Empirical Software Engineering manuscript No.
(will be inserted by the editor)

EnTagRec^{++}: An Enhanced Tag Recommendation System for Software Information Sites

Shaowei Wang · David Lo ·
Bogdan Vasilescu · Alexander Serebrenik

Received: date / Accepted: date

Abstract Software engineers share experiences with modern technologies using software information sites, such as Stack Overflow. These sites allow developers to label posted content, referred to as software objects, with short descriptions, known as tags. Tags help to improve the organization of questions and simplify the browsing of questions for users. However, tags assigned to objects tend to be noisy and some objects are not well tagged. For instance, 14.7% of the questions that were posted in 2015 on Stack Overflow needed tag re-editing after the initial assignment.

To improve the quality of tags in software information sites, we propose EnTagRec^{++}, which is an advanced version of our prior work EnTagRec. Different from EnTagRec, EnTagRec^{++} does not only integrate the historical tag assignments to software objects, but also leverages the information of users, and an initial set of tags that a user may provide for tag recommendation. We evaluate its performance on five software information sites, Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode. We observe that even without considering an initial set of tags that a user provides, it achieves *Recall@5* scores of 0.821, 0.822, 0.891, 0.818 and 0.651, and *Recall@10* scores of 0.873, 0.886, 0.956, 0.887 and 0.761, on Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode, respectively. In terms of *Recall@5* and *Recall@10*, averaging across the 5 datasets, it improves upon TagCombine, which is the prior state-of-the-art approach, by 29.3% and 14.5% respectively. Moreover, the performance of our approach is further boosted if users provide some initial tags that our approach can leverage to infer additional tags: when an initial set of tags is given, *Recall@5* is improved by 10%.

Keywords Software Information Sites · Recommendation Systems · Tagging

S. Wang (shaowei@cs.queensu.ca)
SAIL, Queen's University, Canada

D. Lo (davidlo@smu.edu.sg)
School of Information Systems, Singapore Management University, Singapore

B. Vasilescu (vasilescu@cmu.edu)
School of Computer Science, Carnegie Mellon University, United States.

A. Serebrenik (a.serebrenik@tue.nl)
Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands.

1 Introduction

The growing online media has significantly changed the way people communicate, collaborate, and share information with one another [42]. This is also true for software developers, who create and maintain software by standing on the shoulders of others [37], reuse components and libraries originating from Open Source repositories (e.g., GitHub, Freecode, SourceForge), and forage online for information that will help them in their tasks [10]. When foraging for information, developers often turn to programming question and answer (Q&A) communities such as Stack Overflow, Ask Ubuntu, and Ask Different. Such sites supporting communication, collaboration, and information sharing among developers are known as *software information sites*, while their contents (e.g., questions and answers, project descriptions)—as *software objects* [51].

Typically, tags are short labels not more than a few words long, provided as metadata to software objects in software information sites. Users can attach tags to various software objects, effectively linking them and creating topic-related structure. Tags are therefore useful for providing a soft categorization of the software objects and facilitating search for relevant information. To accommodate new content, most software information sites allow users to create tags freely. However, this freedom comes at a cost, as tags can be idiosyncratic due to users' personal terminology [17]. As tagging is inherently a distributed and uncoordinated process, often similar objects are tagged differently [51]. Idiosyncrasy reduces the usefulness of tags, since related objects are not linked together by a common tag and relevant information becomes more difficult to retrieve. Furthermore, some software information sites (e.g., Stack Overflow) require users to add tags at the time of posting a question, even if they are unfamiliar with the tags in circulation at that time. Due to differences in personal terminology and tagging purpose, it is often difficult for users to select appropriate tags for their content. Having a tag recommendation system that can suggest tags to a new object (e.g., based on how other similar objects have been tagged in the past) could (i) help users select appropriate tags easily and quickly, and (ii) in time help homogenize the entire collection of tags such that similar objects are linked together by common tags more frequently.

To illustrate the importance of tags for the well functioning of a software information site, we note the considerable amount of discussion related to tags on Meta Stack Overflow, a Q&A site with the same user interface as Stack Overflow, that focuses Stack Overflow's functioning and administration: e.g., at the time of writing there were more than 4,587 questions related to tags,[1] as opposed to only 1,312 related to user interface. Furthermore, tags on Stack Exchange sites receive considerable attention from the user community, with between 14.4% and 22.5% of questions in our experiments involving tag re-editing (see Table 1). Finally, we also note that since the earlier, conference version of our work [46], Stack Overflow has been experimenting with, and gradually phasing in across the Stack Exchange network, a tag recommendation system of their own.[2]

In this work, we introduce an automatic tag recommendation system called EnTagRec^{++}, an enhanced version of our previous approach EnTagRec [46]. EnTagRec learns from historical software objects and their tags, and recommends

[1] http://meta.stackexchange.com/questions/tagged/tags

[2] http://meta.stackexchange.com/questions/206907/how-are-suggested-tags-chosen

appropriate tags for new objects based on words that appear in the software objects. EnTagRec consists of two inference components, Bayesian and frequentist, and tries to combine the advantages of the two opposite yet complementary lines of thought in the statistics community [33].

To improve EnTagRec, in EnTagRec^{++}, we integrate two additional components into EnTagRec: User Information Component (UIC) and Additional Tag Component (ATC). We refer to EnTagRec integrated with UIC alone as EnTagRec^{+}. The intuition behind these two components is as follows:

– Users in software information sites tend to exhibit particular interests, thus software objects posted by them are likely to focus on specific domains. In UIC, we leverage this intuition to improve tag recommendation. We first link historical software objects posted by the same user together. Next, for new software objects posted by the same user, we make use of software objects that the user has posted before, to help identify tags that are associated with the new object.
– We believe that it may be easier for a user to assign one or a few initial tags to a question he/she posts, but more difficult for her to provide a comprehensive set of tags. In ATC, we make use of an initial set of tags provided by a user to help identify additional relevant tags.

We evaluate EnTagRec^{+} on datasets from five popular software information sites, Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode, by comparing it to TagCombine [48,51].[3] Our experimental results show that even without considering an initial set of tags that a user provides, our approach achieves *Recall@5* scores of 0.821, 0.822, 0.891, 0.818, and 0.651, and *Recall@10* scores of 0.873, 0.886, 0.956, 0.887, 0.761 on Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode, respectively. Compared with TagCombine, EnTagRec^{+} improves TagCombine by 29.3% and 14.5% in terms of *Recall@5* and *Recall@10*, respectively. Furthermore, to evaluate the effectiveness of ATC, we compare EnTagRec^{+}, with EnTagRec^{++}. We find that when an initial set of tags is given, on average EnTagRec^{++} improves EnTagRec^{+} by 10.0% in terms of *Recall@5*.

Our main contributions are:

– We propose EnTagRec^{++}, a novel automatic tag recommendation system for software information sites. EnTagRec^{++} composes a state-of-the-art Bayesian inference technique (labeled LDA), an enhanced frequentist inference technique that leverages a POS tagger and the spreading activation algorithm, and two other components that analyze the user who posts a software object and the initial set of tags that the user provides, to further boost the recommendation performance.
– We evaluate our proposed approach on datasets from five popular software information sites. Our study shows that our approach can achieve high recall, especially for Stack Overflow, Ask Ubuntu, Super User, and Ask Different, and outperforms a prior state-of-the-art approach.

The rest of this article is organized as follows. We provide more background on tags in several software information sites and approaches to tag recommendation

[3] Since the implementation of Stack Overflow's proprietary system is, to the best of our knowledge, not documented publicly, a meaningful comparison was not possible.

in Section 2. We present the high-level architecture of EnTagRec^{++} in Section 3, followed by detailed descriptions of the Bayesian, frequentist, user information, and additional tag inference components in Sections 4, 5, 6 and 7 respectively, and the specifics of how to integrate the four components in Section 8. We present our evaluation results in Section 9. Finally, we highlight related work in Section 10 and conclude in Section 11.

2 Preliminaries and Examples

In this section, we first describe some preliminary information on tags in software information sites. Then, we present some recent works on tag recommendation on software information sites. Finally, we show some motivating examples to illustrate why it is useful to consider incorporating user information and preliminary tags in the recommendation.

2.1 Tags in Software Information Sites

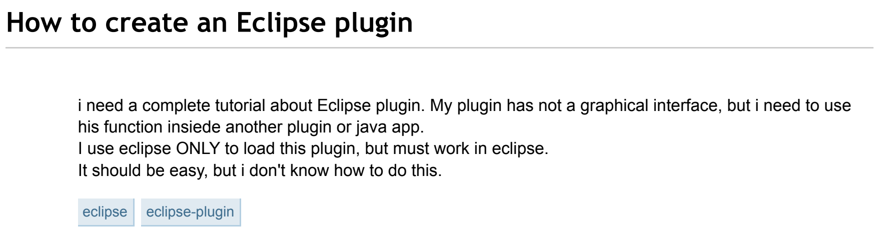
To facilitate navigation, search, and filtering, contents are marked with descriptive terms [17], known as tags; e.g., libraries associate books with authors' names and keywords, while scientific publishers require the authors to choose keywords themselves. In the digital world, tags can be used, e.g., to annotate weblog posts and shared links. Numerous software information sites employ tags, e.g., SourceForge[4] for code projects, Eclipse MarketPlace[5] for plugins, Snipplr[6] for code fragments, and Stack Overflow for questions.

An example Stack Overflow question is presented in Figure 1. The question pertains to the creation of an Eclipse plugin and it has two tags, representing the technical context of the question (`eclipse`) and a specific subject area (`eclipse-plugin`). Figure 2 shows the Freecode description of Apache Ant: in addition to the textual description, two general tags are present, `Software Development` (describing the general domain of Apache Ant) and `Build Tools` (indicating a more specific functionality of Apache Ant, namely building Java programs).

Comparing Figures 1 and 2 we observe that while the basic purpose of tagging—to facilitate navigation, search, and content filtering through the association of related contents via linked descriptive terms—is common to both, specific policies how the tags should be used differ from site to site. For instance, Freecode has no restriction on the number of tags per project, while Stack Exchange sites restrict the total number of tags given to a question to five.

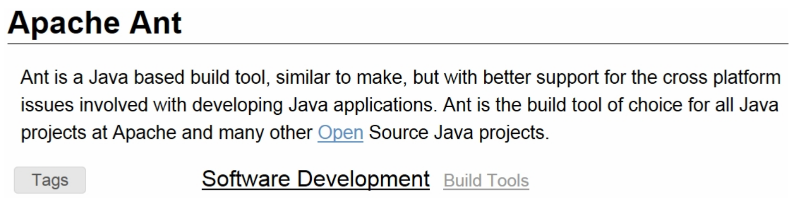
Most software information sites allow users to provide "free text tags". Not being subject to the formal requirements of the sites, such tags can be expected to represent user intent in a more flexible way. However, tagging becomes a distributed and uncoordinated process, introducing different tags for similar objects, which might persist despite moderation or the ongoing correction efforts. For example, questions on Stack Overflow entitled "SIFT and SURF feature extraction implementation

[4] http://sourceforge.net/
[5] http://marketplace.eclipse.org/
[6] http://snipplr.com/


How to create an Eclipse plugin

i need a complete tutorial about Eclipse plugin. My plugin has not a graphical interface, but i need to use his function insiede another plugin or java app.
I use eclipse ONLY to load this plugin, but must work in eclipse.
It should be easy, but i don't know how to do this.

Fig. 1 An example question in Stack Overflow and its tags.


