Duplicate Issue Detection for the Android Open Source Project

Kasthuri Jayarajah, Meera Radhakrishnan, Camellia Zakaria
School of Information Systems, Singapore Management University, Singapore
{kasthurij.2014, meeralakshm.2014, ncamelliaz.2014}@phdis.smu.edu.sg

ABSTRACT

The Android Open Source Project(AOSP) has seen tremendous traction over the past decade, and as such, the bug repository is growing in scale. With this growth, the effort required for project members to triage incoming new reports to identify whether it is a duplicate issue that has already been addressed, or receiving attention, is also on the rise. In this work, we create dataset of issues from the Android issue tracker, and use standard IR techniques such as VSM and LDA to understand their capability in such similar issue retrieval. Further, we combine VSM and LDA to evaluate its usefulness. We find that, overall, VSM performs better with this dataset.

CCS Concepts

•**Information systems** → *Data mining;* •**Software and its engineering** → *Software post-development issues;*

Keywords

Duplicate Issue Detection; Android Open Source Project; VSM; LDA

1. INTRODUCTION

The Android OS has steadily gained popularity, owing much to its open source nature to enable OS support for multiple device manufacturers. Moreover, its low barrier to entry into the market place allows for more independent developers to sell their mobile applications. According to a report from IDC in 2015 [2], the market share of Android was roughly 83%, with its closest competitor, Apple iOS, only securing roughly 14% of the market share. Catering to such mass, the Android Open Source Project (AOSP) has a web portal to allow developers and users report issues and bugs related to the OS. As with most discussion forums and bug trackers, the Android Open Source Platform issue tracker is also disadvantaged by multiple users reporting the same issues. This has implications for both stakeholders: issue reporters (users) and Android project members to whom the issues are assigned. Users who encounter legitimate issues will go over the history of similar issues experienced by other users and members, which typically follows with a solution. On the other hand, new duplicated issues submitted by project member are automatically tagged to allow for a continuous stream of solution.

We seek to improve accessibility of such resources for both stakeholders. First, users can expect to find a list of similar issues logged by different users. These documents will be ranked based on relevance or order of similarity to the original queried issue. Our approach filters duplicates and retrieves original documents related to an issue. We make the following **key contributions:**

1. By systematically scraping the Android Issue Tracker and the Android API documentation, we make available crawlers for extracting issues and comments from the official Issue Tracker, and packages and classes information from the Android API documentation to the research community. In addition, we also make available a subset of the extracted data publicly. Unlike popular bug repositories like Mozilla [1] and software help forums such as StackOverFlow [2], bug reports for the AOSP has not been studied previously.

2. We provide quantitative insights on how well existing information retrieval techniques perform on the issues dataset comprising of all available issues between November 2007 and March 2015 for the AOSP.

3. We evaluate the state-of-the-art IR techniques such as VSM and LDA in duplicate issue detection for Android issues and report our findings. Overall, we find that vector space representations perform better than topic modeling. Further, we propose a technique to combine both VSM and LDA leveraging the Android API dataset.

2. RELATED WORK

Much research has been dedicated to automating the detection of duplicated bug reports. Nilambari et al. surveyed several methodologies and found two primary approaches for duplicate detection. New issues are either categorized as duplicates or ranked based on feature similarity of certain features [7].

Sun et al. used a Support Vector Machine (SVM) trained model to classify new bug reports into duplicates class or other class [10] and extended the work to using BM25F, which considered textual data like summary and description, to differentiate the degrees of importance for a bug report [9]. Similarly, AnhTuan et al. combined topic modeling technique, Latent Dirichlet Allocation (LDA)

[1]https://bugzilla.mozilla.org/

[2]https://archive.org/details/stackexchange

SoftwareMining'16, September 3, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-4511-8/16/09...$15.00
http://dx.doi.org/10.1145/2975961.2975965

and textual based BM25F similarity techniques for duplicate detection on datasets from Open Office, Mozilla and Eclipse [6]. Their approach achieved an accuracy that is higher than any existing techniques for duplicate detection. But this may not work well with very large amount of data, as the recall would go low.

