the dependent relation between tags and entities.

2) Tag Assignment: We extract a subgraph from the original ETG and apply an iterative random walk algorithm to propagate tags to repositories. A random walk on a graph can propagate tags along the edges and provide knowledge sharing between indirectly connected nodes [25]. We finally get a convergent target matrix, which provides the tag probability distribution for entities. We use it to select appropriate tags for each repository.

C. Graph Construction

Transition graph construction. We now illustrate the details of constructing transition graph G_T. All vertices in G_T are entities, including questions or repositories. The edges are bi-directional between entities, representing their influence relation. We use matrix $M_{T_{N\times N}}$ to represent G_T, where $N = |R| + |Q|$. Each row in M_T represents an entity e_i. The top $|R|$ rows in upper-semi transition matrix $M_T^{\top}$ correspond to repositories in R, and the rest in lower-semi transition matrix $M_T^{\perp}$ correspond to questions in Q. Correspondingly, columns in M_T have the same definition as rows. Table I illustrates the matrix representation of the transition graph.

TABLE I: Transition matrix $M_{T_{N\times N}}$.

	Repositories			**Questions**		
	e_1	...	$e_{\|R\|}$	$e_{\|R+1\|}$	...	e_N
e_1	y_{11}	...	$y_{1\|R\|}$	$y_{1\|R+1\|}$	...	y_{1N}
...	...	...	...	...	...	...
$e_{\|R\|}$	$y_{\|R\|1}$	...	$y_{\|R\|\|R\|}$	$y_{\|R\|\|R+1\|}$	...	$y_{\|R\|N}$
$e_{\|R+1\|}$	$y_{\|R+1\|1}$	...	$y_{\|R+1\|\|R\|}$	$y_{\|R+1\|\|R+1\|}$	...	$y_{\|R+1\|N}$
...	...	...	...	...	...	...
e_N	y_{N1}	...	$y_{N\|R\|}$	$y_{N\|R+1\|}$	...	y_{NN}

The element in M_T shows the weight of the edge from entity e_j to e_i in G_T, which is denoted as $y_{ij}|_{i,j\leq N}$. It represents the probability whether the entity e_j would share its tags with e_i. We mainly concern about two factors including *lexical* and *user collaboration* to compute y_{ij}. The details of these factors are as follows:

1) Lexical factor: Lexical factor is the measurement of the similarity between two entities in terms of their documents. Let the document of entity e_i be denoted as d_i. We use *TF-IDF* to compute the word weight in each document. The weight of word w_j in document d_i is denoted as ε_{ij}:

$$\varepsilon_{ij} = \frac{n_{ij}}{\sum_{k=1}^{N} n_{ik}} \cdot \log \frac{N}{|\{k : w_j \in d_k\}| + 1} \tag{1}$$

where n_{ij} denotes the occurrence number of w_j in d_i.

We then construct Vector Space Model(VSM) for each document d_i, denoted as $\vec{V_i} = \{\varepsilon_{i1}, \varepsilon_{i2}, ..., \varepsilon_{i|d_i|}\}$. The lexical similarity is computed as the cosine value with respect to their vectors. We also make an assumption that the entity with a longer document length $|d_i|$ is more valuable and more likely to propagate tags to its neighbors than the other nodes. Thus, the lexical similarity of two entities e_i and e_j is computed as:

$$lsim_{(e_i,e_j)} = \frac{\vec{V_i} \cdot \vec{V_j}}{\left\|\vec{V_i}\right\| \left\|\vec{V_j}\right\|} \times \log\left(\frac{|d_j|}{|d_i|} + 1\right) \tag{2}$$

2) User collaboration factor: We observe that the GitHub users involved in some technical fields are likely to search, raise or answer the corresponding questions in StackOverflow. Thus, we assume that the more same users two entities are linked with, the higher probability they would have the same knowledge.

Let U_{e_i} denote the set of users for e_i. We use *Jaccard Similarity* to compute the user collaboration similarity of two entities e_i and e_j:

$$usim_{(e_i,e_j)} = \frac{\left|U_{e_i} \bigcap U_{e_j}\right|}{\left|U_{e_i} \bigcup U_{e_j}\right|} \tag{3}$$

3) Multi-factors fusion: We compute y_{ij} with the combination of the lexical and the user collaboration factors:

$$y_{ij} = \begin{cases} (1-\alpha) \cdot \log\left(lsim_{(e_i,e_j)} + 1\right) \\ +\, \alpha \cdot \log\left(usim_{(e_i,e_j)} + 1\right),\ i \neq j \\ \\ 0,\ i = j \end{cases} \tag{4}$$

where α is the weighting parameter to balance two factors. We evaluate the performance through setting different α values in our experiments. To prevent self propagation, we set $y_{ij} = 0$ on the diagonal of matrix M_T.