Apache Ant

Ant is a Java based build tool, similar to make, but with better support for the cross platform issues involved with developing Java applications. Ant is the build tool of choice for all Java projects at Apache and many other Open Source Java projects.

Fig. 2 An example project in Freecode and its tags.

using MATLAB"[7] and "Matlab implementation of Haar feature extraction"[8] are both related to image feature extraction but only the second one is labeled with the corresponding tag, i.e., `feature-extraction`.

Table 1 Objects with tag re-editing on Stack Overflow, Ask Different, Ask Ubuntu and Super User.

Dataset	Period	Questions Involving Tag Re-Editing	Total Questions	Tag Re-Editing Ratio	Tag Addition Ratio
Stack Overflow	2015.1.1 - 2015.12.31	331,667	2,250,745	14.7%	72.1%
Ask Different	Before 2016.1	11,243	63,276	17.7%	87.2%
Ask Ubuntu	Before 2016.1	48,942	191,191	25.6%	69.8%
Super User	Before 2016.1	82,618	284,559	29.0%	63.1%

2.2 Tag Recommendation

Tags have been shown to aid users in navigating a site [8, 13, 20, 53]. Thus, more complete tags can help in a number of scenarios, e.g.: (1) More complete tags may shorten the time it takes for a question to receive an answer. On sites such as Stack Exchange, users are allowed to browse *unanswered questions* via tags. Giving more complete tags to a question may increase its chance to be discovered by a suitable user who can answer it well. (2) More complete tags also support developer learning. Developers can use the tags to browse through relevant questions and problems that others have encountered. This can help them avoid making similar mistakes and

[7] `http://stackoverflow.com/q/5550896`
[8] `http://stackoverflow.com/q/2058138`

improve their programming and problem solving skills. (3) More complete tags may help reduce duplicated questions. Moderators can use the tags to identify related questions; by checking these related questions, moderators can decide whether a question is a duplicated one, and if so, it can be marked accordingly. Additionally, before posting new questions, users can use tags to browse for related ones, and avoid posting questions that were answered before.

Indeed, users of Stack Exchange edited tags that were originally assigned to questions demonstrating that they appreciate more complete tags. Table 1 presents the ratio of questions involving tag re-editing on Stack Overflow, Ask Different, Ask Ubuntu, and Super User. From the table, we can see that tag re-editing happens often. From the table, we can also notice that among the tag re-editing cases, 63.1-87.2% of the questions involve tag addition, which implies that tag addition is the major tag re-editing scenario.

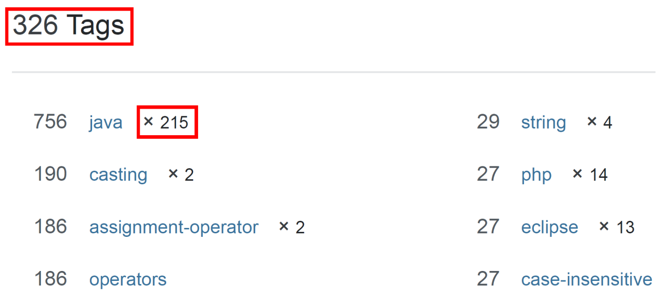
A considerable number of studies have been done on tag recommendation for software information sites [1, 51]. Among these studies, the approach TagCombine that was proposed by Xia et al. is shown to be the state-of-the-art [51] on software information sites. TagCombine combines three components: a multi-label ranking component, a similarity-based ranking component, and a tag-term based ranking component. The multi-label ranking component employs a multi-label classification algorithm (i.e., binary relevance method with naive Bayes as the underlying classifier) to predict the likelihood of a tag to be assigned to a software object. The similarity-based ranking component predicts the likelihood of a tag (to be assigned to a software object) by analyzing the tags that are given to the top-k most similar software objects that were tagged before. The tag-term based ranking component predicts the likelihood of a tag (to be assigned to a software object) by analyzing the number of times a tag has been used to tag a software object containing a term (i.e., a word) before. The multi-label ranking component of TagCombine constructs many one-versus-rest Naive Bayes classifiers, one for each tag. Each Naive Bayes classifier simply predicts the likelihood of a software object to be assigned a particular tag. However, mixture models have been shown to outperform one-versus-rest traditional multi-label classification approaches [16, 30, 31]. Thus, in our approach, we construct only one classifier which is a *mixture model* that considers all tags together to improve the effectiveness of the tag recommendation.

2.3 Motivating Examples

2.3.1 User Information

In addition to user-agnostic features (e.g., tag frequency), we also expect the information about the user posting the question to be useful when predicting tags. Indeed, one can conjecture that users have specific interests or expertise with certain technology and these interests or expertise are likely to manifest in the tags of their questions. To verify this conjecture we queried the users that asked more than one question on Stack Overflow[9] and found that 51% of them have asked at least two questions labeled with the same tag. This suggests that users may post objects associated to some particular tags, rather than all tags, based on their personal background and interests.

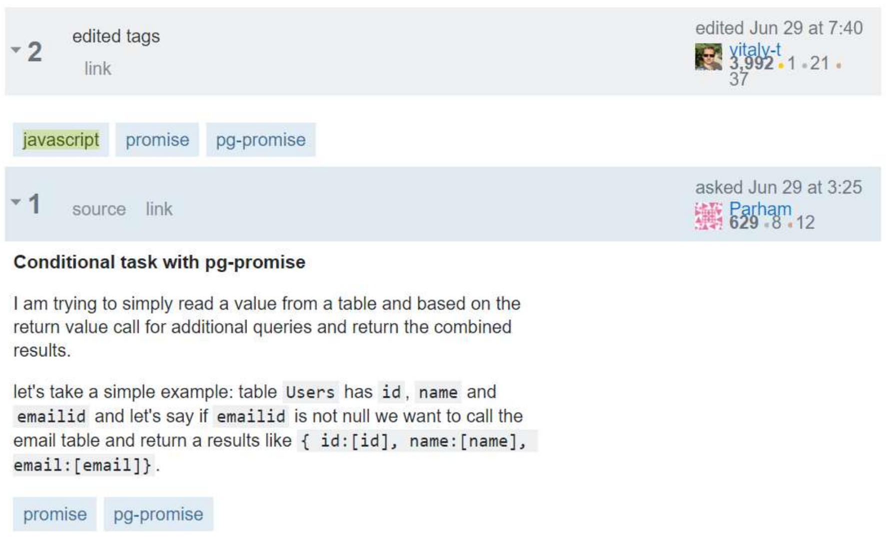
[9] https://data.stackexchange.com/stackoverflow/queries



326 Tags

756	java	× 215	29	string	× 4
190	casting	× 2	27	php	× 14
186	assignment-operator	× 2	27	eclipse	× 13
186	operators		27	case-insensitive	

Fig. 3 Highest rated tags associated to questions/answers posted by a user in Stack Overflow.



2 edited tags
link

edited Jun 29 at 7:40
vitaly-t
3,992 1 21 37

javascript promise pg-promise

1 source link

asked Jun 29 at 3:25
Parham
629 8 12

Conditional task with pg-promise

I am trying to simply read a value from a table and based on the return value call for additional queries and return the combined results.

let's take a simple example: table `Users` has `id`, `name` and `emailid` and let's say if `emailid` is not null we want to call the email table and return a results like `{ id:[id], name:[name], email:[email]}`.

promise pg-promise

Fig. 4 An example of adding a tag in Stack Overflow.

For illustration, Figure 3 presents the highest rated tags of a user in Stack Overflow.[10] As of December 7, 2016, the user posted a number of questions/answers spanning 326 tags, and 215 of the user's posts are tagged "java", this user's most commonly used tag. We can, therefore, conjecture that future software objects posted by the same user are more likely to be tagged with "java" tag, rather than other tags. Based on this observation, we can leverage user information to facilitate tag recommendation.

2.3.2 Additional Tag

In practice, it may be easy for users to label a question they post with a few tags. However, the set of initial tags may not be sufficient and for such cases, extra tags

[10] `http://stackoverflow.com/users/137369/thirler?tab=tags`

need to be added later - see Table 1. Intuitively, the initial tags can provide hints in the identification of missing tags.

We notice that some tags usually appear together. We could leverage tag co-occurrences to infer additional tags based on the initial tags that a user gave. Figure 4 presents an example of tag editing in Stack Overflow. The set of tags assigned to the question was refined – a tag "javascript" was added to the initial set of tags: "promise" and "pg-promise". When we search questions that contain tags "promise", and "pg-promise" and at least one other tag, we find 10 of such questions and 6 of them are also tagged with "javascript". This suggests given an initial set of tags "promise" and "pg-promise", it is likely that "javascript" should be included as well. This observation motivates us to recommend additional tags to users by analyzing the initial set of tags and leveraging tag co-occurrence.

3 General Architecture

In this section we describe the general architecture of our EnTagRec^{++} approach. EnTagRec^{++} contains six processing components: Preprocessing Component (PC), Bayesian Inference Component (BIC), Frequentist Inference Component (FIC), User Information Component (UIC), Additional Tag Component (ATC), and Composer Component (CC). Figure 5 presents the framework of EnTagRec^{++}.

Input software objects are processed by PC to generate a common representation. These textual documents are then input to the four main processing engines, namely BIC, FIC, UIC, and ATC. BIC and FIC infer tags based on words appearing in a software object. UIC infers tags based on the user who posts a software object; it works based on the assumption that a user tends to post similar software objects over time. ATC infers additional tags based on an initial set of tags given to a software object, by considering co-occurrences of tags. For some software objects, users who post them have provided some initial tags, and these tags can be used to better infer missing tags. CC combines the BIC, FIC, UIC, and ATC components.

EnTagRec^{++} works in two phases, a training phase and a deployment phase, as shown in Figure 5(a) and (b), respectively. In the training phase, EnTagRec^{++} trains several of its components using training software objects and corresponding tags. In the deployment phase, the trained EnTagRec^{++} is used to recommend tags for untagged software objects.

The common component in the training and deployment phase is PC, which converts each software object into a bag (or multiset) of words. The PC starts from the textual description of a software object and performs tokenization, identifier splitting, number removal, stop word removal, and stemming. Tokenization breaks a document into word tokens. Identifier splitting breaks a source code identifier into multiple words. We split a token using two splitters: 1) Camel Casing splitter [2], e.g., the identifier "getMethodName" will be split into "get", "method", and "name"; 2) special sign splitter that splits tokens based on special signs (i.e., _, −), e.g., the identifier "get_method_name" will be split into "get", "method", and "name". Number removal deletes numbers. Stop word removal[11] deletes words that are used in almost every document and, therefore, carry little document-specific meaning,

[11] Based on `http://www.textfixer.com/resources/common-english-words.txt`

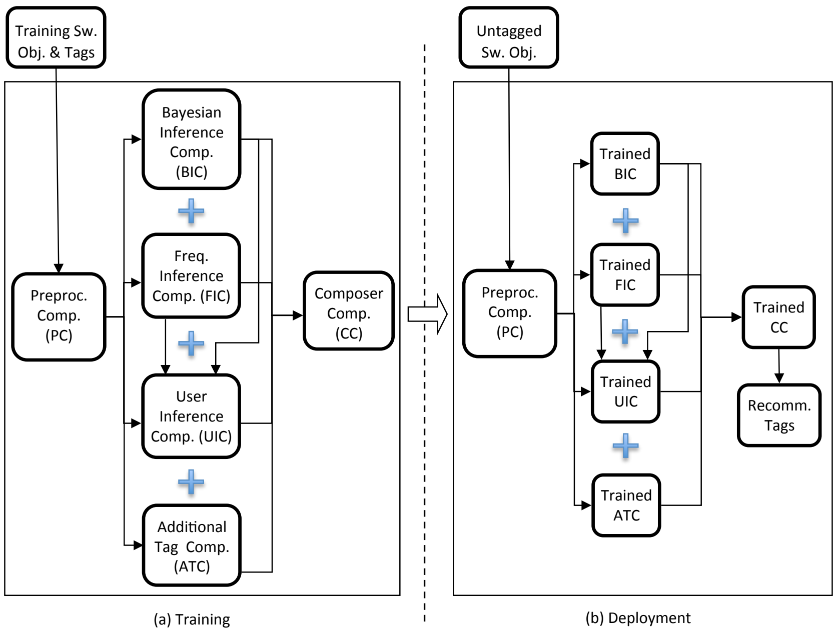


Fig. 5 EnTagRec^{++} Architecture

e.g., "the", "is", etc. Finally, stemming reduces words to their root form. We use the Porter stemming algorithm [29].