In other extended work, Alipour et al. included contextual features such as software architectural words and nonfunctional requirement words to improve bug report de-duplication methods [4]. Jalbert et al. poses the problem of detecting duplicate reports as a linear regression task [5]. Low efficiency is the main drawback of these machine-learning techniques. A recent work [3] followed the duplication bug report detection method proposed by Alipour et.al and extended on it to partially automate the extraction of the contextual features. They evaluated their technique on datasets from years 2007 to 2012 of Open Office, Mozilla, Eclipse and Android, with the performance of their method being slightly worse on Android dataset.

The work we present supports the effort to build a duplicate-free bug reporting system for Android Open Source Project (AOSP), as how open source bug repositories have been addressed. By adopting VSM and LDA on the AOSP dataset, we believe that it can help reduce the triaging efforts of users seeking information and developers fixing bugs related to Android.

3. PROBLEM DEFINITION

In this section, we formally introduce our problem. The duplicate issue detection problem can be approached as an automatic selection of relevant issue (*Definition 1*), or semi-automatic solution where a set of relevant issues are retrieved by the detector and human intervention is followed to pick the most relevant issue (*Definition 2*). We define the following:

Definition 1: Given a set of issues or bug reports, R, and a new incoming issue or report q, find $r \in R$ for which q is similar to r.

Definition 2: Given a set of issues or bug reports, R, and a new incoming issue or report q, find $r_1, r_2, ..., r_k \subseteq R$ which are the top-k issues similar to q.

In this work, we take the semi-automatic approach as defined in *Definition 2*.

4. DATASET

In this work, we consider two sources of data. The primary source of data is the Issue Tracker of the Android Open Source Project [3] which is an online portal that allows developers and programmers to report bugs and issues with the Android platform. The secondary source of data is the online documentation of the Android platform which provides descriptions of the packages and classes included in the platform [4].

The issues available on the Issue Tracker are not available as a data dump for download. Hence, we wrote a crawler script using the Beautiful Soup (a Python library) to scrape the web portal to scrape off the issue topic, description, final status, stars, associated comments, etc. For the period between November, 2007 and March, 2015, a total of 134,567 issues were scraped. Among them, for 4190 of the issues with status "Duplicate", the issue IDs of the related original issues were also scraped. These $(duplicate, original)$ pairs form our ground truth for evaluation purposes (see Section 7). Henceforth, we call the set of "Duplicate" issues as *queries*, Q. For each query $q \in Q$, the associated original issue is $g \in G$, where G is the set of original issues who have a linked duplicate issue.

[3]https://code.google.com/p/android/issues/list

[4]http://developer.android.com/reference/packages.html

Similarly, the Android documentation is also not available online for direct downloads. We wrote two crawler scripts to, one for extracting package information, and the other for extracting class information. For each of the 236 packages, we scraped the package summary (if available) and the classes associated with the package. For each of the 4058 classes, we extracted the class summary, methods, constants and any other description available. In Section 6, we describe in detail how this secondary data is used.

We make available the code for the crawlers and a subset of our data consisting of 3500+ issues (and 47,000 related comments) for public access[5].

5. PREPROCESSING

In this section, we describe the various steps that we performed for pre-processing our primary dataset from AOSP as well as the secondary dataset from Android API.

5.1 Text and Code Separation

A filed issue/query typically consists of text, code and traces. While every piece of textual content is important, standardizing all inputs is a good approach when developing a search recommendation system. This ensures consistency and allows for simpler preprocessing requirements. In separating these entries, we identify similar structures for each type of content. A simple Java program was written to extract them in the order of these steps as follow.