4) Matrix postprocessing: Some high frequent tags can dominate the result during the tag propagation phase. For example, "*javascript*" is one of the most popular tags. That many questions mutually propagate this tag may lead to the result that all entities are finally tagged by the tag. Thus, we first sparsify M_T through respectively retaining the top-γ largest values of some question and repository entities in each row. We then normalize M_T to balance the weights for entities.

$$y'_{ij} = \frac{y_{ij}}{\sum_{k=1}^{N} y_{ik}} \tag{5}$$

Target graph construction. We next describe how to construct a target graph G_O. Both of entities and tags are treated as vertices. Each entity and tag in G_O are connected with an undirected edge, which indicates the dependent relation between them. We also use matrix $M_{O_{N\times|T|}}$ to represent G_O. Each row in M_O has the same definition as M_T, while each column denotes a distinct tag. The weight of element in M_O is denoted as $z_{ij}|_{i\leq N, j\leq|T|}$, which represents the possibility that tag t_j needs to be assigned to entity e_i. Table II illustrates the matrix representation of the target graph.

TABLE II: Target matrix $M_{O_{N\times|T|}}$.

	Tags					
	t_1	...	t_i	t_j	...	$t_{\|T\|}$
e_1	z_{11}	...	z_{1i}	z_{1j}	...	$z_{1\|T\|}$
...	...	...	...	...	...	...
$e_{\|R\|}$	$z_{\|R\|1}$	...	$z_{\|R\|i}$	$z_{\|R\|j}$	...	$z_{\|R\|\|T\|}$
$e_{\|R+1\|}$	$z_{\|R+1\|1}$	...	$z_{\|R+1\|i}$	$z_{\|R+1\|j}$	...	$z_{\|R+1\|\|T\|}$
...	...	...	...	...	...	...
e_N	z_{N1}	...	z_{Ni}	z_{Nj}	...	$z_{N\|T\|}$

We use two methods to initialize the value z_{ij} for target matrix M_O in terms of different entities.

1) Question entities initialization: Considering that questions in StackOverflow have been tagged by users and the questions we selected are in high quality, we use these user-labeled tags to initialize the lower-semi target matrix $M_O^{\perp}$ from the $|R+1|_{th}$ to N_{th} row. For a question e_i, the weight of each tag that occurs in T_{e_i} is initialized to the corresponding column in $M_O^{\perp}$.

Observing that a majority of tags in T occurs with a low frequency, known as a long tail distribution, we provide a penalty term to reduce the propagation efficiency of some frequently-used tags. The formula to compute the weight of element in lower-semi target matrix $M_O^{\perp}$ is:

$$z_{ij} = \frac{T_{e_i}.contain(t_j)}{\left|Q_{t_j}\right|^{\beta}},\ i \in \Big[|R+1|,\ N\Big] \tag{6}$$

where function $contain()$ outputs 1 when T_{e_i} contains tag t_j, or outputs 0 otherwise. $\left|Q_{t_j}\right|$ represents the occurrence time of tag t_j in the question set Q, and parameter β is the penalty degree for tags.

2) Repository entities initialization: Unlike question entities, there is no user-labeled tags existing in repositories. Considering that most of users prefer to use some representative words occurring frequently in document to abstract their questions, we take a simple keyphrase extraction method to initialize the weight of element in the upper-semi target matrix $M_O^{\top}$. For each document of a repository entity e_i, we compute the words weight using formula (1). We then sort all words in descending order by their weight and select the top-k words as a candidate tag set, which is intersected with T. The weight of each tag t_j existing in the candidate tag set is initialized to the corresponding value z_{ij}.

We also normalize the rows in M_O. The formula is close to the formula (5).

D. Tag Assignment

Subgraph extraction. Graph-based algorithms have high time-complexity in general. We must enhance their efficiency as the size of the ETG is too large. Thus, we extract a few significant entities and tags from the original ETG to construct a subgraph, on which our algorithm can be applied efficiently without sacrificing the effectiveness. For this purpose, we present a *User-Entity Graph (UEG)*, in which the entities in $R \cup Q$ and users in U are treated as nodes. The entity $e \in \{r, q\}$ and user u are connected by edges if $u \in U_i|_{i\in\{r,q\}}$. Figure 5(a) presents an example of an original UEG.

The subgraph construction starts with a set of good candidate question entities. Then UEG is expanded by adding repository entities connected to the included entities with edges. The main activities include question filter and repository expansion.