In the training phase, BIC, FIC, UIC, ATC, and CC are trained based on the training data. BIC uses the bag-of-words representation of the software objects and their corresponding tags to train itself. The result is a statistical model which takes as input a bag of words representing a software object, and produces a ranked list of tags along with their probabilities of being related to the input software object. FIC also processes the bag-of-words representations and the corresponding tags to train itself; it produces a statistical model (albeit in a different way than BIC) which also takes as input a bag of words, and outputs a ranked list of tags with their probabilities. UIC processes data about users who posted various software objects, to model the peculiar behaviors of the various users; it creates a statistical model which takes as input a user who posted a particular software object, and outputs a ranked list of tags with their probabilities. ATC takes as input a set of tags appearing together in the past. The result is a conditional statistical model, which takes an initial set of tags given by a user as input, and produces a ranked list of additional tags, with their probabilities. CC learns four weights for BIC, FIC, ATC, and UIC to generate a near-optimal combination of these four components from the training data.

After EnTagRec^{++} is trained, it is used in the deployment phase to recommend tags for untagged objects. For each such object, we first use PC to convert it to a bag of words. Next, we feed this bag of words, including the user information and additional tags (if available), to the trained BIC, FIC, UIC, and ATC. Each of them will produce a list of tags with their likelihood scores. CC will compute the final likelihood score for the tags based on the weights that it has learned in the training phase. The top few tags with the highest likelihood scores will be output as the predicted tags of an input untagged or partially tagged software object.

The following sections detail each of the five major components of EnTagRec^{++}, BIC, FIC, UIC, ATC, and CC.

4 Bayesian Inference

The goal of BIC is to compute the probabilities of various tags, given a bag of words representing a software object, using Bayesian inference. Given a tag t and a software object o, BIC computes the conditional probability of t being assigned to o, given the words $\{w_1, \ldots, w_n\}$ that appear in o. This is denoted as $P(t|w_1 \ldots w_n)$. Using the Bayes theorem [15], this probability can be computed as:

$$P(t|w_1 \ldots w_n) = \frac{P(w_1 \ldots w_n|t) \times P(t)}{P(w_1 \ldots w_n)} \tag{1}$$

The probabilities on the right hand side of the above equation can be estimated based on training data.

A state-of-the-art Bayesian inference algorithm is Latent Dirichlet Allocation (LDA) [9]. LDA has been shown effective to process various software engineering data for various tasks, e.g., [3, 4, 26, 27, 32]. LDA takes as input a set of documents and a number of topics K, and outputs the probability distribution of topics per document. Our problem can be readily mapped to LDA, where a document corresponds to a software object, and a topic corresponds to a tag. Using this setting, LDA outputs the probability distribution of tags for a software object.

However, LDA is an unsupervised learning algorithm. It does not take as input any training data and it is not possible to pre-define a set of tags as the target topics to be assigned to documents. Fortunately, recent advances in the natural language processing community introduced extensions to LDA, such as Labeled LDA (L-LDA) [31]. For L-LDA, the labels can be predefined and a training set of documents can be used to train the LDA, such that it will compute the probability distribution of topics, coming from a predefined label set (tags, in our case), for a document (a software object, in our case), based on a set of labeled training data. In this work, we use L-LDA as the basis for the Bayesian inference component.

BIC works on two phases: training and deployment. In the training phase, BIC takes as input a set of bags of words representing software objects, and their associated tags. These are used to train an L-LDA model. In the deployment phase, given a bag of words corresponding to a software object, the trained L-LDA model is used to infer the set of tags for the input software object along with their probabilities. In the end, the top $K_{Bayesian}$ inferred tags for the object will be output and fed to the Composer Component (CC).

Example. Consider an object has following words {`install, eclipse`} and a tag `eclipse`. In order to compute the probability P(`eclipse`|`install, eclipse`), we need to estimate the value of P(`install, eclipse`|`eclipse`), P(`eclipse`), and P(`install, eclipse`) first based on the Equation 1. By using L-LDA, we could estimate the the value of P(`install, eclipse|eclipse`), P(`install, eclipse`) and P(`eclipse`). Suppose the estimated value of P(`install, eclipse`|`eclipse`) = 0.02, P(`install, eclipse`) = 0.001, and P(`eclipse`) = 0.005, thus the value of P(`eclipse`|`install, eclipse`) is 0.1.

5 Frequentist Inference

FIC computes the probability that a software object is assigned a particular tag based on the words that appear in the software object, while taking into account the *number* of words that appear along with the tag in software objects in a training set. Section 5.1 describes our basic approach and several extensions are presented in Section 5.2. Hereafter, unless stated otherwise, FIC refers to the extended approach.

5.1 Basic Approach

Consider software object o with n words: $\{w_1, w_2, \ldots, w_n\}$ and a tag t, the weight of tag t for object o can be computed as the proportion of the n words that co-appear with tag t in the training data. More formally, the weight is defined as

$$W(o,t) = \frac{\sum_{w_i \in o} I(t, w_i)}{|o|}, \tag{2}$$

where

$$I(t, w_i) = \begin{cases} 1, & \exists\, o \in \textit{TRAIN} \mid o \text{ contains } w_i \;\&\; o \text{ tagged with } t \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

The higher the weight $W(o,t)$, the more representative FIC deems tag t to be for software object o.

5.2 Extended Approach

There are several problems with the basic approach. First, often not all words in a software object are related to the tags that are assigned to the software object. Although the pre-processing component (PC) has removed stop words, still many non-stop words are unrelated to software object tags, thus need to be removed. Second, we have a data sparsity problem, since many tags are not used frequently in the training set. Thus, often a tag is not characterized by sufficiently many words. To address this problem, we leverage the relationships among tags to recommend additional associated tags to an input untagged software object.

5.2.1 Removing Unrelated Words with POS Tagger

One problem in estimating the probabilities $P(t|w_i)$ is that not all words that appear in a software object are related to the tags. We use the example in Figure 1 to illustrate this. Words “need” and “work” are not stop words, but they are unrelated to the tags `eclipse` and `eclipse-plugin`. Thus, there is a need to filter out these unrelated words before we estimate the probabilities.

We observe that nouns and noun phrases are often more related to the tags than other kinds of words. Past studies have also found that nouns are often the most important words [12,34]. Thus, in this extension, we remove all words except nouns and noun phrases. To identify these nouns and noun phrases, we use the Part-Of-Speech (POS) Tagger [39] to infer the POS of each word in the representative bag of words of a software object. In this paper, we use the Stanford Log-linear Part-Of-Speech Tagger.[12] To illustrate this extension, consider the words that appear in the software object shown in Figure 1. After this step, only the words “tutorial”, “eclipse”, “plugin”, “interface”, “function”, “java”, and “app” remain.

Note that we only did this for FIC and not BIC as L-LDA assigns different probabilities to words that are associated to a topic (i.e., a tag). Unrelated words will receive low probabilities. In FIC, the words that appear in objects tagged with tag t are treated as equally important. Thus, we only perform this extended processing step for FIC.

We refer to the basic approach extended by this processing step as *FrePOS*. Given an untagged software object, *FrePOS* outputs the top $K_{Frequentist}$ tags.

5.2.2 Finding Associated Tag with Spreading Activation

Due to the data sparseness problem, *FrePOS* might miss some important tags that are not adequately represented in the training data. To find additional tags, we leverage relationships among tags using a technique named spreading activation [14]. Spreading activation takes as input a network containing weighted nodes that are connected with one another with weighted edges, and a set of starting nodes. Initially, all nodes except the starting nodes are assigned weight 0. Spreading activation then processes the starting nodes, one at a time. For each starting node, it spreads (or propagates) the node’s weight to its neighboring nodes which are at most *MH* hops away from it (where *MH* is a user-defined threshold). At the end of the process, we output all nodes with non zero weights and their associated weights. In our context, the network is a tag network, the starting nodes are the nodes corresponding to tags returned by *FrePOS*, and the weights of these starting nodes are the probabilities assigned to the corresponding tags by *FrePOS*.

To perform spreading activation, we first need to construct a network of tags. Each node in the network corresponds to a tag, and each edge connecting two nodes in the network corresponds to the relationship between the corresponding tags. The weight of each edge measures how similar two tags are. We measure this based on the co-occurrence of tags in software objects in the training set. Consider a set of tags where each of them is used to label at least one software object in the training set. We denote this set as: $Tags = \{t_1, t_2, t_3, ..., t_k\}$, where k is the total number of unique tags. We denote an edge between two tags t_i and t_j as e_{t_i,t_j}. The weight of e_{t_i,t_j} depends on the number of software objects that are tagged by t_i and t_j in the training set. It can be calculated as follows:

$$weight(e_{t_i,t_j}) = \frac{|Doc(t_j) \bigcap Doc(t_i)|}{|Doc(t_i) \bigcup Doc(t_j)|} \tag{4}$$

where $Doc(t_i)$ and $Doc(t_j)$ are the sets of objects tagged with t_i and t_j, respectively, and $|\cdot|$ denotes cardinality.

The edge connecting two tags is assigned a higher weight if the tags appear together more frequently, which means they are more associated with each other. We denote the set of edges connecting pairs of nodes as *Links*. The tag network is

[12] `http://nlp.stanford.edu/software/tagger.shtml`

Algorithm 1 Find associated tags

```
1: FindAssociatedTags
2: Input:
3: TN: Tag network
4: SST: Set of starting tags
5: MH: Maximum hop
6: Output: Set of candidate tags
7: Method:
8: Initialize the weight of each tag in TN with 0
9: for each tag t in SST do
10:     Set weight (TN[t]) = Probability of tag t inferred by FrePOS
11: end for
12: for each tag t in SST do
13:     Call SpreadingActivation(TN, TN[t], 0, MH)
14: end for
15: return $SST \cup \{t|weight(TN[t]) > 0\}$
```

Algorithm 2 Spreading activation for a node

```
1: SpreadingActivation
2: Input:
3: TN: Tag network
4: N: Current node
5: CH: Current hop
6: MH Maximum hop
7: Method:
8: if $CH > MH$ or $weight(N) = 0$ then
9:     return
10: end if
11: for each node $N'$ that is directly connected to N do
12:     Set $w = weight(N) \times weight(E(N', N))$
13:     if $weight(N') < w$ then
14:         $weight(N') = w$
15:         SpreadingActivation(TN, $N'$, CH+1, MH)
16:     end if
17: end for
```

then a graph *TN* defined as $(Tags, Links)$. Given a tag t, we denote the node in TN corresponding to t as $TN[t]$. Given a node n and an edge $E(n_1, n_2)$, we denote their weights as $weight(n)$ and $weight(E(n_1, n_2))$, respectively.

