Topic Matching Based Change Impact Analysis from Feature on User Interface of Mobile Apps

Qiwen Zou[1], Xiangping Chen[2,*], Yuan Huang[1,3]
[1]School of Information Science and Technology, Sun Yat-sen University, Guangzhou, China, 510006
[2]Institute of Advanced Technology, Sun Yat-sen University, Guangzhou, China, 510006
[3]Ocean University of China, Qingdao Haier Intelligent Home Appliance Technology Co.,Ltd, Qingdao, China
Email: cathyzqw@163.com, chenxp8@mail.sysu.edu.cn, huangyjn@gmail.com

***Abstract*—The complexity of mobile applications often lies in the user interface (UI). To update function provided by UI or just fix bugs related to UI, software maintainers primarily need to obtain the location of source code implementation and detect change set. Since UI related feature is tightly related to the class containing the declaration of the UI component, this paper proposes a topic matching based change impact analysis method from feature on user interface of mobile apps. Our approach combines LDA model with program dependency to realize the change impact analysis. Considering app′s small scale and few comments, a novel preprocessing method combining *tf-idf* with term weight based on structural information is applied to LDA model. Experiments on 16 update records of 4 open source apps show the effectiveness of our proposed method.**

I. Introduction

With the rapid development of mobile application industry, most mobile applications (i.e., apps) are updating frequently, which challenges software maintainers. During software maintenance, developers must spend much efforts on program comprehension when related documents are missing and even original developers are no longer available. However, to seek out relative classes manually is difficult and time-consuming. Change impact analysis [12] is always a special topic of determining potential consequences of a proposed change.

Different from traditional software, the user-friendly of apps has become a key point to attract users thus modifying user interface (UI) is frequent. In addition, app has to deal with users′ various requirements through UI and this can be error prone for UI with the complexity of the demands [1]. Thus we guess frequent function update is associated with app UI and it′s an active area of software maintenance tasks. To validate our conjecture, we manually browse 1007 update records of 306 apps collected from Google Play Store[1], in which 458 changes are related to function provided by UIs, excluding these changes such as data storage, configuration file and ambiguous changes (fix bugs, improve performance, etc.). The rate which reaches 45% shows a frequent modification of source code related to UI. Detailed data are available in the online appendix[2]. Further, we investigate keywords on UI and want to know how many words appearing on UI will exist in the topics. Results suggest an average of 35%, which indicates that function provided by UI can be well described by topics.

To update function provided by UI or just fix bugs related to UI, software maintainers primarily need to obtain the location of source code implementation and detect change set (i.e., classes that may be modified to accomplish an update of app UI). Feature location is always an option and then change impact analysis can help to find relative classes. A UI related software feature is tightly related to the class containing the declaration of the UI component providing its corresponding function. In this context, locating the feature on UI can automatically detect an initial class for impact analysis.

In recent years, text retrieval model such as Latent Dirichlet Allocation (LDA) [3] is generally used to locate feature while impact analysis depends on program dependency [9, 10, 12]. However, to do impact analysis for the update of feature provided by UI, those information implied by keywords on UI is very important, using LDA to mine relative classes and combining program dependency, we expect to obtain improved recommended list of classes. Considering app′s small scale and few comments, using entire source code corpus not just identifiers and comments is essential. If so, the words extracted may contain too much noise, extracting topic directly from source code may be less effective. Novel preprocessing techniques term frequency-inverse document frequency (*tf-idf*) and structural information based term weight can be applied to filter out less meaningful words and make topics prominent.

In this paper, we propose a topic matching based change impact analysis method for maintaining UI of apps. Our approach starts from locating software feature on UI to its implementation in source code as the initial class for impact analysis. Then, we combine LDA with program dependency to realize change impact analysis. For LDA, a novel preprocessing method combining *tf-idf* with term weight is applied.

We have conducted experiments on 16 update records of 4 open source apps to evaluate the effectiveness of our method, results show that our approach works well for recommending appropriate classes for corresponding feature on UI.

II. Related Work

A. Feature Location

Feature location, also called concept location, is a program comprehension phase during software maintenance to detect

[1]http://www.androidcentral.com/google-play-store
[2]http://research.defool.me/dataset/website/1.html
DOI reference number: 10.18293/SEKE2015-078

source code implementation of features of target system [2]. In recent years, most researches on feature location have focused on (semi-)automated techniques to alleviate manual operation. The most common analysis techniques are static, dynamic, textual, or a blend of several analysis approaches. Static analysis just uses source code text. Independently and in parallel, some other researches [6, 7] use dynamic information (i.e., execution trace) gathered from scenarios to locate features.