1) Question filter: We first select some good candidate question entities, each of which satisfies the minimal degree dr and document length dl. The satisfied question entity is put into the subgraph with its connected user nodes. Then we

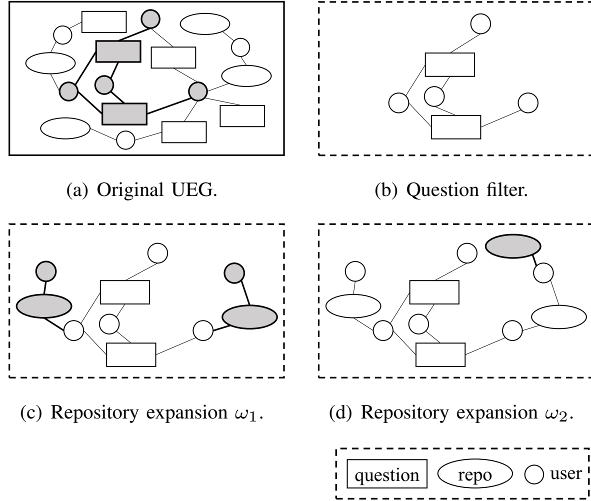


Fig. 5: Process of Subgraph Construction.

get a reduced question set and user set. Figure 5(b) shows the result after this step when $dr = 3$ (dl is not considered in this example). The result is also highlighted in Figure 5(a).

2) Repository expansion: We use the existing question entities to expand repository entities. Repository adding is an iterative process, whose iteration time is denoted as ω. In this step, repository set and user set are updated at each iteration ω_i, denoted as R^i and U^i. Let the user and repository set in subgraph be U^0 and $R^0 = \phi$ at the start, respectively. During each iteration ω_i, the repository entity r is added to the R^i if at least one of its connected users exists in U^{i-1}. Meanwhile, users existing in U_r but not in U^{i-1} are added to U^i. Figures 5(c) and 5(d) show the iterative process of expanding repositories when $\omega = 2$. The new nodes added in each iteration are highlighted.

Tag propagation. Inspired by the idea of random walk with restart, we apply an iterative algorithm to make tag propagation. A random walk with restart is a stochastic process, resulting in a probability distribution over the vertices corresponding to the likelihood those vertices are visited. This probability can be interpreted as the relatedness between nodes in the graph. The random walk starts with an initial distribution over the nodes in the graph, propagating the distribution to adjacent vertices proportionally, until convergence.

The element of M_O is interpreted to be the probability that tag is annotated to the entity, which is iteratively updated in the algorithm. We respectively denote the calculation result of M_O and its element value z_{ij} at t_{th} iteration as ${M_O}^t$ and ${z_{ij}}^t$. Thus, the value ${z_{ij}}^{t+1}$ in target matrix is computed as:

$$ {z_{ij}}^{t+1} = \sum_{e_i \in out(e_k)} y_{ik} \cdot {z_{kj}}^t \tag{7} $$

where $out(e_k)$ is a function. It returns a set of entities that the entity e_k points to.

For guaranteeing convergence, we incorporate a random restart probability θ in the initial target matrix M_O as customary. Formally, the random walk model can be modeled as:

$$ {M_O}^{t+1} = \theta \cdot M_T {M_O}^t + (1-\theta) \cdot M_O \tag{8} $$

When a random walk process converges to a stationary state, we obtain a *stationary distribution* — a tag probability distribution for entities in target matrix ${M_O}^t$. The process is converged when each element in ${M_O}^t$ satisfies the condition:

$$ \left| \frac{{z_{ij}}^t - {z_{ij}}^{t-1}}{{z_{ij}}^{t-1}} \right| \leq \delta, \ i \in [1, N], j \in [1, |T|] \tag{9} $$

Since a column corresponds to a distinct tag t_j and a row corresponds to a repository entity e_i in ${M_O^\top}^t$, we get the top-k columns with largest values ${z_{ij}}^t$ for each row to select several tags for each repository.

IV. GRETA-based Engine Implementation

A. Engine Overview

Developers must spend much time on searching a repository when they have some specific requirements. So we design an application to search repository by employing assigned tags. The search engine takes account of not only explicit repository description, but also latent software characteristics, represented by tags.

For this purpose, we use a software programming taxonomy[4] built with tags from StackOverflow [26]. Taxonomy is a kind of simplified knowledge base, which omits many complicated semantic relations between nodes. Its structure is a multi-rooted tree, in which nodes represent the concepts, and edges provide a hyponymy relation with a confidence coefficient. Each node has one parent node (except the root nodes) and one or more child nodes. For example, tag "*ide*" has a child node "*eclipse*" in the taxonomy, which means "*ide*" subsumes "*eclipse*".

There are two main algorithms in our search engine. One is for query expansion and the other for repository ranking. We next describe the two algorithms.