5.1.1 Separate Traces

Android follows specific patterns for its logs in that it consists of keywords such as "D/ - Debug", "V/ - Verbose", "W/ - Warn", "I/ - Info", "E/ - Error". This represents the system component from which the error originates and will be reflected when printing messages using the Logcat class. Accordingly, the program singles out lines based on regular expressions that represent this format.

5.1.2 Separate Code

Similarly using regular expressions, the program extracts code and comments from the remaining content. A straightforward application of regex like "$/\ast([^*]|[\backslash r\backslash n]|(\ast+([^*/]|[\backslash r\backslash n])))\ast\backslash\ast+/$", is used to disassociate comments. Further, a number of rules were set to identify code. This includes (1) validation of code blocks (2) lines containing "=" and endsWith ";", (3) lines that contain characters such as "public String", "private int" and many more. A single regular expression pattern may not always achieve an intended result. In many of these cases, a code is extracted based on a combination of regular expression in a simple parser.

After separating out code and traces, we were essentially left with content, categorized as "Text".

5.2 Data Cleaning

After separating out text and code from the issue description, we considered the issue topic and the textual description for further analyses. As a result of scraping using html tags, the issue descriptions were left with URLs and ad-hoc html markups for certain issues. These were removed from the data using Regex patterns using Java. In addition, removed any remaining punctuation marks before proceeding.

5.3 Spam Removal

From our preliminary analysis of the dataset, we noted that there are many issues that are spam or has irrelevant content. We cleaned our dataset by removing these spam issues. This was done in two

[5]https://github.com/oscar4arr/AOSP-DuplicateIssueDetection

Table 1: Examples of word stemming.

Sample words	Stem
“discovery", “discovering", “discoverable"	discoveri
“device", “devices"	devic

ways: (1) the issues already marked in AOSP as Spam was removed, (2) we created a bag of words from the commonly occurring words in spam contents and removed all the issues containing those words in either the issue topic or issue detailed description. For example, “black magic”, “astrology”, “marriage specialist” are some of the words that frequently occurred in spams. Among the total issues, 22% were spam contents.

5.4 Tokenization

As described previously, we evaluate multiple techniques for retrieving similar documents. For LDA, MALLET [1] provides a command line option that handles tokenization. However, for our own implementation of VSM, we needed a tokenizer. To standardize the *bag of words* across the techniques (to make sure all techniques worked using the same bag of words), we used an intermediate representation of the MALLET output to convert each document into a *bag of words*. To this end, we used the Java API provided by MALLET. In particular, after reading in every issue using the *CSVIterator* object and passing through the relevant instance pipes, we use the *Alphabet* object to retrieve the tokenized words present in the issue.

5.5 Stemming

One key observation from the issues were that they contain natural language in that a word could be expressed in many forms. For example, in describing an issue regarding Bluetooth connectivity, we observed texts containing multiple words such as “connect", “connecting", “connectivity", etc. all of which stem from the same root “connect". Hence, we used the stemming API from Snowball[6] to stem the tokenized words. Table 1 lists some common examples from the Android issues.

5.6 Stopword Removal

Initially, we used the standard English stop words provided by MALLET for removing stop words. However, we observed that the list of words were too generic. To this end, we tried to identify the most common words across the set of issues, and adding them to the list of stop words. We used the Inverse Document Frequency (*IDF*) of the words to quantify the *rarity* or *information value* of a word where the *IDF* of a word w is defined as $IDF_w = log(N/n_w)$. Here, N is the total number of words and n_w is the number of documents the word w appeared in. In Figure 1, we plot the CDF of the IDF of all words in the issue set. The x-axis is the logarithmic *IDF* and the y-axis is the cumulative frequency. In Algorithm 1, we outline the steps we took to identify the threshold *IDF* (below which are “common" words added to the existing stop words list). By varying the threshold systematically, we increasingly add more words to the stop word list and observe the resulting recall by running the dataset through VSM (explained in Section 6). We finalize the threshold value as 4 by picking the value that maximizes the recall (See Section 7).