In particular, textual analysis based on modern information retrieval technique, LDA [3] and LSI [4], has been increasing popular. Marcus et al. [4] propose LSI-based feature location. Dynamic analysis combining with LSI (SITIR) results in a better performance comparing with LSI alone [7]. Lukins et al. [8] have evaluated LDA-based feature location. Experiments with Eclipse and Mozilla suggest LDA-based approach is more effective than using LSI for this task. An approach based on genetic algorithm is proposed by Annibale Panichella [10] to detect a near-optimal configuration for LDA which leads to a higher accuracy of feature location. Considering the structural characteristics of source code different from natural language, Blake Bassett [9] introduces a novel term weighting scheme for LDA to improve accuracy of feature location.

In our research of detecting relative classes for corresponding function provided by UI, feature location with characteristics of apps is used to find initial class for change impact analysis. Meanwhile, considering the effectiveness of LDA for detecting functional related classes in those previous works, we combine the information mined by LDA with program dependency to perform impact analysis.

B. Change Impact Analysis

Change impact analysis is used to determine the potential consequences of a proposed change during software maintenance [12]. In recent years, change impact analysis for software maintenance is hot. Acharya, M. et al. [11] design a static program slicing based method to do change impact analysis for large and evolving industrial software systems. Malcom Gethers et al. [13] configure a best-fit including information retrieval, dynamic analysis, and data mining of past source code commits to present an adaptive approach to perform impact analysis from a proposed change. Hoa Khanh Dam Dam [12] makes efforts on impact analysis for client-based systems. On the whole, there have various techniques supporting change impact analysis from procedural to object-oriented system [11, 12, 13, 14, 15].

Different from those researches, our method starts change impact analysis with a recommended class which is auto-located with our tool while traditional approach selects an initial class that needs to be changed by developers. In addition, we perform change impact analysis by combining linguistic information with structural dependency of source code.

III. Approach

A. Approach Overview

Fig. 1 shows the overview of our approach. In the general framework, users provide interface feature as a query and a recommended list related to the function of the UI is identified with which developers can easily detect relative classes.

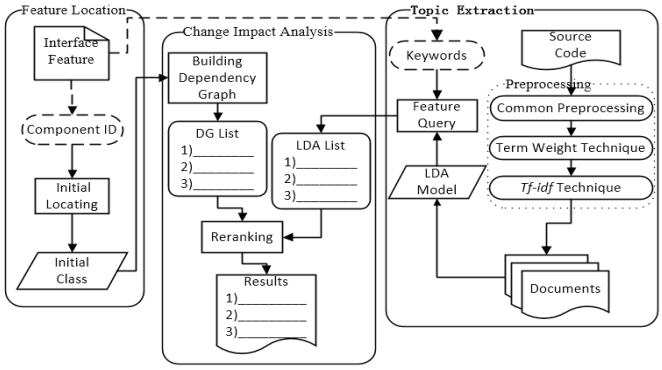


Fig. 1. The approach overview

Our approach starts from locating software feature on UI of mobile apps to its implementation class in source code by matching the ID of UI component.

For topic extraction, novel preprocessing techniques, *tf-idf* and term weight, are applied. Term weight technique based on the conjecture that the importance of words among different entities (e.g., classes, methods, attributes and others) are different and *tf-idf* technique stemming from information retrieval are applied to filter out less meaningful words and core words are taken as input for LDA.

Two class lists ranked based on dependency relation and topic matching degree are taken as input to rank its possibility of being impacted when maintaining the app feature on UI.

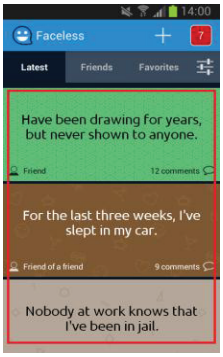
B. Feature Location

For change impact analysis, the primary step is locating the initial source code implementation of function provided by app UI, feature location is always an option.