B. Query Expansion

We expand the query through adding some terms that are relevant with the original one. The algorithm takes a phrase or sentence S and a maximum traversal layer number L as input, and outputs an expanded vector S_q. Each term in S_q has a relevant weight v_i.

For each term in S, it is treated as an important term if it occurs in T. If so, its neighboring nodes in taxonomy such as parent, brothers and children are added into S_q, by multiplying some weakening coefficients γ_p, γ_c and a confidence coefficient $c(s_i, s_j)$ to its relevant weight. Otherwise, the term is added as a normal term by multiplying γ_o if it is not a stopword. The detailed process of Algorithm 1 is as follows.

[4]The data is available at http://datahub.io/dataset/software-zhishi-schema. There is a website for demonstration: http://swenet.me

Algorithm 1: Query Expansion

Input: $S = \{s_1, s_2, s_3, ..., s_m\}, L$
Output: $S_q = \{(s_1^*, v_1), (s_2^*, v_2), ..., (s_n^*, v_n)\}$

1: Initialize $S_q = \phi, \gamma_p = 0.5, \gamma_c = 0.8, \gamma_o = 0.9$
2: **for all** $s_i \in S$ **do**
3: **if** $s_i \in T$ **then**
4: $S_q \leftarrow$ Add $(s_i, 1)$
5: $s_p \leftarrow GetParent(s_i)$
6: **if** s_p exists **then**
7: $S_q \leftarrow$ Add $(s_p, c_{(s_i,s_p)} \cdot \gamma_p)$
8: **end if**
9: **for all** $s_j \in GetBrothers(s_i)$ **do**
10: $S_q \leftarrow$ Add $(s_j, c_{(s_i,s_p)} \cdot c_{(s_j,s_p)} \cdot \gamma_p \cdot \gamma_c)$
11: **end for**
12: $N_c \leftarrow$ Children nodes of s_i in L Layers
13: **for all** $s_j \in N_c$ **do**
14: $l =$ Layer number from s_j to s_i
15: $S_q \leftarrow$ Add $(s_j, c_{(s_i,s_j)} \cdot {\gamma_c}^l)$
16: **end for**
17: **else if** $s_i \notin Stopwords$
18: $S_q \leftarrow$ Add (s_i, γ_o)
19: **end if**
20: **end for**
21: **return** S_q

C. Repository Ranking

We use the weighted query to compute a relatedness score for each repository, and also rank them. The details are depicted in Algorithm 2.

We first build two inverted tables T_w and T_t: one is word-based and the other is tag-based. T_w includes all the words occurring in the repository documents. Each word w_i has a corresponding set G_i, which includes the repositories that contain the word. Each element in set G_i is a key-value pair such as (q_j, α_{ij}), in which α_{ij} represents the weight of w_i in q_j. Similar to T_w, T_t includes all the tags annotated by users. Each tag also has a set H_i including all repositories annotated with the tag. For an expanded weighted query input S_q, the relatedness between query and repositories are calculated from three aspects.

1) Name-based: We consider not only the *edit distance* ($dist()$) but also their *mutual coverage* ($cover()$) between repository name and query string.

2) Lexical-based: We increase the weight of the repositories that contain the word occurring in S_q.

3) Characteristic-based: We increase the weight of the repositories that are annotated by the tag occurring in S_q.

The algorithm ranks the repositories by their relatedness from the query, and outputs a top-k list of repositories Γ.

V. Evaluations

We evaluate the performance of our approach and compare it with some existing methods. We also evaluate our search engine. The results show that (1) both factors contribute to a positive effect to tag assignment, and (2) GRETA outperforms the baseline methods in that it achieves up to 35% of F-Measure. Besides, the GRETA-based search engine gains a higher NDCG value than the existing one in GitHub, indicating that the engine is more effective than the engine of GitHub in searching for repositories using software characteristics.

Algorithm 2: Repository Ranking

Input: $S_q = \{(s_1^*, v_1), ..., (s_n^*, v_n)\},\ S = \{s_1, s_2, ..., s_m\}$
$T_w = \{G_1, G_2, ..., G_m\}$, where $G_i = \{(r_1, \alpha_{i1}), ..., (r_k, \alpha_{ik})\}$
$T_t = \{H_1, H_2, ..., H_l\}$, where $H_i = \{(r_1, \beta_{i1}), ..., (r_k, \beta_{ik})\}$
Output: Repository List Γ

