Recommendflow: Use Topic Model to Automatically Recommend Stack Overflow Q&A in IDE

Sun Fumin(✉), Wang Xu, Sun Hailong, and Liu Xudong

Beihang University, Beijing, China
{sunfumin,wangxu,sunhl,liuxd}@act.buaa.edu.cn

Abstract. Developers often look information in the web during software development and maintenance. That means they spend time to formulate query, retrieve documents and process the results from many sources of information. Stack Overflow, one of the most popular question and answer sites and the most important information sources for developers, has become one of the most important information sources for developers. In this paper, we proposed a new approach that use LDA model and Q&A meta-information to automatically generate query from code context and recommend the retrieval Q&A to developers. We implemented the approach in Recommendflow, an Eclipse plugin. We considered one existing recommendation model as baseline and conducted an experiment to compare our approach with baseline. Our experiment on the test data set shows that LDA-based model outperforms existing Stack Overflow recommendation model.

Keywords: Crowd knowledge · Software development · Q&A · Topic model · Recommend system

1 Introduction

With the explosive development of software technology, developers face new software or programming issues more and more often. Developers always need knowledge beyond that they have already processed [1]. Stack Overflow is one of the most popular and important ways to look for information on the Internet [2].

However, retrieving information in internet forces developers switching between IDE and the web browsers, that is really time-consuming [3]. To solve this problem, some scholars have already proposed studies that integrate Stack Overflow into IDE which can help developers work efficiently. *Seahawk* [4], an eclipse plugin proposed by Ponzanelli et al. [6] in 2013, offers a way to retrieve and recommend Stack Overflow posts within IDE. Rahman et al. [5] propose an eclipse plugin, *SurfClipse*, that can retrieve Stack Overflow posts according to exceptions that are selected from console view or error log by developers. All the studies mentioned above just get the meta information in Stack Overflow (eg. votes, user reputation and posts content). We use LDA model [7], a suit of

S. Wang and A. Zhou (Eds.): CollaborateCom 2016, LNICST 201, pp. 521–526, 2017.
DOI: 10.1007/978-3-319-59288-6_50

machine learning algorithms aim to discover latent topic structure in collections of documents.

The contribution of the paper are the following points. First we proposed LDA-based query generation model. We discuss query generation tasks and present LDA-based query generation model in Sect. 2. Second we proposed LDA-based ranking model and combine LDA-based Ranking with Meta-information ranking in Sect. 3. We implement this approach in Recommendflow, an Eclipse plugin, which is detailed in Sect. 4. Then we collected a test data set and conducted our experiment on the test data set in Sect. 5. Finally, Sect. 6 concludes and discusses possible direction for future work.

2 Query Generation

Our query generation model is motivated by the information retrieval ranking. In information retrieval, the basic approach for ranking query results is the query likelihood model where each document is scored by the probability of the model generating a query:

$$P(Q|D) = \prod_{q \in Q} P(q|D) \tag{1}$$

where d is a document, Q is the query and q is a query term in Q. $P(q|D)$ is the probability of the document model generating a term. The score shows how much a document matches the query. However, in query generation model there is not a given query. If all possible generated queries of the document are computed, we can easily generate one or Top N best query as follow:

$$P(Q|D) = Q = \arg\max_{Q \in \Omega} P(Q|D) = \arg\max_{Q \in \Omega} \prod_{i=1}^{L} P(q_i|D) \tag{2}$$

Note that since query has a fixed length, we just need to compute all $P(q|D)$ and then get Top L query terms to maximize the $P(Q|D)$. In this section, we introduce two methods of computing $P(q|D)$.