The pseudocode of our approach to infer associated tags from the initial set of tags returned by *FrePOS* is shown in Algorithm 1. The algorithm takes as input a tag network *TN* constructed from all tags in the training data, a set of starting tags *SST* returned by *FrePOS*, and a threshold *MH* that restricts the weight propagation to a maximum number of hops. Then, it initializes the weights of nodes corresponding to tags in the set of starting tags with the probabilities returned by *FrePOS*, and it sets the weights of other nodes to 0 (Lines 8–11). For each starting tag, our algorithm then performs spreading activation starting from the corresponding node in the tag network by calling the procedure `SpreadingActivation` (Lines 12–14). Finally, the algorithm outputs all nodes in the set of starting tags, along with the associated tags, which correspond to nodes in *TN* whose weights are larger than zero (Line 15).

The procedure `SpreadingActivation` spreads the weight of a node to its neighbors. It takes as input a tag network *TN*, a starting node N, the current hop *CH*, and the maximum hop *MH*. The procedure first checks if it needs to propagate the weight of node N—it only propagates if the current hop *CH* does not exceed the threshold *MH*, and the weight of the current node is larger than zero (Lines 8–10). It then iterates through nodes N' that are directly connected to N (Lines 11–17).

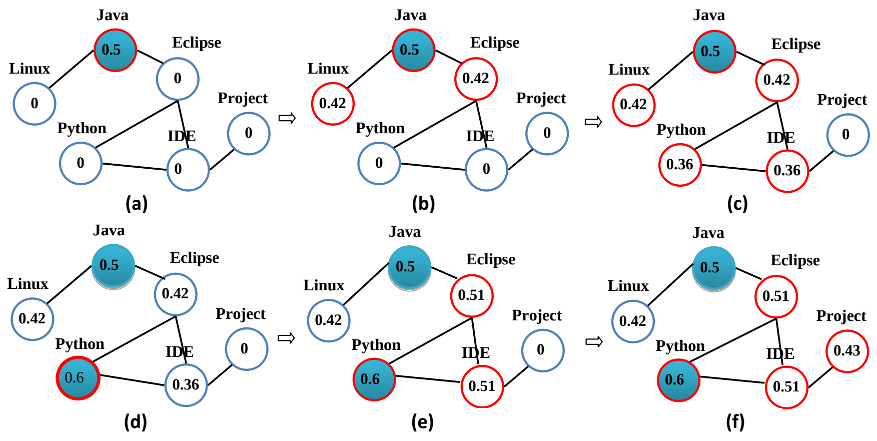


Fig. 6 Finding associated tags using spreading activation: an example.

For each such node, we compute a weight w which is a product of the weight of node N and the weight of the edge N–N' (Line 12). If the weight of node N' is less than w, we assign w as the weight of node N' (Lines 13–14). The procedure then tries to propagate the weight of N' to its neighbors by a recursive call to itself (Line 15).

Example. Consider a set of starting tags $SST = \{\text{JAVA} = 0.5, \text{PYTHON} = 0.6\}$ output by *FrePOS*, a tag network TN shown in Figure 6 and a threshold $MH = 2$. Let us assume the weights of all edges in the tag network are 0.85. At the beginning, our approach initializes the weight of the node corresponding to tag JAVA in TN with 0.5 (Figure 6(a)). Then, the weight of node JAVA is propagated to its neighbors LINUX and ECLIPSE and their weights are both updated to 0.42 (Figure 6(b)). The weight is recursively propagated to all neighbors of node JAVA of distance MH hops or less (Figure 6(c)). Then, our approach processes tag PYTHON, node PYTHON's weight is updated to 0.6, which is the weight of tag PYTHON output by *FrePOS*. (Figure 6(d)). Our approach then propagates the weight of node PYTHON to its neighbors. If a neighbor's weight is lower than that which is propagated from PYTHON, the original weight is replaced with the new weight. Otherwise, the original weight remains unchanged. Thus, the weights of ECLIPSE and IDE are updated to 0.51 (0.51 exceeds 0.425, the current weights of these tags, Figure 6(e)). The weight of node PROJECT is updated to 0.43 (Figure 6(f)). Finally, the tags $\text{JAVA} = 0.5, \text{PYTHON} = 0.6, \text{ECLIPSE} = 0.51, \text{IDE} = 0.51, \text{LINUX} = 0.42, and \text{PROJECT} = 0.43$ will be output.

The spreading activation process requires a parameter MH (maximum hop); by default, we set the parameter MH to 1, as the complexity of spreading activation is exponential to the value of MH. At the end, our FIC component outputs candidate tags that are output by *FrePOS* and the associated tags that are output by the spreading activation procedure described above. These tags are input to the composer component (CC).

Note that we only apply this spreading activation step to FIC and not BIC. L-LDA used in BIC is more robust than *FrePOS* to the data sparsity problem. We find that the application of this step to BIC does not improve its effectiveness.

6 User Information Component

In this component, we make use of tags attached to software objects that a user has posted before, in order to infer tags for a new software object posted by the same user. As we show in the Section 2.3.2, the user usually posts questions that associated to certain specific tags. Based on this intuition, we compute the weight of a tag t given a new software object o posted by a user u as:

$$w(t,o,u) = \begin{cases} \frac{|\{o \in Doc(u) | o \text{ tagged with } t\}|}{|Doc(u)|}, & \text{if } t \in T_{BIC \bigcup FIC} \\ 0, & \text{otherwise} \end{cases}, \tag{5}$$

where $Doc(u)$ is the set of past objects posted by u, $T_{BIC \bigcup FIC}$ is the set of tags with non-zero weights from BIC and FIC, and $|\cdot|$ denotes cardinality. Note that we define $w(t,o,u)$ as 0 for tags not in $T_{BIC \bigcup FIC}$ to avoid noise due to irrelevant tags [13].

Example. To illustrate how UIC works, consider the user in Figure 3. Suppose she posts a new question o. We first find questions and answers posted by her in the past. Second, we use BIC and FIC to get a list of candidate tags for the question o, say $T_{BIC \bigcup FIC}$ is {`java`, `netbeans`, `string`, `algorithm`}. Third, using Equation 5, we compute the tags' weights: `java` receives weight $\frac{203}{317}$, `string` $\frac{4}{317}$, `algorithm` $\frac{2}{317}$, and `netbeans` $\frac{2}{317}$ (she happens to have used all four tags recommended by BIC&FIC).

7 Additional Tag Component

In this component, we make use of an initial set of tags provided by a user to infer additional tags based on historical tag co-occurrences. Consider a software object o and a set of initial tags $\{t_1, t_2, ..., t_k\}$ provided by a user. The probability of the object o to be assigned a tag t is:

$$P(o,t|t_1,t_2,...,t_k) = \prod_{i=1}^{k} P(o,t|t_i) \tag{6}$$

The above probabilities ($P(o,t|t_i)$ for $1 \leq i \leq k$) can be estimated from the training data as follows:

$$P(o,t|t_i) = \frac{\text{Number of objects labeled with } t \text{ and } t_i}{\text{Number of objects labeled with } t_i} \tag{7}$$

Example. Suppose a user posts a question o and provides an initial tag `string`. In the training data, let us say there are 100 software objects labeled with tag `string`, and among them 40, 30, 10, 20, and 10 posts are also labelled with tags `java`, `c#`, `io`, `javascript`, and `python`, respectively. Using Equation 7, we estimate the probabilities: $P(o,\texttt{java}|\texttt{string}) = 0.4$, $P(o,\texttt{c\#}|\texttt{string}) = 0.3$, $P(o,\texttt{io}|\texttt{string}) = 0.1$, $P(o,\texttt{javascript}|\texttt{string}) = 0.2$ and $P(o,\texttt{python}|\texttt{string}) = 0.1$.

[13] Our experiments show that the effectiveness of UIC substantially degrades if it takes into consideration all tags.

8 Composer Component

Given a target software object o and a tag t, BIC, FIC, UIC, and ATC each produces a probability for the tag to be relevant. We need to combine these probabilities to estimate the overall relevance of the tag. A simple solution is to take an average of the four probabilities. However, this assumes that BIC, FIC, UIC, and ATC are equally accurate in predicting tag relevance, which may not be the case. To accommodate for differences in the effectiveness of the four components, we can assign weights to them. More accurate components can be given higher weights, and these weights can be learned from a training data. After these weights are learned, for every tag, we can compute a weighted average of its probabilities and use it as its overall relevance score. This score can then be used to produce a final ranked list of tags. This strategy is commonly referred to as fusion via a linear combination of scores which is a classical information retrieval technique [44].

More formally, we define EnTagRec^{++} ranking score as EnTagRec$^{++}{}_o(t)$ as follows:

$$\text{EnTagRec}^{++}{}_o(t) = \alpha \times B_o(t) + \beta \times F_o(t) + \gamma \times U_o(t) + \delta \times A_o(t), \qquad (8)$$

where $B_o(t)$, $F_o(t)$, $U_o(t)$, and $A_o(t)$ are the probabilities of tag t computed by BIC, FIC, UIC, and ATC, respectively, and α, β, γ,[14] and $\delta \in [0, 1]$ are the weights the composer component assigns to BIC, FIC, UIC, and ATC, respectively.[15] Note that if there is no additional tag provided by users, ATC will be deactivated and, correspondingly, δ will be set to 0.

To automatically tune α, β, γ, and δ, we use a set of training software objects and employ grid search [7]. The pseudocode of our weight tuning procedure is shown in Algorithm 3. The weight tuning procedure takes as input the set of training software objects *TO*, an evaluation criterion *EC*, and the four sets of tags returned by BIC, FIC, UIC, and ATC (along with their probabilities). Our tuning procedure initializes α, β, γ, and δ to 0 (Line 12). Then, it incrementally increases the value of α, β, γ, and δ by 0.1 until they reach 1.0 (Lines 13–16). For each combination of four parameters and each software object o in *TO*, our tuning procedure computes the EnTagRec^{++} scores for each tag returned by BIC, FIC, UIC, and ATC (Lines 17–19). Then tags are ordered based on their EnTagRec^{++} scores (Line 21). This is the ranked list of tags that are recommended for o. Next, our tuning procedure evaluates the quality of the resulting ranking based on particular α, β, γ, and δ values using *EC* (Line 22). The process is repeated for all objects in *TO* and again the quality of the resulting ranking is evaluated using *EC* (Line 24). The process continues until all combinations of α, β, γ, and δ have been exhausted and our tuning procedure finally outputs the best combination of α, β, γ, and δ based on *EC* (Line 29).

Various evaluation criteria can be used in our weight tuning procedure. In this paper, we make use of *Recall@k*, which has been used as the evaluation criterion in many past tag recommendation studies, e.g., [1, 52]. *Recall@k* was also used in the previous state-of-the-art study on tag inference for software information sites [51].

[14] By construction, γ is an extra weight given to some of the tags in $T_{BIC \cup FIC}$.
[15] Since EnTagRec$^{++}{}_o(t)$ is itself a probability score, it could also be expressed as a function of only three coefficients α', β', and γ', with the fourth being automatically $1 - \alpha' - \beta' - \gamma'$. We chose the four-coefficient expression to better reflect the four components of EnTagRec^{++}.