5.7 Frequent Sentences Removal

In AOSP tracker, the issues are mainly classified into 14 types – for example, ‘User bug report’, ‘Developer bug report’, ‘Tools bug report’, ‘Android Studio bug’ and ‘Jack bug report’. Each of these issues has a template with a default issue description. We consider these issue descriptions as frequent sentences occurring in most of the issues. We remove these terms from the actual description of issues for our processing.

[6]http://snowball.tartarus.org/

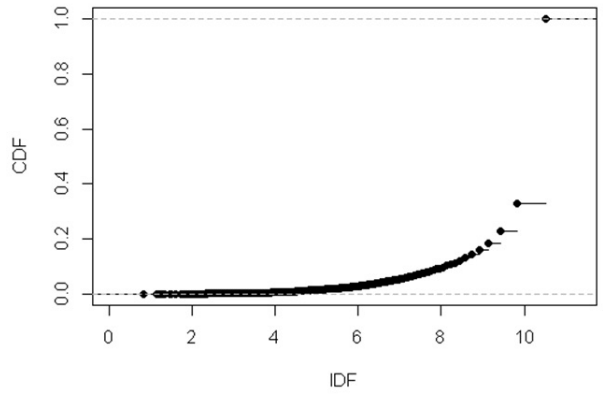

Figure 1: IDF of all words in the dataset.

Algorithm 1 Stop Words Expansion algorithm

for $i = 1$ to 10 **do**
  Step 1:
    Select words w with $idf \leq i$ to set W
    Add W to the stop words list
  Step 2:
    Run VSM and choose top-k similar issues for the query set Q
    Calculate and store recall
end for
Choose i with the highest recall and retain W_i as the expanded stop words list

6. APPROACH

In this section we describe the different approaches that we followed for duplicate bug report detection from the Android Open Source Tracker. Figure 2 shows our overall methodology and steps involved.

6.1 Vector Space Model (VSM)

Vector space model (VSM) [8] is a technique that has been extensively studied and applied in information retrieval over the years. Fundamentally, text information is made up of documents, in this case are issues, and terms, which are processed words in each document. In its entirety, Natural Language processing struggles with two problems, namely, Polysemy[7] and Synonymy[8].

Accordingly, this technique does not rely on natural language processing, but rather, on term occurrences in a vector. VSM models the documents (issues) and queries as sets of terms. Each term will be individually assigned to a weight. This allows semantic content to be retained as much as possible. Then, VSM computes

[7]Polysemy is having the same word occur in document with different meanings. For example, “ant” could refer to an insect or the ANT wireless communication protocol.

[8]Synonymy is having different words to describe the same thing. For example, “bug”, “issue”, and “problem” can all be used to describe an error

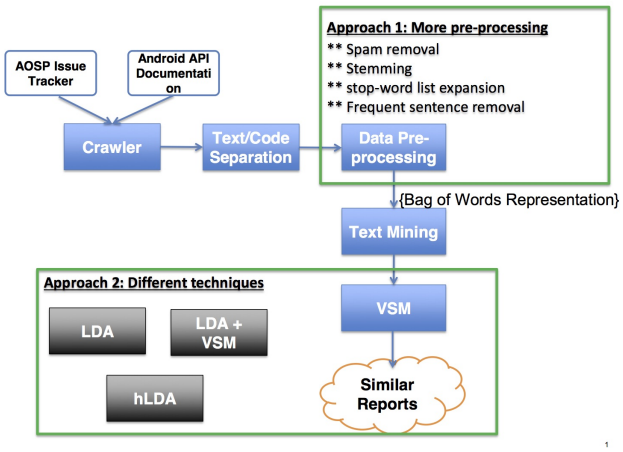

Figure 2: Approach

Generating the model

1. bin\mallet import-dir --input C:\mallet\issues\ --output issues_topics.mallet --keep-sequence --remove-stopwords