1: Initialize $\Gamma = \{r_1, r_2, ..., r_n\}, Scores = \{0, ..., 0\}$
2: **for all** $r_j \in R$ **do**
3: $star(r_j) = 1 + \log\left(1 + \sqrt{1 + StarGazer(r_j)}\right)$
4: $n_j \leftarrow GetRepoName(r_j).splitToTerms()$
5: **for all** $n_{jk} \in n_j$ **do**
6: **for all** $s_i^* \in S_q$ **do**
7: **if** $n_{jk}.cover(s_i^*)$ **then**
8: $coversim_{ik} = |s_i^*| \ / \ [|n_{jk}| \cdot (|Q| + 1)]$
9: **else if** $s_i^*.cover(n_{jk})$ **then**
10: $coversim_{ik} = |n_{jk}| \ / \ [|s_i^*| \cdot (|Q| + 1)]$
11: **end if**
12: **end for**
13: **end for**
14: $titleSim(j) = (\sum coversim_{ik} + 1) \ / \ [dist(n_j, S) + 1]$
15: **end for**
16: **for all** $s_i^* \in S_q$ **do**
17: **for all** $G_k \in T_w$ **do**
18: **if** $s_i^*\ equals\ G_k$ **then**
19: **for all** $r_j \in G_k$ **do**
20: $textSim(j) += v_i \cdot \alpha_{kj} * star(r_j)$
21: **end for**
22: **end if**
23: **end for**
24: **for all** $H_k \in T_t$ **do**
25: **if** $s_i^*\ equals\ H_k$ **then**
26: **for all** $r_j \in H_k$ **do**
27: $tagSim(j) += v_i \cdot \beta_{kj} \cdot \left|s_i^*.splitToTerms()\right|$
28: **end for**
29: **end if**
30: **end for**
31: **end for**
32: **for all** $r_j \in R$ **do**
33: $score = star(j) \cdot \prod [factorSim(j) + 1], \hookleftarrow$
$factorSim \in \{titleSim, textSim, tagSim\}$
34: $\Gamma \leftarrow$ Add $(r_j, score)$
35: **end for**
36: Rank Γ in descending order and select top-k
37: **return** Γ

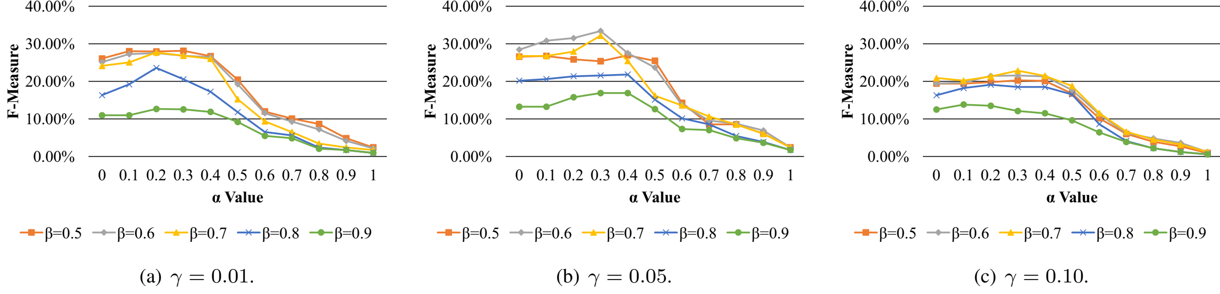
A. Experimental Settings

We first present our experimental data set and some essential parameter settings. We experimentally set dr and dl to 8 and 60, respectively, when extracting a subgraph, and iteration time ω is set to 5. Thus, we get a subgraph whose statistics are summarized in Table III.

TABLE III: Statistics of subgraph.

Description	Value
Repository Entity Number	9,163
Question Entity Number	10,724
Tag Number	4,641

In the subgraph, the average document length of all entities is 261.49. Each of them has on average 7.36 relative users. The tag frequency distribution follows a power-law like distribution. Most tags are used in few times by few users, while a small number of tags are extremely popular and have been used to annotate many questions. Each question is annotated with 3.39 tags on average. 20.6% of questions are annotated by more than 4 tags. A very small number of



(a) $\gamma = 0.01$. (b) $\gamma = 0.05$. (c) $\gamma = 0.10$.

Fig. 6: Performance with different parameter values.

questions are annotated with one tags. The remaining questions are annotated with 2∼4 tags.

We follow the standard convention by setting random restart probability $\theta = 0.85$ and convergence threshold $\delta = 0.001$ [27].

B. Compared Methods

Since there is no existing mature tool to perform the tag assignment task in cross-communities, we compare four general methods with our approach. The four baseline methods belong to two categories, *i.e.*, unsupervised methods and supervised ones.

1) Unsupervised Methods

Term Frequency-inverse Document Frequency. We compute the word weight for each repository document and select the top-ranked words.

Latent Dirichlet Allocation *(LDA)* [28]. We use repository documents to train a LDA model and select the top-3 topics for each repository. The top-ranked words contained in topics are taken as entity tags.