In LDA, when we get the posterior estimates of θ and φ, we can compute the probability of a term generating in a document as:

$$P(q|D, \hat{\theta}, \hat{\varphi}) = \sum_{z=1}^{K} P(q|z, \hat{\varphi}) P(z|\hat{\theta}, D) \tag{3}$$

where $\hat{\theta}$ and $\hat{\varphi}$ are the posterior estimates of θ. Blei et al.[9] used a variational Bayes approximation of the posterior distribution. There are alternative inference techniques: expectation propagation and Gibbs Sampling. We use Gibbs Sampling technique in this paper, so we can get $P(q|D)$ as follow:

$$p(q|D) = \sum_{z=1}^{K} \frac{c_{v,z}^{-} + \beta}{c_z + V\beta} \frac{c_{i,k}^{-} + \alpha}{L_i + K\alpha} \tag{4}$$

where $c^{-}_{v,z}$ is the number of instances of term q_i assigned to topic z except for the current word, with hyper-parameters α and β. $c^{-}_{i,k}$ is the number of terms in ith document assigned to topic z except for the current word, c_z is the total number of words assigned to topic z and L_i is the length of the ith document.

3 Ranking

As mentioned before, in information retrieval, scores between query and documents is used to sort results. In this paper we compute the score between code context and document instead of the score between query and document, because the query is generated from code context. We introduce two method of computing score of Stack Overflow posts, S_D.

As we mentioned in Sect. 2, the posterior estimates of θ and φ can be used to compute $P(t|D, \theta)$, the probability of document D generating the topic t. Thus we can get a probability topic vector for every document. And with the posterior estimates, LDA model can infer topic structure in developer's code context. We use *Cosine Similarity* to measure the similarity between the topic vector of Stack Overflow posts and the topic vector of developer's code context. Finally we define the S_D as:

$$S_D = \frac{\sum_{t=1}^{K} P(z|D, \varphi) \cdot P(z|D', \varphi')}{\sqrt{\sum_{i}^{K} P(z|D, \varphi)^2} \cdot \sqrt{\sum_{i}^{K} P(z|D', \varphi')}} \tag{5}$$

where D is the Stack Overflow post,D' is code context. LDA treat the Q&A as a vector of words which may ignore meta-information such as votes, user reputation and so on. So we combine LDA and meta-information. Note that if consider similarity between two topic vectors as a feature, we can combine LDA-based Model with meta information. We update the LDA-based Ranking model as:

$$\begin{aligned} S_D =& \lambda \frac{\sum_{t=1}^{K} P(z|D, \varphi) \cdot P(z|D', \varphi')}{\sqrt{\sum_{i}^{K} P(z|D, \varphi)^2} \cdot \sqrt{\sum_{i}^{K} P(z|D', \varphi')}} \\ &+ (1 - \lambda) \sum_{i=1}^{n} w_i \cdot s_i \end{aligned} \tag{6}$$

There is some meta information in Stack Overflow from which we can extract features as follow:

Question Votes: Every user can give every question a positive vote or a negative vote. It reflects the quality of the question.

Accepted Answer Votes: The accepted answer votes is normalized like the question votes. It reflects the quality of the accepted answer.

Questioner Reputation: Every user on Stack Overflow community has a reputation. It is a rough measurement of how much the community trusts the user.

Answerer Reputation: The reputation of user whose answer is accepted in the question.

Content Similarity: The similarity between retrieved content of Stack Overflow posts and developer's code context. We first split the contents to words then removing stop words and apply Porter Stemming. Finally we compute similarity of two word vectors.

API method Similarity: It's similar with content similarity but compute similarity of API method word vector which extract from content and code context.

4 Recommendflow

Figure 1 show the architecture of *Recommendflow*. It is based on client-server model which consists of two major components - client (Eclipse plugin) and server (web service server). They communicate through Hyper Text Transfer Protocol and send files in json format to each other.

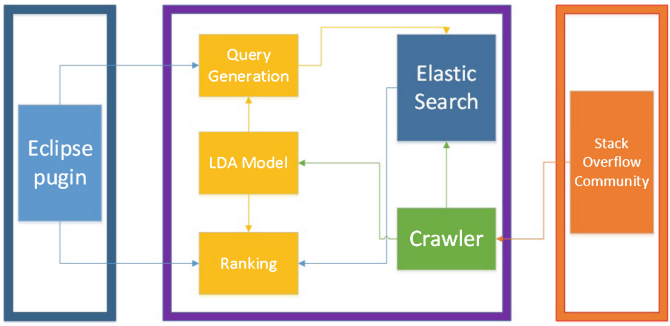

Fig. 1. Architecture.