Considering the special characteristics of apps, we model feature on UI of apps as $feature =< ID, keywords >$, in which ID is the unique identifier of UI component in the whole app project and keywords are core words on UI or close related to this component (i.e., words existing in the ID). When text information on UI is too long and indistinctive or there has no words on the corresponding component, words in ID are considered to build keywords. Keywords are used to construct query for LDA model described in next part. ID is used to locate the initial class declaring this component. Based on our previous work on searching UI component of apps [18], we develop an auto-locating tool[3] to find ID and detect the initial class by just clicking this component using a screenshot of the UI. This tool is developed based on the characteristics of apps that every component has unique ID and unique location on its belonging UI. When user is running an app and he wants to modify one component or function provided by the component on a UI, our tool can help to locate the initial location.

Fig. 2 represents examples of optimization of *message list* in project *Faceless* and *music scan* in project *Kjmusic*. The red rectangular box in Fig.2 (a) shows a message list in which

[3]http://research.defool.me/uidroid/



Keywords: message list
ID: R.id.listViewMessages

(a)

Keywords: scan
ID: R.id.scan_music_title

(b)

Fig. 2. Examples of components on UI of apps.

text is too long and words are less significant, we use core words in the id *R.id.listViewMessages* to build keywords. If we need to alter the display of message or add information on the list, we can click this part and locate the initial class. And in Fig.2 (b), we build keywords with text information (scan music) in the red rectangular box which is distinctive after being translated into English. We can click this part to obtain corresponding initial class and update the function of music scan.

C. Topic Extraction

The generation of topic model has following steps. Abstract Syntax Tree (AST[4]) can be used to extract the information of source code. Then, the documents are preprocessed. LDA outputs the word-topic probability distribution and the topic-document probability distribution. Thus, LDA model is constructed and can be queried with keywords of feature on UI.

1) Source Code Preprocessing: Common preprocessing steps include identifiers splitting, abbreviations expanding, removing stop words and stemming. In our method, we use novel preprocessing techniques, term weight and *tf-idf* to process source code corpus aiming at filtering out noise words and making topics prominent.

a) Term weight technique: This technique is proposed based on the experience that term in different entities (i.e. class, method etc.) has different importance by Girish Maskeri [16]. Considering the hidden but important information, term weight is taken into account to make important terms outstanding.

Empirically, a weight-based rule $f : Ttype \overset{\text{yields}}{\rightarrow} v$ that assigns various positive integers v to five types of terms (i.e., all types T={term in class names, term in method names, term in attribute names, term in comments, term in others}) is applied. To differentiate the importance, for example, $v(class)$ is assigned higher than $v(method)$ because in object-oriented software system, class as functional implementation of domain problem, it is more promising to acquire the intended functional knowledge encoded in the class name than method. And empirically, other values $v(comment)$, $v(attribute)$, $v(other)$ are assigned diminishingly. Then, we use formula

$$weight_{i,j} = \sum_{Ttype \in T} v(Ttype) \times n_{i,j} \tag{1}$$

to calculate weight sum of term i in document j. $n_{i,j}$ denotes the number of occurrences of i in the forms of $Ttype$ in document j. Further, we normalize term weight $weight_{i,j}$ to $\omega_{i,j}$ with the fomula

$$\omega_{i,j} = \frac{weight_{i,j}}{\sum\limits_{k \in D} weight_{k,j}} \tag{2}$$

because different documents have different size of vocabulary and $\omega_{i,j}$ reflects term′s importance to the document j.

b) tf-idf technique: This technique is used to evaluate the importance of a word to a document in corpus and has been widely used in the domain of information retrieval. In our study aiming at removing noise words, *tf-idf* is applied.

tw (term weight) denotes the proportion of word i in document j. $tf_{i,j}$ is the number of occurrences of word i in document j and m stands for the number of different words occurring in document j.

$$tw_{i,j} = \frac{tf_{i,j}}{\sum\limits_{k=1}^{m} tf_{k,j}} \tag{3}$$

tf-idf (term frequency-inverse document frequency) is proposed with the principle that the importance of word i to document j is in proportion to $tf_{i,j}$ while inversely proportional to the number of documents df_i containing the word i. And n is the number of all documents in the corpus.

$$tf - idf_{i,j} = tw_{i,j} \times \log \frac{n}{df_i} \tag{4}$$

Both in term weight and *tf-idf* technique, threshold needs to be set to filter out less meaningful words. A word with higher value is more representative of the document than others. We use cut points δ_{weight} and δ_{tf-idf}. Word i in the document j will be retained if $\omega_{i,j} > \delta_{weight}$ and $tf - idf_{i,j} > \delta_{tf-idf}$, if the weight $\omega_{i,j}$ or $tf - idf_{i,j}$ for a word is low, that means the word is not significant and can be considered as noise word.