2) Supervised Methods

K-Nearest Neighbors *(KNN)*. We select the k-nearest neighboring questions for each repository and rank the corresponding tag weights.

Labeled Latent Dirichlet Allocation *(LLDA)* [29]. We train a LLDA model using the questions and tags, and then select the topics with highest probabilities according to each document's topic distribution.

C. Performance Measures

In our evaluation, 200 repositories were randomly selected from our assignment result. Considering that StackOverflow users may annotate several tags to one question, we use their explicit textual materials and latent meaning to manually add 1∼5 tags to these repositories. These user-labeled tags have been inspected and corrected twice by three researchers.

We use our approach and four baseline methods respectively to assign a tag set containing 5 distinct tags for each repositories. In order to reduce the negative impacts caused by the synonyms or similar tags, we normalize these user-annotated and automatic assigned tags by removing their hyphens and version numbers. For example, since the two tags named "*jdk6*" and "*jdk-1.7*" can be normalized to "*jdk*", they are taken as the same tag.

Some metrics for evaluating the accuracy in information retrievals (*IE*) can also be used for tag assignment, for example, *Precision* (Λ_P^r), *Recall* (Λ_R^r) and *F-Measure* (Λ_F^r). We use Θ_u^r to represent the set of the manual tags of repository r and Θ_g^r the set of the automatically assigned tags. The metrics are based on the concepts of True Positives **(TP)**, False Positives **(FP)** and False Negatives **(FN)**. The concepts are defined as:

- **TP** contains tags both in Θ_u^r and Θ_g^r.
- **FP** contains tags in Θ_g^r but not in Θ_u^r.
- **FN** contains tags in Θ_u^r but not in Θ_g^r.

Thus, the metric indices can be computed as:

$$\Lambda_P^r = \frac{TP}{TP+FP},\ \Lambda_R^r = \frac{TP}{TP+FN},\ \Lambda_F^r = \frac{2\cdot\Lambda_P^r\Lambda_R^r}{\Lambda_P^r+\Lambda_R^r}$$

D. Parameters Tuning

There are three parameters that need to be tuned in the evaluation, including factor weighting value α, tag penalty degree β and matrix sparsity degree γ. We set different values and compute an F-Measure to determine these parameters. We respectively set α from 0 to 1 with a step 0.1, β from 0.5 to 0.9 with a step 0.1, and γ from 0.01 to 0.10 with a step 0.05. Figure 6 shows the tag assignment performance with different parameter values. From the results, we achieve the best performance when $\alpha = 0.3$, $\beta = 0.6$ and $\gamma = 0.05$.

We then evaluate whether both of two similarity factors can effectively help assign tags. Note that the performance of the tag assignment reaches up to 33.41% when α is 0.3. In particular, the lexical factor reaches more than 25% even without the user collaboration factor. However, the prediction works better when it combines the two factors. The result indicates that both factors are helpful in tag assignment when they are combined. The result also agrees with our intuition that each factor may lead to a low performance value when used independently.

E. Tag Assignment Evaluation

We then compare the performance of GRETA with those baseline methods. We use the optimal tuned parameter values

mentioned above. We compute the F-Measure by setting the size of the assigned tag set from 1 to 5. Figure 7 shows the tag assignment performance by different methods.

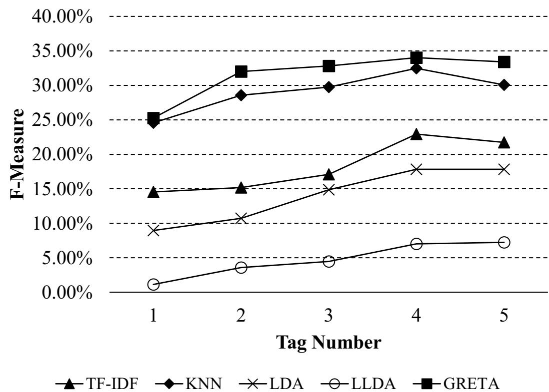


Fig. 7: Performance of different methods.

By observing the results shown in Figure 7, we find that each method shows a progressive trend of performance along with the increase of the number of assigned tags. However, when the tag number is greater than 4, most of them start to decrease.

More specifically, text similarity-based methods performs better than those topic-based ones. LDA and LLDA cannot obtain a precision of 20% even when they produce 5 tags for a repository. Unlike topic-based methods, both KNN and TF-IDF achieve an acceptable performance value during evaluation. They reach an value of more than 20% when the tag number is greater than 4. However, in most cases, GRETA achieves the best performance, which is about 3% higher than the best baseline method on average. It indicates that GRETA can produce some tags that cannot be produced using the other methods. Thus, we draw out two conclusions: (1) repositories with high similarity in the text always share similar tags; (2) users tend to use many words with the most occurrences in the document as tags.