Algorithm 3 Weight Tuning Algorithm

```
1: TuneWeights
2: Input:
3: $TO$: Training Tagged Software Objects
4: $EC$: Evaluation Criterion
5: $Tags^B$: Set of tags inferred by BIC
6: $Tags^F$: Set of tags inferred by FIC
7: $Tags^U$: Set of tags inferred by UIC
8: $Tags^A$: Set of tags inferred by ATC
9: Output:
10: $\alpha$, $\beta$, $\gamma$, and $\delta$
11: Method:
12: Set $\alpha = 0$, $\beta = 0$, $\gamma = 0$, $\delta = 0$
13: for each $\alpha$ from 0 to 1, each step increases $\alpha$ by 0.1 do
14:     for each $\beta$ from 0 to 1, each step increases $\beta$ by 0.1 do
15:         for each $\gamma$ from 0 to 1, each step increases $\gamma$ by 0.1 do
16:             for each $\delta$ from 0 to 1, each step increases $\delta$ by 0.1 do
17:                 for each object $o$ in $TO$ do
18:                     for each tag $t$ in $Tags^B \bigcup Tags^F \bigcup Tags^U \bigcup Tags^A$ do
19:                         Compute EnTagRec$^{++}{}_o(t)$ according to Eq. 8
20:                     end for
21:                     Sort tags based on their EnTagRec$^{++}$ scores (desc. order)
22:                     Evaluate the effectiveness of $\alpha$, $\beta$, $\gamma$, and $\delta$ on $o$ based on $EC$
23:                 end for
24:                 Evaluate the effectiveness of $\alpha$, $\beta$, $\gamma$, and $\delta$ on $TO$ based on $EC$
25:             end for
26:         end for
27:     end for
28: end for
29: return the best $\alpha$, $\beta$, $\gamma$, and $\delta$ based on $EC$
```

Definition 1 Consider a set of n software objects. For each object o_i, let the set of its correct (i.e., ground truth) tags be $Tags_i^{correct}$. Also, let $Tags_i^{topK}$ be the top-k ranked tags that are recommended by a tag recommendation approach for o_i. *Recall@k* for n is given by:

$$Recall@k = \frac{1}{n} \sum_{i=1}^{n} \frac{|Tags_i^{topK} \bigcap Tags_i^{correct}|}{|Tags_i^{correct}|} \tag{9}$$

In the deployment phase, the composer component combines the recommendations made by the inference components by computing the EnTagRec^{++} scores using Equation 8 for each recommended tag. It then sorts the tags based on their EnTagRec^{++} scores (in descending order) and outputs the top-k ranked tags.

9 Experiments and Results

In this section, we first present our experiment settings in Section 9.1. Our experiment results are then presented in Sections 9.2. We discuss some interesting points in Section 9.3.

9.1 Experimental Setting

We evaluate EnTagRec^{++} on five datasets: Stack Overflow, Ask Ubuntu, Ask Different, Super User (all four part of the Stack Exchange network),

Table 2 Basic Statistics of the Four Datasets.

Dataset	Period	Objects	Tags	Objects per tag		Avg. age of users (days)
				Max	Avg	
Stack Overflow	2008.6 - 2008.12	47,668	437	6,113	234.93	54
Freecode	2001.1 - 2012.6	39,231	243	9,615	545.08	NA
Ask Ubuntu	Before 2012.4	37,354	346	6,169	234.03	237
Ask Different	Before 2012.4	13,351	153	2,019	180.88	253
Super User	Before 2012.4	47,996	460	7,009	245.7	745

and Freecode, which were used to evaluate EnTagRec [46]. Stack Overflow is a Q&A site for software developers to post general programming questions. Ask Different is a Q&A site related to Apple devices, e.g., iPhone, iPad, mac. Ask Ubuntu is a Q&A site about Ubuntu. Super User is a Q&A site for systems administrators and power users. Freecode is a site containing descriptions of many software projects.

Table 2 presents descriptive statistics of the four datasets, including period of the data, number of objects, number of tags, maximum and average number of objects for each per tag, and average elapsed time since registration of all studied users. The Stack Overflow and Freecode datasets are obtained from Xia et al. and they have been used to evaluate TagCombine [51]. The Ask Ubuntu, Ask Different, and Super User datasets are new. We collect all questions in Ask Ubuntu, Ask Different, Super User that are posted before April 2012. Following [51], to remove noise corresponding to tags that are assigned idiosyncratically, we filter out tags that are associated with less than 50 objects. These tags are less interesting since not many people use them, and thus they are less useful to be used as *representative tags* and recommending them does not help much in addressing the tag synonym problem addressed by tag recommendation studies. The numbers summarized in Table 2 are *after filtering*.

We perform ten-fold cross validation [19] for evaluation. We randomly split the dataset into ten subsamples. Nine of them are used as training data, to train EnTagRec^{++}, and one subsample is used for testing. We repeat the process ten times and use *Recall@k* as the evaluation metric. Note that we conduct ten-fold cross validation 100 times and take averages as results.Unless otherwise stated, we set the values of $K_{Bayesian}$ and $K_{Frequentist}$ at 70 as the setting in EnTagRec. We conduct all our experiments on a Windows 2008 server with 8 Intel®2.53GHz cores and 24GB RAM.

9.2 Evaluation Results

The goal of our evaluation is to compare the effectiveness of EnTagRec^{++} with those of EnTagRec and TagCombine. TagCombine is the tag recommendation approach proposed by Xia et al [51]. EnTagRec is our earlier version of EnTagRec^{++} [46], which did not include the user information component and the additional tag component. Our goal can be refined into the following research questions:

RQ1. How effective is EnTagRec^{+} compared to EnTagRec and TagCombine in terms of *Recall@k*?

Table 3 Basic Statistics of Questions Involving Tag Re-Editing on Stack Overflow, Ask Ubuntu, Super User, and Ask Different.

Dataset	Tags	Objects	Avg. initial tag set size	Avg. edited tag set size
Stack Overflow	649	42,493	2.9	3.4
Ask Ubuntu	483	31,881	2.4	2.8
Ask Different	157	7,762	2.3	3.1
Super User	196	13,796	2.0	2.7

To answer this research question, we perform ten-fold cross validation, and compare EnTagRec^{++}, EnTagRec and TagCombine in terms of *Recall@5* and *Recall@10*. To make the comparison fair, we do not provide an initial set of tags to EnTagRec^{++} because EnTagRec and TagCombine cannot accept an initial set of tags as input. We refer to the version of EnTagRec^{++} without additional tags as EnTagRec^{+}.

RQ2. Does the additional tag component improve the effectiveness of EnTagRec^{+}?

The additional tag component makes use of the additional tags provided by a user to recommend associated tags. We want to investigate whether this component improves EnTagRec^{+}. To answer this research question, we collect the questions whose tags have been updated in the past from Stack Overflow, Ask Ubuntu, and Ask Different. In the experiment, we take the initial tags labeled by users when they created the questions, and use the recent tags of the question as the ground truth. We also remove the tags that are associated with less than 50 objects, as before in RQ1. We use the datasets in Table 1 for this experiment; the number of tags and objects after filtering is shown at Table 3. For Ask Ubuntu, Ask Different and Super User, we collect all questions involving tag re-editing before December 2015. After filtering, we are left with 483 tags and 31,881 objects associated with the tags from Ask Ubuntu, with 157 tags and 7,762 objects from Ask Different, and with 196 tags and 13,796 objects from Super User. For Stack Overflow, because the number of questions involving tag re-editing is too large, we randomly sample 50,000 questions; after filtering, we are left with 649 tags and 42,493 objects. We do not consider Freecode for this experiment because we cannot obtain historical tag data from Freecode.

9.2.1 RQ1: Overall Effectiveness of EnTagRec^{+}

We compare EnTagRec^{+} with competing approaches: TagCombine proposed by Xia et al. [51] and EnTagRec by Wang et al [46].

Table 4 summarizes the comparison between EnTagRec^{+}, EnTagRec, and TagCombine. We also show the beanplots of the comparison between those three approaches in Figure 7. We performed ten-fold cross-validation 100 times and evaluated the approaches in terms of the average *Recall@5* and *Recall@10*. EnTagRec^{+} achieves sizeable improvements over TagCombine for the Stack Exchange datasets (more than 34.7% for *Recall@5* and more than 18.3% for *Recall@10*), and performs comparably to TagCombine on Freecode. Averaging across the 5 datasets, EnTagRec^{+} improves TagCombine in terms of *Recall@5* and *Recall@10* by 27.8% and 14.1% respectively. We perform a Wilcoxon signed-rank test [50] to test the significance of the differences in the performance of TagCombine and EnTagRec^{+}

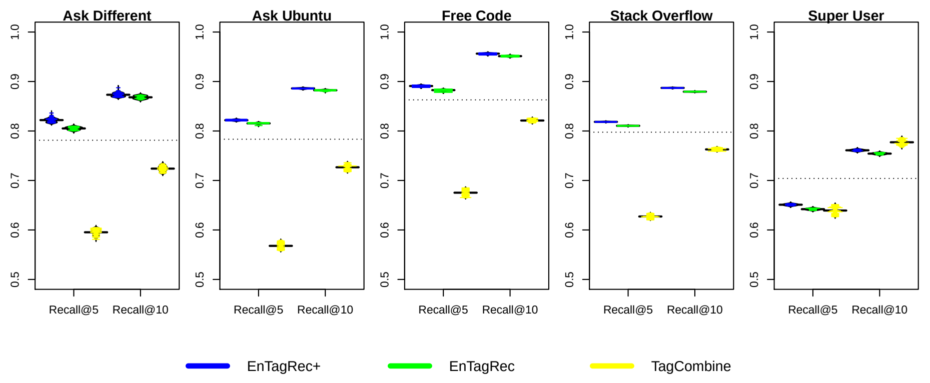


Fig. 7 Beanplots of EnTagRec$^+$, EnTagRec and TagCombine in terms of *Recall@5* and *Recall@10*.

Table 4 The Comparison of EnTagRec$^+$, EnTagRec and TagCombine in terms of *Recall@5* and *Recall@10*. The highest value is typeset in boldface.

Recall@5			
Dataset	EnTagRec	EnTagRec$^+$	TagCombine
Stack Overflow	0.805	**0.821**	0.595
Ask Ubuntu	0.815	**0.822**	0.568
Ask Different	0.882	**0.891**	0.675
Super User	0.810	**0.818**	0.627
FreeCode	0.642	**0.651**	0.639
Recall@10			
Dataset	EnTagRec	EnTagRec$^+$	TagCombine
Stack Overflow	0.868	**0.873**	0.724
Ask Ubuntu	0.882	**0.886**	0.727
Ask Different	0.951	**0.956**	0.821
Super User	0.879	**0.887**	0.763
FreeCode	0.754	0.761	**0.777**

measured in terms of *Recall@5* and *Recall@10*. We also perform Benjamini Yekutieli procedure [6] to adjust the *p*-value obtained from Wilcoxon signed-rank test to deal with the impact of multiple comparisons. For the Stack Exchange datasets (Stack Overflow, Ask Ubuntu, Ask Different, Super User), the results show that EnTagRec$^+$ outperforms TagCombine in terms of *Recall@5* and *Recall@10* significantly. For FreeCode, EnTagRec$^+$ outperforms TagCombine in terms of *Recall@5* significantly. However, TagCombine significantly outperforms EnTagRec$^+$ in terms of *Recall@10* but the absolute difference is small (0.016).