2) Model Generation: Latent Dirichlet Allocation (LDA) is a probabilistic generation model from the term occurrences in a corpus, proposed by Blei et al [3]. Source code documents are taken as input for LDA, the documents are considered unstructured and described as bag-of-words in which the order of the words is neglected. Through training, LDA expects to obtain two matrix $\theta_d =< p_{t1}, p_{t2}, ..., p_{tk} >$ and $\varphi_t =< p_{w1}, p_{w2}, ..., p_{wn} >$. For each document d, p_{ti} denotes probability that d maps to topic t_i. For each topic t, p_{wi} denotes probability that word w_i belongs to t. For those results, documents having the same relevant topic are grouped into the same cluster.

In our approach, the classes of app source code are taken as a collection of documents. For that, we use term weight technique and *tf-idf* technique for preprocessing, appropriate thresholds should be set to choose the most likely words reflecting corresponding document, which lays a solid foundation for our purpose of change impact analysis. Therefore,

[4]c2.com/cgi/wiki?AbstractSyntaxTree

we invite three graduate students of Sun Sat-sen University to manually check the words retained by preprocessing with a large number of experiments through reduplicated adjusting and feedback, so that near-optimal thresholds can be obtained. To avoid any bias, students are not aware of the experimental goals.

We apply LDA to this entire collection of preprocessed data with Gibbs sampling. The number of Gibbs iterations n is required while every iteration samples a topic for each word. In addition, the Dirichlet hyperparameter for topic proportions ∂ and the Dirichlet hyperparameter for topic multinomials β need to be set to control the smoothing of the model. For those configuration parameters, we use suggested values from [17], $\partial = 0.5$, $\beta = 0.1$. Moreover, we set the number of topics and the number of top words in a topic by taking account of both app scale and the number of keywords on app UI.

3) Feature Query: As is illustrated in the section of introduction, we find that core words on UI will exist in topics. In that case, feature on UI can be matched with a topic and even mapped to source code classes. Having modeled feature on UI, we use keywords of feature as query, those words have been processed so that they can be matched with topic words. For each query, we compute the similarity of the feature and all topics and select the most similar topic using words matching (i.e., the topic is more relative when more words exist in both query and this topic). Then topic-document distribution is used to sort classes with descending order, we name it as LDA list.

D. Change Impact analysis

Having located the initial class, change impact analysis is used to detect recommended list. Dependency graph (actually a tree, the child B of a node A is its relative class, including two cases that A depends on B and B depends on A) starting with the initial class can be obtained. This dependency graph gives relative classes and dependency depth and can be constructed as a dependency list (DG list) in which the initial class is always in the first place, and the children of a node have no certain sequence. Source code class with smaller depth indicates it is more likely to be changed to adapt to the update. To keep the parent-child relationship, parent class is attached to every class in the DG list.

Finally, an improved recommended list will be generated by change impact analysis combining LDA list with DG list. Topics are functional description of source code, and dependency graph represents structural relations. However, in DG list, a class as a child node may have higher probability assigned by LDA than a class as parent node when parent class is used as an interface to this child class and doesn$'$t implement core function. In theory, the parent class is important for feature update. In that case, we reassign the probability P of every class A in the LDA list considering the probabilities of its all children $B_1, B_2, ..., B_n$, namely,

$$P(A) = max(P(A), P(B_1), P(B_2), ..., P(B_n)) \quad (5)$$

where n is the number of A$'$s all children. And the parent-child relationship can be detected in the DG list. In addition, the class in LDA list may be unreachable in DG list, in most cases, this class is not related to the function of the UI and can$'$t be impacted by the update, so we remove it from LDA list. Consequently, the order of classes in the topic is optimized and we obtain final recommended list.