We also use some statistical metrics to measure the results. First, we count each method's distinct tag number produced in all sampled repositories. We observe that GRETA produced 249 distinct tags (for example, some tags like "*jekyll*", "*jestjs*", "*rake*"), the highest value among all these methods. The reason may be that the entities gain some valuable knowledge from indirected entities by the random walk. Second, we compare the average occurrence of the top-20 frequently used tags (*i.e.*, "*javascript*" , "*java*" , "*c#*") in StackOverflow between different methods. The result shows the average occurrence value in GRETA is 21.6, less than the other methods except LDA, whose value is 18.9. These metrics show that the tag assignment result of GRETA is more diverse and produces some tags that are not commonly used by users. It has the minimal impact from long tail distribution of tags.

F. Engine Evaluation

A good search engine needs to meet the following requirements: The most relevant results should be ranked in the top positions, and the result of the entire list should be relevant to the query as much as possible. Thus, we use *NDCG (Normalized Discounted Cumulative Gain)* to measure the performance of our search engine. NDCG is a measure of ranking quality. It is often used to measure effectiveness of web search engine algorithms or related applications in information retrieval.

We generate 10 typical queries, and compare our search performance against the search engine of GitHub with its *"Best Match"* type. Each query returns a list of 10 most relevant repositories. We rate each repository in the list based on the relevancy between the repository and query, and compute NDCG value using the formula:

$$N(n) = Z_n \sum_{j=1}^{n} \frac{2^{r(j)} - 1}{\log(1+j)} \qquad (10)$$

where Z_n is a normalization term and j the index of a repository in the list. $r(j)$ returns an integer score ranging from 1∼5, which reveals the relevancy between repository and query. The higher the NDCG value, the better the search performance is.

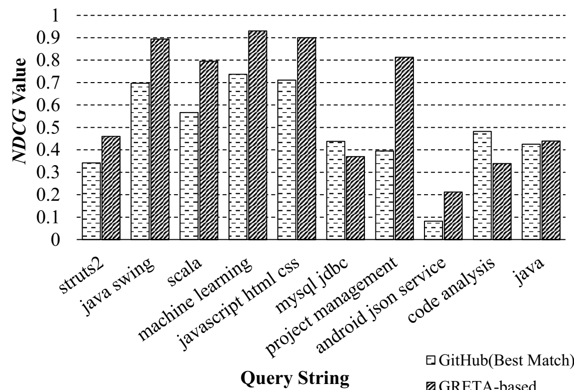


Fig. 8: NDCG value of top-10 repositories.

Figure 8 compares the performance of two engines. In general, our engine has a higher NDCG value for a majority of queries. Specifically, when people search for some popular words such as "*java*", "*struts2*", "*mysql*", both engines have the similar performance. But when the query contains multiple software characteristics or a few ones that do not appear in the repository description, such as "*javascript html css*", "*project management*", GRETA-based engine performs better than that of GitHub.

We also use an example to explain why the performance of our engine is better than that of GitHub. The GRETA-based engine returns a repository when searching "*apache spark*". The repository description is "*Set of real time algorithms used by big data streaming platform*" and its language is marked with "*Java, Scala*". We note that the relevancy between the

repository and the query is high. However, GitHub's search engine does not return this repository, as the query does not appear as any part of its name or description. The reason why our engine can return this repository is that a tag named "*apache-spark*" has been assigned to this repository by GRETA.

The evaluation results show that some latent software characteristics actually exist in repositories, which are not described explicitly in text but searched by users. Moreover, GRETA-based engine works more effectively than GitHub when searching for repositories using software characteristics.

VI. Conclusion and Future Work

GRETA is a novel graph-based approach to employing cross-community resources to assign tags to GitHub repositories. It constructs an ETG for GitHub using the domain knowledge from StackOverflow, and then uses an iterative random walk algorithm on ETG to tag repositories. We have implemented GRETA and also designed a GRETA-based search engine for searching for repositories in GitHub. The evaluation results show that GRETA outperforms four baseline methods and the search engine is more precise in searching repositories than the existing search engine in GitHub.

As future work, we plan to study how to assign tags for describing different respects of the GitHub repository. This work may enhance the capability of current tag system. Also, we plan to assign tags for users by analyzing their historical experience and community behaviors. The user-based tag system can be enriched by user technical expertise abilities in the community.

Acknowledgement

This research is supported by 973 Program in China (Grant No. 2015CB352203) and National Natural Science Foundation of China (Grant No. 61472242, 61572312). Yuting Chen is also partially supported by Science and Technology Commission of Shanghai Municipalitys Innovation Action Plan (No. 15DZ1100305).