Training the model

2. bin\mallet train-topics --input issues_topics_sample.mallet --num-topics 120 --inferencer-filename inference_issues.mallet --optimize-interval 20 --output-state issues_topics-state_new.gz --output-topic-keys issues_topics_keys.txt --output-doc-topics issues_topics.txt

Inferring Topics

3. bin\mallet import-dir --input C:\mallet\queries\ --keep-sequence --output queries.mallet --use-pipe-from issues_topics_sample.mallet

4. bin\mallet infer-topics --input queries.mallet --inferencer inferencer_issues.mallet --output-doc-topics queries_inferred_topics.csv --num-iterations 100

Figure 3: MALLET Commands for Topic modeling

the cosine distance measure between document and query vectors. This cosine value represents the relative influence of terms in each document with a query from a scale of 0 to 1; 1 being identical, otherwise their values differ.

There are many ways to derive weights mentioned in the IR literature. For this project, we used the TF-IDF weighting scheme. TF-IDF is widely preferred as it typically achieves higher precision-recall compared to other weighting approaches.The following describes the steps taken to calculate the cosine distance measure.

6.1.1 Term-Frequency (TF)

For each document, we first calculated the term frequency of each term in a document. Additionally, a global record of the term frequency for all terms in the collection of documents is kept for calculating the cosine similarity.

6.1.2 Term-Frequency Inverse-Document-Frequency (TF-IDF)

The scarcity of a term across the entire document collection is measured to be important. The inverse-document frequency is described to favor this. Additionally, we considered length normalization for the documents. Longer issues are more likely to have higher term frequencies. This increases the probability of a document matching a query. By normalizing the length, this gives equal chances for both long and short issues to be picked. IDF weights of terms (in natural log) is calculated as follows:

$$TF_t = n/N$$

where n = Occurrence count of a term and N = Total number of terms in the document

6.1.3 Cosine Similarity

To calculate the cosine similarity, we used the following equation. Finally, the cosine similarity is sorted in descending order to retrieve only the top 20 relevant documents.

$$similarity(x,y) = cos(\theta) = (x.y)/(|(|x|)| * ||y||)$$

where x = query vector and y = document vector

VSM is used as our baseline method for comparing precision and recall with other approaches.

6.2 Latent Dirichlet Allocation (LDA)

To mine topics for the corpus set of issues that we obtained from the Android Open Source project (AOSP) we chose the latent Dirichlet allocation model and used its open source implementation available in Java with the tool called *Mallet* ("MAchine Learning for LanguagE Toolkit") [1]. LDA characterizes each topic it generates as a probability distribution over the main keywords that the topic is related to and is referred to as word distribution over a topic. A probability distribution over topics for each of the document is also generated as each document can refer to several topics.This is called as the topic distribution or topic proportions. The number of topics to be mined is a model parameter and determined based on the dataset and its relevance to different topics. LDA also has certain other model parameters called Dirichlet priors α, β, φ which are generally obtained by a technique called Gibbs Sampling.

In our work, the bag of words obtained from the set of issues after extensive preprocessing as explained in previous section is used to generate the topic model using LDA. We divided the AOSP dataset into two: (i) original set of issues and (ii) set of issues that are already marked as 'Duplicates' to any of the original issue. In the following, we use the generic term 'document(s)' to refer to set of original issues and 'query' to refer to the set of duplicate issues. The steps involved in mining topics is as follows:

6.2.1 Generating the Model

A set of feature instances or feature vectors are generated as a MALLET file by taking in the corpus bag of words from each of the documents as input. We use the above command to generate our initial topic model.

In the command for topic model generation (See Figure 3 Command 1), we give the path to the input directory, which has number of files relating to our set of issues. Each one of these files has the bag of words related to the particular issue which we generate after all our pre-processing steps. Therefore, the entire folder can be considered to be a corpus of data. We use MALLET to convert this corpus of bag of words to single MALLET format file *issues_topics.mallet* to generate the topics. Using the import command all the files from input directory are imported and then transformed into MALLET file, which is our model based on the set of issues. The 'keep-sequence' command preserves text in order of their original appearance. With the 'remove-stopwords' command, frequent or common words found in the default English stop-words dictionary are stripped out.