TABLE I
THE PERCENTAGE OF KEYWORDS AND TOPIC WORDS

App project	Classes	Words on UI	Keywords	$N_t = N_i$	$N_t = 2N_i$	$N_t = 3N_i$
Lightning-Browser	21	131	74	24.32%	39.19%	39.19%
Notify	95	339	114	26.32%	35.09%	35.09%
Jamendo	115	70	52	21.15%	28.85%	28.85%
EasyToken	24	198	78	15.38%	24.36%	25.64%

IV. Experiments

We have conducted an empirical study to evaluate the effects of our method. In this section, we discuss the significance of change impact analysis from feature on app UI, as well as present and evaluate the resulting data.

A. Research Motivation

The key point of our research is based on the following question:

Is the change impact analysis from feature on app UI meaningful and promising?

We extract words on app UI, the data are simply preprocessed and we remove repeated words. Meanwhile, we use LDA to extract topics, composed of a couple of words, from source code corpus. We expect to validate that the function of UI can be well described by topics. Representative results are shown in Table I. The percentage of how many words existing on UI are in topics lists in the five column when the number of topic words N_t equals the number of keywords N_i, and when N_t is double of N_i the percentage lists in the six column, and so on.

From Table I, we can see that core words on UI will appear in topics, the large percentage is near 40% when the rate is double and it is lower when $N_t = N_i$. However, many experiments show no larger percentage when we continue increasing the number of the topic words. In that case, we go deep in the projects to check those unmatched words, such as *please, sure, thanks* for *EasyToken*. Obviously, those words are less significant for this app comparing with these words such as *easy token*. Therefore, we can conclude that those unmatched words are not core words, if appropriate number of topics and number of top words in a topic are set, the query using keywords on UI can find matched topic, and then find corresponding classes with different probability. For more data, this online appendix[5] is avaliable.

B. Data Set & Effectiveness Measure

Our approach is used to recommend a list of classes which are probably affected by maintaining a software feature on UI. We evaluate our method with update history of apps to see the position of changed classes related to an update of UI in the list. As a result, we choose open source apps with well-written source code and available update records.

[5]http://research.defool.me/dataset/website/2.html

TABLE II
THE EXPERIMENT DATA

App project	Change	Update Date	All Classes	Update Classes	Change Description	Feature-keywords
Oschina	01	2014-02-19	162	9	Fixes the function of report message	report message
	02	2014-02-24	162	7	Added welcome screen to start different figure with different festivals	start welcome
	03	2014-02-10	162	1	Fixes flashing with tweet audio player	audio player tweet
	04	2014-03-03	162	5	Fixes the keyboard up when refreshing detail message	detail editor
	05	2012-09-14	162	2	Fixes a bug that users cant use the camera to upload new image	user image editor
Kjmusic	06	2014-01-29	41	5	Optimization of scan interface	scan
	07	2014-01-28	41	1	Repair the bug that current play pictures is hidden after disappearing	player picture
	08	2014-01-28	41	2	Optimization of lyrics playlist UI display interface	lrc
	09	2014-01-15	41	4	Repair logic error of playing a looping pattern	loop mode play
Faceless	10	2014-12-09	30	5	Display approximate distance to message author on Android	message list
	11	2014-12-09	30	2	Added location input when composing message in Android	location input
	12	2014-12-11	30	5	Added 'Nearby' messages feature on Android	nearby message
	13	2014-12-15	30	1	Hide secondary options in message compose window by default	advance option expand message
Jamendo	14	2012-09-13	114	1	Fixes preset naming	preset equalizer
	15	2012-07-05	114	2	User can customizer equalization	customizer equalization
	16	2011-04-14	103	9	Added paginated retrieval of all Album's tracks	album track

We choose 4 open source apps in our experiment: Oschina[6], Kjmusic[7], Faceless[8], Jamendo[9]. Oschina is an open source china community for sharing open source software. Faceless is an anonymous social software where you can talk freely. Kjmusic and Jamendo are two music players. The words on UI of Oschina and Kjmusic are Chinese, we use a translator to translate them into English during feature location.

Table II summarizes app information that we use to conduct the experiments, including update date, the number of classes and update classes, change description and feature-keywords. We use a number to denote a change in our paper.

To evaluate the performance of our method, we use the effectiveness measure in [9] to evaluate our results. Descriptive statistics is to go deep in the ranked list to check the position of the updated classes including min, median, max. And min, median, max represents the rank of first class, middle class, last class that are related to the update, respectively.