References

[1] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 384–387, ACM, 2014.

[2] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Dept. of Informatics, Aristotle University of Thessaloniki, Greece*, 2006.

[3] G. Tsoumakas, I. Katakis, and I. Vlahavas, "Mining multi-label data," in *Data mining and knowledge discovery handbook*, pp. 667–685, Springer, 2010.

[4] G.-R. Xue, D. Xing, Q. Yang, and Y. Yu, "Deep classification in large-scale text hierarchies," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 619–626, ACM, 2008.

[5] O. Madani, M. Connor, and W. Greiner, "Learning when concepts abound," *The Journal of Machine Learning Research*, vol. 10, pp. 2571–2613, 2009.

[6] P. Turney, "Learning to extract keyphrases from text," *National Research Council Canada, Institute for Information Technology, Technical Report ERB-1057.*, 1999.

[7] E. Frank, G. W. Paynter, I. H. Witten, C. Gutwin, and C. G. Nevill-Manning, "Domain-specific keyphrase extraction," in *Proceedings of 16th International Joint Conference on Artificial Intelligence*, pp. 668–673, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] I. H. Witten, G. W. Paynter, E. Frank, C. Gutwin, and C. G. Nevill-Manning, "Kea: Practical automatic keyphrase extraction," in *Proceedings of the fourth ACM conference on Digital libraries*, pp. 254–255, ACM, 1999.

[9] W.-t. Yih, J. Goodman, and V. R. Carvalho, "Finding advertising keywords on web pages," in *Proceedings of the 15th international conference on World Wide Web*, pp. 213–222, ACM, 2006.

[10] S. N. Kim and M.-Y. Kan, "Re-examining automatic keyphrase extraction approaches in scientific articles," in *Proceedings of the workshop on multiword expressions: Identification, interpretation, disambiguation and applications*, pp. 9–16, Association for Computational Linguistics, 2009.

[11] X. Jiang, Y. Hu, and H. Li, "A ranking approach to keyphrase extraction," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 756–757, ACM, 2009.

[12] R. Mihalcea and P. Tarau, "Textrank: Bringing order into texts," in *Proceedings of EMNLP*, pp. 404–411, 2004.

[13] X. Wan and J. Xiao, "Single document keyphrase extraction using neighborhood knowledge.," in *AAAI*, vol. 8, pp. 855–860, 2008.

[14] Z. Liu, P. Li, Y. Zheng, and M. Sun, "Clustering to find exemplar terms for keyphrase extraction," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pp. 257–266, Association for Computational Linguistics, 2009.

[15] M. Grineva, M. Grinev, and D. Lizorkin, "Extracting key terms from noisy and multitheme documents," in *Proceedings of the 18th international conference on World wide web*, pp. 661–670, ACM, 2009.

[16] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," 2006.

[17] P. D. Turney, "Learning algorithms for keyphrase extraction," *Information Retrieval*, vol. 2, no. 4, pp. 303–336, 2000.

[18] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec: an enhanced tag recommendation system for software information sites," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 291–300, IEEE, 2014.

[19] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 287–296, IEEE Press, 2013.

[20] S. Wang, D. Lo, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 604–607, IEEE, 2012.

[21] F. Crestani, "Application of spreading activation techniques in information retrieval," *Artificial Intelligence Review*, vol. 11, no. 6, pp. 453–482, 1997.

[22] W. Mo, B. Shen, Y. Chen, and J. Zhu, "Tbil: A tagging-based approach to identity linkage across software communities," in *22st Asia-Pacific Software Engineering Conference, APSEC 2015, Volume 1: Research Papers*, pp. 56–63, IEEE, 2015.

[23] Z. Ma, A. Sun, Q. Yuan, and G. Cong, "Tagging your tweets: A probabilistic modeling of hashtag annotation in twitter," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 999–1008, ACM, 2014.

[24] Z. Ding, X. Qiu, Q. Zhang, and X. Huang, "Learning topical translation model for microblog hashtag suggestion," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pp. 2078–2084, AAAI Press, 2013.

[25] T. H. Haveliwala, "Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 4, pp. 784–796, 2003.

[26] J. Zhu, B. Shen, X. Cai, and H. Wang, "Building a large-scale software programming taxonomy from stackoverflow," in *The 27th International Conference on Software Engineering and Knowledge Engineering*, pp. 391–396, 2015.

[27] Z. Guo and D. Barbosa, "Robust entity linking via random walks," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 499–508, ACM, 2014.

[28] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[29] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning, "Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pp. 248–256, Association for Computational Linguistics, 2009.