To initiate, crawler crawled all Stack Over flow data (eg. question&answer posts, user data) and processed them. Server added the processed data to elastic search and trained LDA model on these data. Once developers change their source code, the plugin sends code context to server side where LDA model infers latent topics structure of it and query generation service formulates query with the inferred structure. Then the server calls a new search through Elastic Search. After computed score and sorted in *Ranking Service*, the retrieved posts will be sent to plugin. To ensure our data are up to date, the server crawls Stack Overflow and retrains LDA model regularly.

As we introduced before, *Recommenflow* both automatically recommends Stack Overflow posts and offers search service. If developers formulate their query by themselves, the server just skips over generating query step and retrieves documents directly. Figure 2 represents *Recommendflow* user interface.

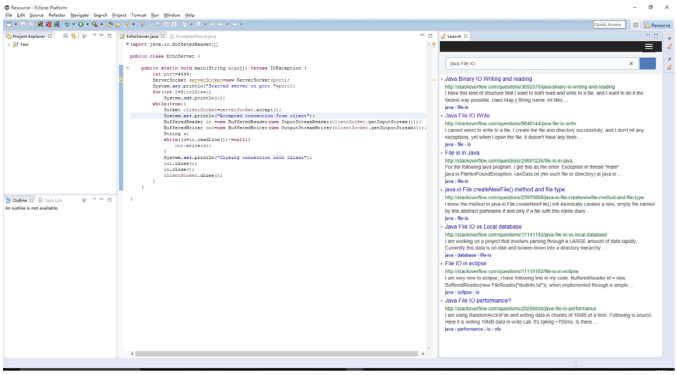

Fig. 2. Recommendflow User Interface.

5 Experiment

To evaluate our approach, we conducted our experiments on the test data set which is collected by ourselves. The test data set consisted of some codes and related Q&A. All related Q&A conducted a Q&A set. We processed each code and recommend Q&A from the Q&A set and record if the recommended is related to the code. We considered recommending model in [6] as baseline and compared our approach and baseline. The experiment result is presented in Fig. 3.

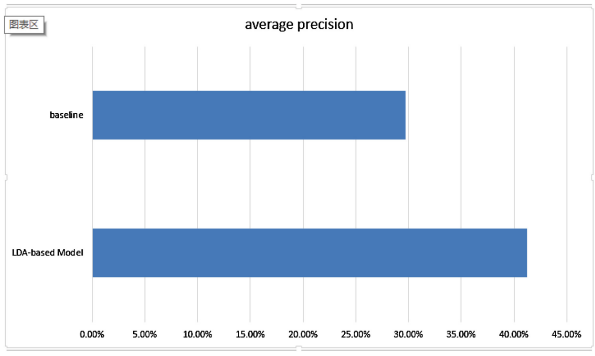

Fig. 3. Experiment result of the test data set in average precision.

6 Conclusion and Future Work

We have proposed a new approach to automatically generate query from code context and rank retrieval results, implemented it in a tool named Recommendflow and evaluated the method in the test data set. Our approach is based on

LDA model and experiment results on the test data set demonstrate that the LDA-based method outperforms the baseline.

We also have a plan to optimize our system and offer it to programmer community for free.

References

1. Ko, A.J., DeLine, R., Venolia G.: Information needs in collocated software development teams. In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society (2007)
2. Subramanian, S., Inozemtseva, L., Holmes, R.: Live API documentation. In: Proceedings of the 36th International Conference on Software Engineering. ACM (2014)
3. Brandt, J., Guo, P.J., Lewenstein, J., et al.: Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM (2009)
4. Ponzanelli, L., Bacchelli, A., Seahawk, L.M.: Stack overflow in the IDE. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press (2013)
5. Rahman, M.M., Surfclipse, R.C.K.: Context-aware meta-search in the IDE. In: 2014 IEEE International Conference on Software Maintenance and Evolution (2014)
6. Ponzanelli, L., Bavota, G., Di Penta, M., et al.: Mining stackoverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM (2014)
7. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)