In addition, we change mean reciprocal rank (MRRC) as the average of the reciprocal of the location of relevant classes:

$$MRRC = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{r_i} \tag{6}$$

where C are all classes related to an update, and r_i is the rank of the relevant class in the recommended list. A higher MRRC implies a better rusults.

C. Results and Analysis

In this section we represent the results of topic matching based change impact analysis method from feature on UI for four apps. For the reason that it's firstly proposed, we use traditional LDA results without considering term weight and *tf-idf* (LDA), LDA list (CLDA) and DG list (DG) to compare with and discuss the advantages of our method.

[6]http://git.oschina.net/oschina/android-app
[7]http://git.oschina.net/kymjs/KJmusic
[8]https://github.com/delight-im/Faceless
[9]https://www.jamendo.com/en/

1) MRRC: Fig. 3 shows MRRC for 16 changes described in Table II. Compared with LDA, CLDA and DG, our method generally obtains higher MRRC than others which implies a better performance. And obviously, LDA with novel preprocessing obtains higher accuracy than traditional LDA. Sometimes, our method seems poorer than DG when doing minor changes, actually they are the same because the probabilities of the first few classes are the same such as change 15, the probability of the first class is the same as that of the second class. In addition, it's obvious that there have break points in the DG line for change 03, 07, 13, 14 because we don't show MRRC of DG when there is only one changed class. For that, feature location can always detect the initial class and the comparison is less promising.

2) Descriptive Statistics: This part we report the descriptive statistics for 16 changes of our method compared with LDA, CLDA and DG in Table III.

The character "-" in the table denotes meaningless results as explained in MRRC. We note a better performance (i.e., lower rank) of our method compared with LDA, CLDA and DG. In addition, the results of traditional LDA show that topics are scattered and its accuracy is lower than CLDA. Our method takes advantage of both DG and CLDA so that min is almost 1 (some are not because the probabilities of the first

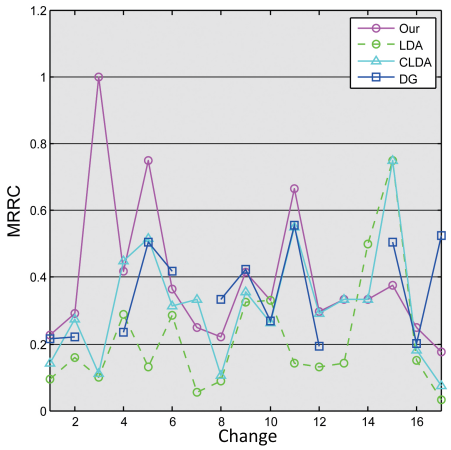


Fig. 3. MRRC of our method compared with LDA, CLDA and DG

TABLE III
DESCRIPTIVE STATISTICS OF OUR METHOD COMPARED WITH LDA, CLDA, DG

Change	Our Method			LDA			CLDA			DG		
	Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
01	1	7	105	2	76	162	2	106	161	1	22	94
02	1	5	70	1	106	152	1	16	162	1	13	60
03	1	1	1	10	10	10	9	9	9	-	-	-
04	1	4	7	2	4	6	1	3	6	1	23	25
05	1	1.5	2	4	46	84	1	15	30	1	37.5	75
06	1	8	11	1	9	39	1	6	40	1	3	18
07	4	4	4	18	18	18	3	3	3	-	-	-
08	3	6	9	8	14	20	7	10.5	14	2	4	6
09	1	5	30	1	20	40	1	9	39	1	3.5	9
10	1	5	17	1	4	29	1	24	30	1	12	19
11	1	2	3	4	16.5	29	1	5.5	10	1	5	9
12	1	7	15	3	14	29	1	11	26	2	9	19
13	3	3	3	7	7	7	3	3	3	-	-	-
14	3	3	3	2	2	2	3	3	3	-	-	-
15	2	3	4	1	1.5	2	1	1.5	2	1	40	79
16	1	9	90	1	31	115	1	36	92	1	13	24

several classes are the same) and most of the classes that are updated are in lower rank. However, this combination may make some results (larger rank) worse when CLDA and DG interact together. For change 02, our method obtains lower rank except max because the max of CLDA is large that affects our recommended list. For that, Fig. 3 shows overall results that the effect is little and our method is effective for change impact analysis from feature on app UI.