We also compare the effectiveness of EnTagRec$^+$, which employs user information, to that of EnTagRec in terms of *Recall@5* and *Recall@10*. Analyzing Table 4, we observe that EnTagRec$^+$ outperforms EnTagRec on all datasets. In terms of *Recall@5*, EnTagRec$^+$ improves EnTagRec by 1.3% on average. In terms of *Recall@10*, EnTagRec$^+$ achieves a 0.3% improvement over EnTagRec. We perform a Wilcoxon signed-rank test [50] and Benjamini Yekutieli procedure [6] on each dataset. The results indicates the improvement achieved by EnTagRec$^+$ is statistically significant (adjusted *p*-value < 0.05).

To investigate if the differences in the *Recall@5* and *Recall@10* values are *substantial*, we also compute Cliff's Delta [18] which measures effect size. The results

Table 5 Effect Sizes

Recall@5		
Dataset	EnTagRec+ vs. EnTagRec	EnTagRec+ vs. TagCombine
Stack Overflow	1	1
Ask Ubuntu	1	1
Ask Different	1	1
Super User	0.99	0.99
Freecode	1	0.92
Recall@10		
Dataset	EnTagRec+ vs. EnTagRec	EnTagRec+ vs. TagCombine
Stack Overflow	0.68	1
Ask Ubuntu	1	1
Ask Different	1	1
Super User	0.99	0.99
Freecode	1	-1

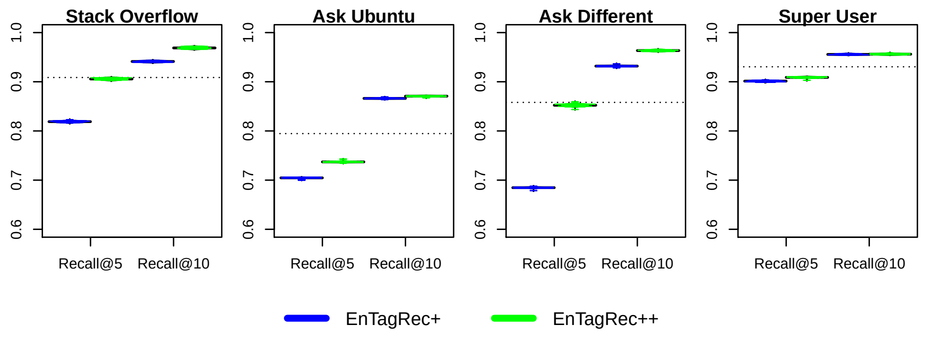


Fig. 8 Beanplots of EnTagRec+ and EnTagRec++ in terms of *Recall@5* and *Recall@10*.

are shown in Table 5. It interprets the effect size values as small for 0.147 < |d| < 0.33, medium for 0.33 < |d| < 0.474, and large for |d| > 0.474 [18]. If the effect size is close to 0, it means that the difference is not substantial. If the effect size equals to 1, it means that all values of one group are larger than those of another group. From the results we can conclude that EnTagRec+ substantially outperforms TagCombine and EnTagRec on Stack Overflow, Ask Ubuntu, Super User, and Ask Different datasets (with large effect sizes). For the Freecode dataset, EnTagRec+ also substantially outperforms EnTagRec (with large effect sizes). In terms of *Recall@5*, EnTagRec+ outperforms TagCombine substantially. However, in terms of *Recall@10*, TagCombine outperforms EnTagRec+ substantially. which means that all values in one group are larger or smaller than those in another group when comparing two groups.

9.2.2 RQ2: Effectiveness of Additional Tag Component

To answer the research question, we compare the effectiveness of two versions of EnTagRec++: one with additional tag component (EnTagRec++) and another without additional tag component (EnTagRec+). Figure 8 presents the beanplots of EnTagRec+ and EnTagRec++ in terms of *Recall@5* and *Recall@10*. Table 6 summarizes the comparison between EnTagRec++ and EnTagRec+ in terms of

Table 6 The Comparison of EnTagRec^{++} and EnTagRec^{+} in terms of *Recall@5* and *Recall@10*.

Recall@5		
Dataset	EnTagRec^{++}	EnTagRec^{+}
Stack Overflow	**0.905**	0.819
Ask Ubuntu	**0.737**	0.705
Ask Different	**0.852**	0.685
Super User	**0.908**	0.901
Recall@10		
Dataset	EnTagRec^{++}	EnTagRec^{+}
Stack Overflow	**0.968**	0.941
Ask Ubuntu	**0.87**	0.866
Ask Different	**0.963**	0.932
Super User	**0.956**	0.955

Table 7 Effect Sizes

Recall@5	
Dataset	EnTagRec^{++} vs. EnTagRec^{+}
Stack Overflow	1
Ask Ubuntu	1
Ask Different	1
Super User	0.99

Recall@10	
Dataset	EnTagRec^{++} vs. EnTagRec^{+}
Stack Overflow	1
Ask Ubuntu	0.99
Ask Different	1
Super User	0.28

Recall@5 and *Recall@10*. From the results, we notice that EnTagRec^{++} outperforms EnTagRec^{+} on all datasets. On average, EnTagRec^{++} achieves 10.0% and 4.8% improvements over EnTagRec^{+} in terms of *Recall@5* and *Recall@10*, respectively. We also perform a Wilcoxon signed-rank test [50] and Benjamini Yekutieli procedure [6] on each dataset, which indicates the improvement achieved by EnTagRec^{++} is statistically significant (adjusted p-value < 0.05). Thus, we demonstrate that additional tag component helps to improve EnTagRec when additional tag provided.

We also compute Cliff's Delta [18] which measures effect size to test if the differences in the recall values are *substantial.* The results are shown in Table 7.

9.3 Discussion

Illustrative Examples. Figure 9 shows a software object from Stack Overflow with the `ruby` and `rdoc` tags. TagCombine cannot infer any of the tags. On the other hand EnTagRec can infer all tags. This is one of the many examples where the performance of EnTagRec is better than TagCombine.

Figure 10 presents a software object from Stack Overflow with tags `python`, `apache-spark`, `apache-spark-sql`, and `pyspark`. The object is initially tagged with `python`, `apache-spark`, and `apache-spark-sql`. Later, the tag `pyspark` is added.

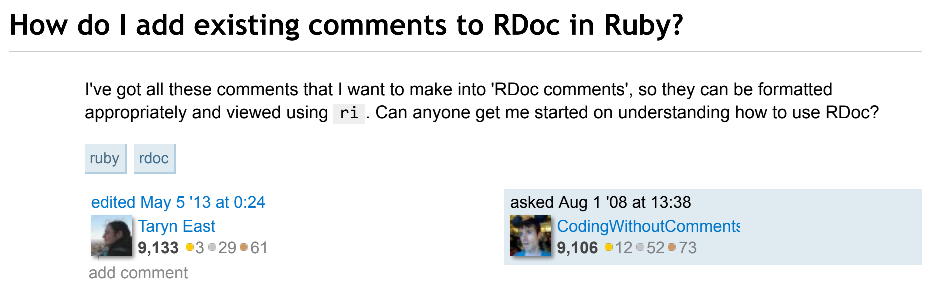
Table 8 The summary of the length (in words) of objects in the five dataset.

Dataset	Entire dataset	$Group_{short}$	$Group_{long}$
Stack Overflow	75.8	30.9	118.8
Ask Ubuntu	77.9	28.4	125.5
Ask Different	60.9	26.4	93.6
Super User	62.7	27.9	96.5
Freecode	19.5	11.6	27.3

EnTagRec^{++} can infer the tag `pyspark` given the initial set of tags, while EnTagRec fails to do so.

The impact of different MH on EnTagRec^{+}. To understand the impact of parameter MH in Algorithm 1 on our approach, we test different values of MH of EnTagRec^{+} on the five datasets and see how the effectiveness of EnTagRec^{+} varies. Figure 11 presents the results, which show that the effectiveness of EnTagRec^{+} remains stable when we increase MH from 1 to 5. Since the difference in effectiveness is negligible for different MH values, we choose to set MH to 1 to reduce the computing cost.

Stack Exchange Sites vs. FreeCode. From the experimental results, we note that EnTagRec^{+} performs much better than TagCombine on Stack Exchange Sites (i.e.,Stack Overflow, Ask Ubuntu, Super User, Ask Different), while it performs similarly to TagCombine on Freecode. To understand why the performance of EnTagRec^{+} varies on different sites, we check the length (in words) of objects in the five datasets; the summary is shown in Table 8. We see that the length of objects in Freecode is much shorter than that of Stack Exchange sites. This may explain why the performance of EnTagRec^{+} on Freecode is not as good as on the other sites. BIC is based on L-LDA, which usually requires training documents to be relatively long in order to achieve good results – c.f. [21]. Unfortunately, objects in Freecode are short, which results in poor results from BIC. To further verify our conjecture, we divide the objects into two groups. We sort the objects of each dataset by their length (in words) in ascending order. We take the top 50% of the objects as one group (i.e., $Group_{short}$) and the rest as the another group (i.e., $Group_{long}$). We evaluate the effectiveness of EnTagRec^{+} on each group of the five datasets in terms of *Recall@5* and *Recall@10*. The results are shown at Table 9. We could see that EnTagRec^{+} consistently achieves better *Recall@k* scores on $Group_{long}$, which suggests that our approach is more effective on long objects rather than short ones.


How do I add existing comments to RDoc in Ruby?

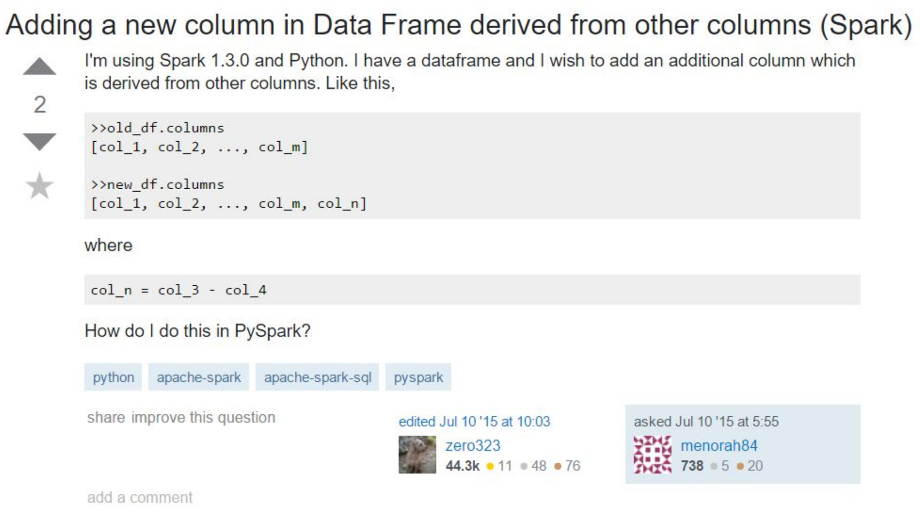
I've got all these comments that I want to make into 'RDoc comments', so they can be formatted appropriately and viewed using `ri`. Can anyone get me started on understanding how to use RDoc?

ruby rdoc

edited May 5 '13 at 0:24
Taryn East
9,133 ●3 ●29 ●61
add comment

asked Aug 1 '08 at 13:38
CodingWithoutComments
9,106 ●12 ●52 ●73

Fig. 9 EnTagRec correctly suggests tags `ruby` and `rdoc` for this Stack Overflow question, while TagCombine does not.