6.2.2 Training the Model

The training instances from the model generated is used to train the topics. We fixed the number of topics to be 120 based on the number of Android packages with summary described, obtained from our secondary dataset of Android API. A topic inferencer is created which can be later used to infer topics from an existing model for new set of data. We did hyper parameter optimization of the Dirichlet prior α by setting the optimization interval to 20. Therefore, LDA will automatically adjust the parameters during optimization so as to improve the accuracy of the model. Upon training, three main outputs are generated. The first one is a 'word-topic' file with the corpus of words in all set of documents and the topics it belongs to. The second file is the 'topic_keywords' file which lists top keywords for each of the 120 topics generated and last one is the 'topic-proportion' file which includes breakdown, by percentage, of each topic within each original document file.

The command used for model training (See Figure 3 Command 2) takes as input the generated *issues_topics.mallet* file, and runs the topic model routine using the 'train-topics' command. It trains the MALLET to find 120 topics. Every word from the corpus of data is related to the topics that it belongs to and is outputted as a compressed file *issues_topics_state_new.gz*. We obtain the top keywords for each of the 120 topics in the output file *issues_topics_keys.txt* using the 'output_topic_keys' command. The *issues_topics.txt* file has the topic proportions for each topic within each original issue file. A topic inferencer file is created using the 'inferencer-filename' command so as to use during the inference of topics for new set of data.

6.2.3 Inferring Topics

In this step, we use MALLET to infer topics for the set of duplicate issues by using the already trained model. The output obtained is a file with topics inferred for each of the queries and also a topic proportion within the query set. For each query, the topics are ranked in descending order of their topic proportion showing the most relevant topic first.

In this first step of topic inference (Figure 3 Command 3), we take the files in queries directory as input (which is the set of duplicate issues), we generate a new MALLET topic model. To make sure that the new model created is compatible with the trained model, we used the 'use-pipe' option to specify our initial trained model on the set of issues.

For inferring topics for the set of duplicate issues, the queries.mallet model and the inference file are taken as input and topics are inferred using 'infer-topics command (See Figure 3 Command 4). The output obtained is a file with set of topics and their breakdown by percentage within each of the topics inferred for the duplicate issues.

6.3 Combination of LDA and VSM

We extended the baseline VSM approach by combining it with the LDA for Android API. We first trained a model using LDA by taking the Android API package summary as the input. We obtained a new set of topics and top keywords based on the API package summary and also distribution of each of the topic.

Recall that the VSM approach calculates the relatedness of a document to the query based on the frequency of terms. In this approach, we modified VSM to instead measure similarity based on Android topics, using words in the same topics as synonyms. For example, the topic, "android.bluetooth", consists of a different set of synonyms compared to topic, "android.appwidget" (See Table 2).

Table 2: Topics from LDA

android.bluetooth	android.appwidget
Bluetooth	View
devic	handl
transfer	screen
ble	draw
energy	design
low	screen
gatt	dynamiclayout
socket	appwidgethost
connect	proxim

Table 3: Topics from LDA

Topic X	Topic Y
voic, doesn, messag, googl, latest instal, contact, messag, text, rout, applic, behaviour, applic, told, friend, peopl, confus	send, messag, google, voic, crash, program, latest

In doing so, we would need to consider the 'value proportion' of a document and a query to a topic.The value proportion of topics is based on our second approach, LDA.

7. EVALUATION

In this section, we introduce the performance metrics used in our evaluation and summarize our findings from the three techniques.

7.1 Performance Metrics

As briefed in Section 4, the set of queries, Q, is the set of issues with status "Duplicate". The corresponding one-to-one set of original issues is the set G. We define *Recall* in two ways: (1) based on the exact match of the similar issues retrieved by the techniques against the set G, and (2) based on manually sampling a subset of the issues (roughly 10%), and marking whether the most similar issue retrieved is similar in content with the query even though it is not an exact match from G.