V. Conclusion and Future Work

In this paper we propose topic matching based change impact analysis from feature on UI of mobile apps. Focusing on the function of app provided by UI, we firstly model it with keywords and ID and help users find appropriate classes related to the update of the function.

In our method, we develop a tool to locate the initial location by just clicking this component using a screenshot of the UI. Then, we use LDA to model source code in which novel preprocessing techniques (i.e., term weight, *tf-idf*) are applied and keywords related to UI are used as query to acquire proper class list with descending probability. Finally, dependency graph (DG) starting with initial class is detected, we take the advantage of DG and LDA with novel preprocessing techniques to do impact analysis, experiments with four apps show a better performance of our method.

In future work we plan to investigate change impact analysis from feature on UI at method level. In addition, we find some updates related to UI just modify the layout file (i.e., .xml), how to discover those files is also meaningful.

Acknowledgment

This research is supported by NSFC-Guangdong Joint Fund (No. U1201252), the Educational Commission of Guangdong Province (No. 2013CXZDB001), the Fundamental Research Funds for the Central Universities, and the National Science & Technology Pillar Program (No. 2012BAH12F02).

References

[1] Zhifang Liu, Xiaopeng Gao and Xiang Long, *Adaptive random testing of mobile application*, in Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET 10), IEEE Computer Society, Washington, DC, USA, 2, 297-301.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, *Feature location in source code: a taxonomy and survey*, Journal of Software: Evolution and Process, vol. 25, pp. 53-95, 2013.

[3] Blei, D. M., Ng, A. Y., Jordan M I, and Jordan, M. I. , *Latent Dirichlet Allocation*, Journal of Machine Learning Research, vol. 3, pp. 993-1022, 2003.

[4] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., *Indexing by Latent Semantic Analysis*, Journal of the American Society for Information Science, vol. 41, pp. 391-407,1990.

[5] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, *An information retrieval approach to concept location in source code*, in Proc. of the11th Working Conf. on Reverse Engineering, 2004, pp. 214C223.

[6] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R., *A systematic survey of program comprehension through dynamic analysis*, Software Engineering, IEEE Transactions on, 2009, 35(5): 684-702.

[7] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, *Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval*, IEEE Transactions on Software Engineering, vol. 33, no. 6, pp. 420C432, Jun. 2007.

[8] S. Lukins, N. Kraft, and L. Etzkorn, *Source code retrieval for bug localization using latent Dirichlet allocation*, in Proc. of the 15th Working Conf. on Reverse Engineering, 2008.

[9] B. Bassett and N. A. Kraft, *Structural information based term weighting in text retrieval for feature location*, in IEEE Int'l. Conf. on Program Comprehension, 2013, pp. 133-141.

[10] Panichella A, Dit B, Oliveto R, et al., *How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms*, in ICSE, 2013, pp. 522C531.

[11] Acharya, M., Robinson, B, *Practical change impact analysis based on static program slicing for industrial software systems*, in Proceedings of the 33rd international conference on software engineering. ACM, 2011.

[12] Hoa Khanh Dam Dam, *Automated change impact analysis for agent systems*, in Software Maintenance (ICSM), 2011 27th IEEE International Conference on (pp. 33-42). IEEE.

[13] M. Gethers, H. H. Kagdi, B. Dit, and D. Poshyvanyk, *An adaptive approach to impact analysis from change requests to source code*, in ASE, 2011, pp. 540C543.

[14] Yi Wang, Jian Yang, Weiliang Zhao, *Change impact analysis for service based business processes*, IBM Systems Journal, 2005, 44(4): 653-668.

[15] Kama, N., Azli, F, *A change impact analysis approach for the software development phase*, in Software Engineering Conference (APSEC), 2012 19th Asia-Pacific (Vol. 1, pp. 583-592). IEEE.

[16] Maskeri, Girish, Santonu Sarkar, Kenneth Heafield. *Mining business topics in source code using latent dirichlet allocation*, in Proceedings of the 1st India software engineering conference. ACM, 2008.

[17] Biggers L R, Bocovich C, Capshaw R, et al., *Configuring latent Dirichlet allocation based feature location*, Empirical Software Engineering, 2012.

[18] Kaiyuan Li, Zhensheng Xu, Xiangping Chen, *A platform for searching UI component of android application*, in ICDH 2014, Nov. 28-30, 2014, Guangzhou, P. R. China.