Adding a new column in Data Frame derived from other columns (Spark)

2

I'm using Spark 1.3.0 and Python. I have a dataframe and I wish to add an additional column which is derived from other columns. Like this,

```
>>old_df.columns
[col_1, col_2, ..., col_m]

>>new_df.columns
[col_1, col_2, ..., col_m, col_n]
```

where

```
col_n = col_3 - col_4
```

How do I do this in PySpark?

python apache-spark apache-spark-sql pyspark

share improve this question

edited Jul 10 '15 at 10:03
zero323
44.3k ● 11 ● 48 ● 76

asked Jul 10 '15 at 5:55
menorah84
738 ● 5 ● 20

add a comment

Fig. 10 EnTagRec^{++} correctly suggests tags **pyspark** for this Stack Overflow question given the initial tags **python**, **apache-spark**, and **apache-spark-sql**, while EnTagRec does not.

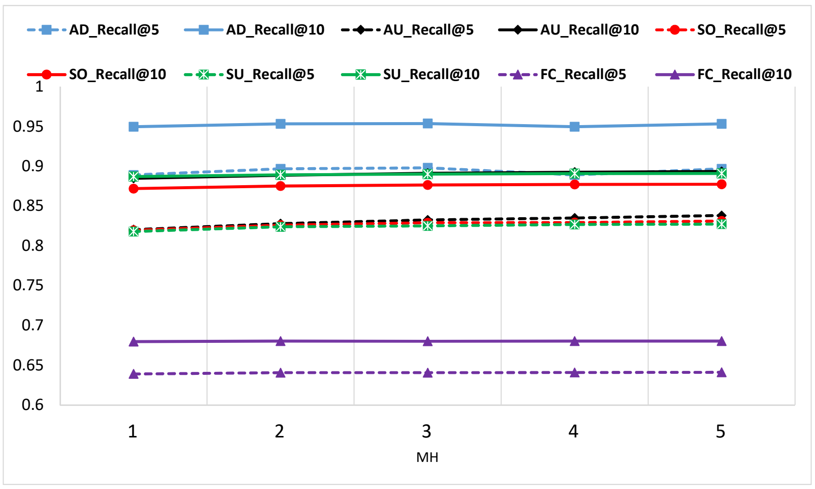


Fig. 11 The results of different values of *MH* on Stack Overflow (SO), Ask Ubuntu (AU), Ask Different (AD), Super User (SU), and Freecode (FC). Y axis is truncated (i.e., 0.6 - 1.0) and differences are smaller than they appear.

***Precision@k* Results.** Aside from *Recall@k*, *Precision@k* (definition 2) has also been used to evaluate information retrieval techniques. In this paper, we focus on *Recall@k* as the evaluation metric. A similar decision was made by past tag recommendation studies [51, 52]. This is the case as the number of tags that are attached to an object is often small (much less than K). Thus, the value of *Precision@k* is often very low and is not meaningful.

Table 9 The comparison between $Group_{long}$ and $Group_{short}$.

Recall@5		
Dataset	$Group_{long}$	$Group_{short}$
Stack Overflow	**0.912**	0.680
Ask Ubuntu	**0.882**	0.703
Ask Different	**0.954**	0.807
Super User	**0.908**	0.689
Freecode	**0.635**	0.629
Recall@10		
Dataset	$Group_{long}$	$Group_{short}$
Stack Overflow	**0.956**	0.751
Ask Ubuntu	**0.940**	0.866
Ask Different	**0.986**	0.781
Super User	**0.965**	0.778
Freecode	**0.680**	0.674

Definition 2 Consider a set of n software objects. For each object o_i, let the set of its correct (i.e., ground truth) tags be $Tags_i^{correct}$. Also, let $Tags_i^{topK}$ be the top-k ranked tags that are recommended by a tag recommendation approach for o_i. Average *Precision@k* for n is given by:

$$Precision@k = \frac{1}{n} \sum_{i=1}^{n} \frac{|Tags_i^{topK} \bigcap Tags_i^{correct}|}{|Tags_i^{topK}|} \quad (10)$$

Still, for the sake of completeness, we show the *Precision@k* results in Table 10. The results show that EnTagRec^{+} outperforms EnTagRec and TagCombine on all datasets in terms of *Precision@5*. EnTagRec^{+} outperforms EnTagRec on Ask Ubuntu, Ask Different, Super User, and Freecode in terms of *Precision@10*. In terms of *Precision@10*, EnTagRec^{+} also outperforms TagCombine on four out of the five datasets. The difference between *Precision@10* of EnTagRec^{+} and TagCombine is small (i.e., 0.008). We also performed the Wilcoxon signed-rank test [50] and Benjamini Yekutieli procedure [6]. We found that in terms of *Precision@5*, EnTagRec^{+} significantly outperforms TagCombine. Also, in terms of *Precision@10*, EnTagRec^{+} significantly outperforms EnTagRec on all datasets and TagCombine on Stack Overflow, Ask Ubuntu, Super User, and Ask Different. For Freecode, TagCombine significantly outperforms EnTagRec^{+} terms of *Precision@10*.

When the additional tags are given, the precision of EnTagRec^{+} and EnTagRec^{++} are presented at Table 11. EnTagRec^{++} outperforms EnTagRec on all datasets in terms of *Precision@5* and *Precision@10*. We also performed the Wilcoxon signed-rank test [50] and Benjamini Yekutieli procedure [6]. We found that in terms of *Precision@5*, EnTagRec^{++} significantly outperforms EnTagRec on all dataset.

To investigate if the differences in the precision values are substantial, we also compute Cliff's Delta which measures effect size. The results are shown in Table 12. From the results we can conclude that EnTagRec^{+} substantially outperforms TagCombine and EnTagRec on the Stack Overflow, Ask Ubuntu, Super User, and Ask Different datasets (with at least medium effect sizes). For the Freecode dataset,EnTagRec^{+} still substantially outperforms TagCombine in terms of *Precision@5*. However, TagCombine substantially outperforms EnTagRec in terms of *Precision@10*. EnTagRec^{++} substantially outperforms EnTagRec^{+} on Stack

Table 10 *Precision@5* and *Precision@10* for three approaches EnTagRec$^+$, EnTagRec, and TagCombine. The highest value is typeset in boldface.

Precision@5			
Dataset	EnTagRec	EnTagRec$^+$	TagCombine
Stack Overflow	0.346	**0.353**	0.221
Ask Ubuntu	0.358	**0.361**	0.251
Ask Different	0.364	**0.373**	0.278
Super User	0.376	**0.380**	0.285
Freecode	0.382	**0.396**	0.381
Precision@10			
Dataset	EnTagRec	EnTagRec$^+$	TagCombine
Stack Overflow	**0.187**	**0.187**	0.151
Ask Ubuntu	0.196	**0.197**	0.158
Ask Different	0.205	**0.202**	0.173
Super User	0.201	**0.207**	0.177
Freecode	0.240	0.241	**0.249**

Table 11 *Precision@5* and *Precision@10* for two approaches EnTagRec$^+$ and EnTagRec^{++}. The highest value is typeset in boldface.

Precision@5		
Dataset	EnTagRec^{++}	EnTagRec$^+$
Stack Overflow	**0.225**	0.202
Ask Ubuntu	**0.202**	0.191
Ask Different	**0.225**	0.181
Super User	**0.249**	0.247
Precision@10		
Dataset	EnTagRec^{++}	EnTagRec$^+$
Stack Overflow	**0.122**	0.118
Ask Ubuntu	**0.122**	0.121
Ask Different	**0.130**	0.125
Super User	**0.133**	0.132

Overflow, Ask Ubuntu, Super User, and Ask Different datasets (with large and medium effect sizes). EnTagRec$^+$ substantially outperforms EnTagRec in terms of *Precision@5*. In terms of *Precision@10*, EnTagRec$^+$ outperforms EnTagRec on the Ask Ubuntu, Super User, and Ask Different datasets with large size and on the Freecode with medium size.

Efficiency. We find that EnTagRec^{++} runtimes for the training and deployment phases are reasonable. EnTagRec^{++}'s training time can mostly be attributed to training an L-LDA model in the Bayesian inference component of EnTagRec^{++}, which never exceeds 18 minutes (it is the maximum time across the ten iterations measured on the Stack Overflow dataset, the largest of the four). The Frequentist inference component is much faster; its runtime never exceeds 40 seconds (measured on the Stack Overflow dataset). The training time of user information component and additional tag component never exceeds 1 minute. In the deployment phase, the average time EnTagRec^{++} takes to recommend a tag never exceeds 0.14 seconds.

Retraining frequency. Since EnTagRec^{++} is efficient (i.e., model training can be completed in minutes), we can afford to retrain it daily. For example, a batch script can be run at a scheduled hour every day. Within a day, software objects and tagging behaviors are very likely to remain unchanged, and thus there is no need to retrain EnTagRec^{++} more frequently.

Table 12 Effect Sizes (Precision)

Precision@5			
Dataset	EnTagRec^{+} vs. EnTagRec	EnTagRec^{++} vs. EnTagRec^{+}	EnTagRec^{+} vs. TagCombine
Stack Overflow	0.939	1	1
Ask Ubuntu	1	1	1
Ask Different	1	1	1
Super User	0.461	0.92	1
Freecode	1	NA	1
Precision@10			
Dataset	EnTagRec^{+} vs. EnTagRec	EnTagRec^{++} vs. EnTagRec^{+}	EnTagRec^{+} vs. TagCombine
Stack Overflow	-0.024	1	1
Ask Ubuntu	1	0.568	1
Ask Different	1	1	1
Super User	0.99	0.47	0.99
Freecode	0.341	NA	-0.457

ATC usage. Since more complete tags may shorten the time it takes for a question to be discovered and receive answer, we suggest to apply ATC just after a question is created with initial tags. Moreover, ATC can even be applied in real time, i.e., when users are entering tags.

Threats to Validity. Threats to external validity relate to the generalizability of our results. We have analyzed five popular software information sites (i.e., four Stack Exchange sites and Freecode) and more than 160,000 software objects. In the future, we plan to reduce this threat further by analyzing even more software objects from more software information sites. As a threat to internal validity, we assume that the data in the software information sites are correct. To reduce the threat we only used older data—assuming people correct wrongly/poorly assigned tags. Also, two of our datasets (i.e., Stack Overflow and Freecode) were used in a past study [51]. We use a lot of data and only consider tags that are used to label at least 50 objects to further reduce the impact of noise. Furthermore, manual inspection of a random sample of 100 Stack Overflow objects (questions) revealed that only 1 had a clearly irrelevant tag (out of a total of 3 tags for that object).

Threats to construct validity relate to the suitability of our evaluation metrics. We have used *Recall@k* and *Precision@k* to evaluate our proposed approaches EnTagRec and EnTagRec^{++} in comparison with other approaches. These measures are standard information retrieval measures used by prior tag recommendation studies, e.g., [1,51,52]. We have also performed statistical test and effect size test to check if the differences in *Recall@k* and *Precision@k* are significant and substantial. Thus, we believe there is little threat to construct validity.

10 Related Work

Tag Recommendation: Al-Kofahi et al. proposed TagRec which recommends tags in work item systems (e.g., IBM Jazz) [1]. There are a number of studies from the data mining research community, that recommend tags for social media sites like Twitter, Delicious, and Flickr [22,35,52]. Among these studies, the work by Zangerle et al. is the latest approach to recommend hashtags for short messages in Twitter [52].

Xia et al. proposed TagCombine, which combines three components: a multi-label ranking component, a similarity-based ranking component, and a tag-term based ranking component [51]. Xia et al. have shown that TagCombine outperforms TagRec and Zangerle et al.'s approach in recommending tags in software information sites.

The closest work to ours is TagCombine proposed by Xia et al. which is also the prior state-of-the-art work [51]. There are a number of technical differences between EnTagRec, proposed in our preliminary work [46], and TagCombine. EnTagRec combines two components: a Bayesian inference component that employs Labeled LDA (BIC), and an enhanced frequentist inference component that removes unrelated words with the help of a parts-of-speech (POS) tagger, and finds associated tags with a spreading activation algorithm (FIC). Our BIC is related to the multi-label ranking component of TagCombine since both of them employ Bayesian inference. The multi-label ranking component of TagCombine constructs many one-versus-rest Naive Bayes classifiers, one for each tag. Each Naive Bayes classifier simply predicts the likelihood of a software object to be assigned a particular tag. In EnTagRec, we construct only one classifier which is a *mixture model* that considers all tags together. Mixture models have been shown to outperform one-versus-rest traditional multi-label classification approaches [16, 30, 31]. Also, our FIC removes unrelated words (using POS tagger) and finds associated tags (using spreading activation) while none of the three components of TagCombine perform these. We have compared our approach with TagCombine, on four datasets: Stack Overflow, Ask Ubuntu, Ask Different, and Freecode. We show that our approach outperforms TagCombine on three datasets (i.e., Stack Overflow, Ask Ubuntu, Ask Different), and performs as well as TagCombine on one dataset (i.e., Freecode). EnTagRec^{++} extends EnTagRec by including two additional components, User Information Component (UIC) and Additional Tag Component (ATC), which boosts performance further.

Tagging in Software Engineering: The need for automatic tag recommendation has been recognized both by practitioners [23, 25, 49] and by researchers. Aside from tag recommendation studies mentioned above, there are several software engineering studies that also analyze tagging and leverage tags for various purposes. Treude et al. performed an empirical study on the impact of tagging on a large project with 175 developers over a two years period [40]. Wang et al. analyzed tags of projects in Freecode, inferred the semantic relationships among the tags, and expressed the relationships as a taxonomy [45]. Thung et al. detected similar software applications using software tags [38]. Storey et al. proposed an approach called TagSEA that allows one to create, edit, navigate, and manage annotations in source code [36]. Treude et al. performed an empirical study on several professional projects that involved more than 1,000 developers, and found that tagging can play an important role in the development process [41]. They found that tags are helpful in articulation work, finding of tasks, and exchange of information. Cabot et al. conducted an empirical study on the labels that are used to classify issues on issue tracking system and they found that the use of such labels improves issue resolution process [11]. Wang et al. have demonstrated that the practice of tagging helps in assisted tracing (a process where analysts inspect results produce by automated traceability techniques) [47]. Through a user study, they find that tagging is readily adopted by analysts and improve the quality of the trace matrices produced at the end of the study.

Furthermore, several studies of Stack Overflow have used tags to focus on questions or answers pertaining to a certain technology [5, 43] or to enhance studies of related websites such as GitHub [28] or Wikipedia [24].

11 Conclusion and Future Work

In this work, we propose a novel approach to recommend tags to software information sites. Our approach, named EnTagRec^{++}, an enhanced version of EnTagRec, learns from tags of historical software objects to infer tags of new software objects. To recommend tags, EnTagRec^{++} enhances EnTagRec by adding two more inference components. One, named user information component (UIC), makes use of historical tagging information peculiar to a user to infer tags for a current software object the user creates. Another one, named additional tag component (ATC), makes use of an initial set of tags given by a user to recommend additional tags better. EnTagRec^{++} composes the four components by finding the best weights that optimize the performance of EnTagRec^{++} on a training dataset. We evaluate the performance of EnTagRec^{++} on four datasets, Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode, which contain 47,688, 39,231, 37,354, and 13,351 software objects, respectively. We find that that without leveraging ATC, our approach (named EnTagRec^{+}) achieves *Recall@5* scores of 0.821, 0.822, 0.891 and 0.651, and *Recall@10* scores of 0.873, 0.886, 0.956 and 0.761, on Stack Overflow, Ask Ubuntu, Ask Different, Super User, and Freecode, respectively. In terms of *Recall@5* and *Recall@10*, averaging across the 4 datasets, EnTagRec^{+} improves TagCombine [51], which is the prior state-of-the-art approach, by 29.1% and 14.2% respectively. In addition, with ATC, EnTagRec^{++} achieves a 13.1% improvement over EnTagRec^{+} in terms of *Recall@5*. We have published the code and datasets that we used online [16]. Admittedly, we have only tested our approach on Stack Exchange sites and FreeCode.

As future work, we plan to reduce the threats to validity by experimenting with more software objects from more software information sites. In this paper, we only consider the major tag re-editing scenario – tag addition (see Table 1). In the future, we also plan to support tag deletion and tag correction. Furthermore, we plan to improve the *Recall@5* and *Recall@10* of EnTagRec further by investigating cases where EnTagRec^{++} is inaccurate, and by building a more sophisticated machine learning solution.

References

1. J. M. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Fuzzy set approach for automatic tagging in evolving software. In *ICSM*, pages 1–10, 2010.
2. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Oct. 2002.
3. H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *ICSE*, pages 95–104, 2010.

[16] https://sites.google.com/site/wswshaoweiwang/projects/entagrec

4. P. Baldi, C. V. Lopes, E. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, pages 543–562, 2008.
5. B. Bazelli, A. Hindle, and E. Stroulia. On the personality traits of stackoverflow users. In *2013 IEEE International Conference on Software Maintenance*, pages 460–463, Sept 2013.
6. Y. Benjamini and D. Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, 29:1165–1188, 2001.
7. J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13:281–305, 2012.
8. S. Bindelli, C. Criscione, C. Curino, M. L. Drago, D. Eynard, and G. Orsi. Improving search and navigation by combining ontologies and social tags. In *On the Move to Meaningful Internet Systems: OTM 2008 Workshops, OTM Confederated International Workshops and Posters, ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent + QSI, ORM, PerSys, RDDS, SEMELS, and SWWS 2008, Monterrey, Mexico, November 9-14, 2008. Proceedings*, pages 76–85, 2008.
9. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *JMLR*, pages 993–1022, 2003.
10. J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI*, pages 1589–1598. ACM, 2009.
11. J. Cabot, J. L. C. Izquierdo, V. Cosentino, and B. Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 550–554, 2015.
12. G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.
13. U. Cress, C. Held, and J. Kimmerle. The collective knowledge of social tags: Direct and indirect influences on navigation, learning, and information processing. *Computers & Education*, 60(1):59–73, 2013.
14. F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.
15. A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. CRC Press, 2003.
16. N. Ghamrawi and A. McCallum. Collective multi-label classification. In *CIKM*, pages 195–200, 2005.
17. S. A. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2):198–206, Apr. 2006.
18. R. J. Grissom and J. J. Kim. Effect sizes for research: A broad practical approach, 2005.
19. J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2011.
20. C. Held, J. Kimmerle, and U. Cress. Learning by foraging: The impact of individual knowledge and social tags on web navigation processes. *Computers in Human Behavior*, 28(1):34–40, 2012.
21. L. Hong and B. D. Davison. Empirical study of topic modeling in twitter. In *Proceedings of the First Workshop on Social Media Analytics*, SOMA '10, pages 80–88, 2010.
22. R. Jäschke, L. B. Marinho, A. Hotho, L. Schmidt-Thieme, and G. Stumme. Tag recommendations in folksonomies. In *PKDD*, 2007.
23. jmac. Select and display 'suggested tags' for all posts based on related questions (or other logic), Sept. 2013. `http://meta.stackexchange.com/q/196702/182512`.
24. A. Joorabchi, M. English, and A. E. Mahdi. Automatic mapping of user tags to wikipedia concepts: The case of a q&a website âĂŞ stackoverflow. *Journal of Information Science*, 41(5):570–583, 2015.
25. Jud.Her. Tag recommendations for Stack Overflow, Apr. 2011. `http://meta.stackexchange.com/q/88611/182512`.
26. S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972–990, 2010.
27. A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. D. Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE*, pages 522–531, 2013.
28. D. Pletea, B. Vasilescu, and A. Serebrenik. Security and emotion: Sentiment analysis of security discussions on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 348–351, New York, NY, USA, 2014. ACM.

29. M. F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, pages 313–316. Morgan Kaufmann, 1997.
30. A. Puurula. Mixture models for multi-label text classification. In *10th New Zealand Computer Science Research Student Conference*, 2011.
31. D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled lda: a supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP '09*, pages 248–256, 2009.
32. M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor. An empirical study on the usage of the swift programming language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 634–638, March 2016.
33. F. I. Samaniego. *A Comparison of the Bayesian and Frequentist Approaches to Estimation.* Series in Statistics. Springer, 2010.
34. R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *MSR*, 2013.
35. B. Sigurbjörnsson and R. van Zwol. Flickr tag recommendation based on collective knowledge. In *WWW '08*, pages 327–336, 2008.
36. M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, and M. Muller. How software developers use tagging to support reminding and refinding. *IEEE Transactions on Software Engineering*, 35(undefined):470–483, 2009.
37. M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *FoSER '10*, pages 359–364, 2010.
38. F. Thung, D. Lo, and L. Jiang. Detecting similar applications with collaborative tagging. In *ICSM*, pages 600–603, 2012.
39. K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL*, 2003.
40. C. Treude and M.-A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *ICSE '09*, pages 12–22, 2009.
41. C. Treude and M.-A. Storey. Work item tagging: Communicating concerns in collaborative software development. *IEEE Trans. Softw. Eng.*, 38(1):19–34, Jan. 2012.
42. B. Vasilescu, A. Serebrenik, P. T. Devanbu, and V. Filkov. How social Q&A sites are changing knowledge sharing in open source software communities. In *CSCW*, pages 342–354, 2014.
43. B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand. The babel of software development: Linguistic diversity in open source. In A. Jatowt, E.-P. Lim, Y. Ding, A. Miura, T. Tezuka, G. Dias, K. Tanaka, A. Flanagin, and B. T. Dai, editors, *Social Informatics: 5th International Conference, SocInfo 2013, Kyoto, Japan, November 25-27, 2013, Proceedings*, pages 391–404. Springer International Publishing, 2013.
44. C. C. Vogt and G. W. Cottrell. Fusion via a linear combination of scores. *Inf. Retr.*, 1(3):151–173, Oct. 1999.
45. S. Wang, D. Lo, and L. Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *ICSM*, pages 604–607, 2012.
46. S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik. EnTagRec: An enhanced tag recommendation system for software information sites. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 291–300. IEEE Computer Society, 2014.
47. W. Wang, N. Niu, H. Liu, and Y. Wu. Tagging in assisted tracing. In *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*, pages 8–14, May 2015.
48. X.-Y. Wang, X. Xia, and D. Lo. Tagcombine: Recommending tags to contents in software information sites. *Journal of Computer Science and Technology*, 30(5):1017–1035, 2015.
49. D. Warbox. Auto-tagging, July 2009. `http://meta.stackoverflow.com/questions/1377/auto-tagging`.
50. F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(4):80–83, 1945.
51. X. Xia, D. Lo, X. Wang, and B. Zhou. Tag recommendation in software information sites. In *MSR '13*, pages 287–296, 2013.
52. E. Zangerle, W. Gassler, and G. Specht. Using tag recommendations to homogenize folksonomies in microblogging environments. In *SocInfo'11*, pages 113–126, 2011.
53. A. Zubiaga. Enhancing navigation on wikipedia with social tags. *CoRR*, abs/1202.5469, 2012.