We formalize the two metrics in the following manner.

For (1), where the set of top-k similar issues retrieved is, R_k, then

$$Recall_k = \frac{|R_k \cap G|}{|Q|}$$

In the case of manual sampling (2), where the set of most similar issues (top-1) retrieved for each $q \in Q$ is S, and the subset $S' \subseteq S$ is relevant to Q, then,

$$Recall = \frac{|S'|}{|Q|}$$

Further, in the case of (1) (exact matching), we also measure the Mean Average Precision over the top-k ranked retrieval.

7.2 Results

7.2.1 Pre-processing

In this section, using the example of VSM, we describe the improvement we see in recall, for major improvements in pre-processing.

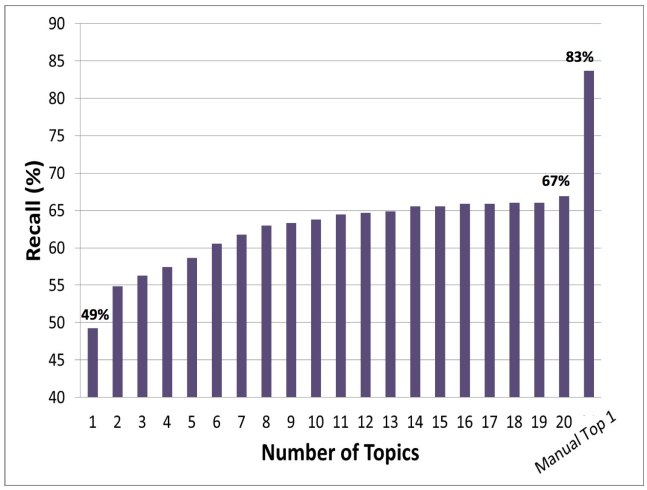

Figure 4: Recall for VSM.

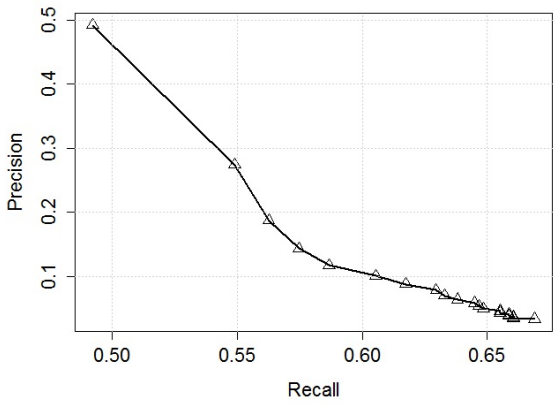

Figure 5: Precision-Recall Curve for VSM.

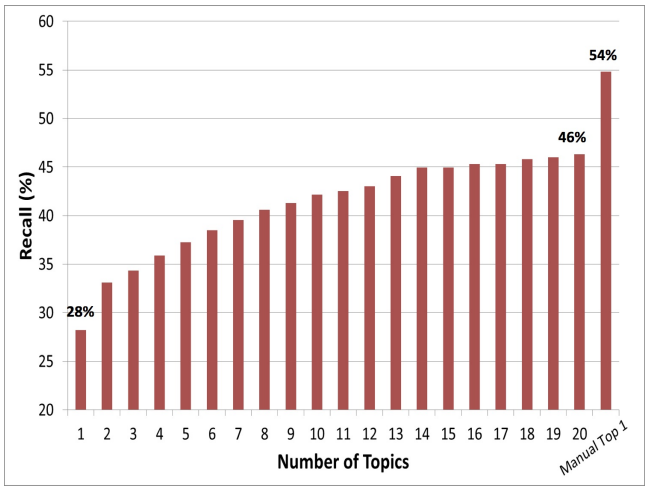

Figure 6: Recall for LDA

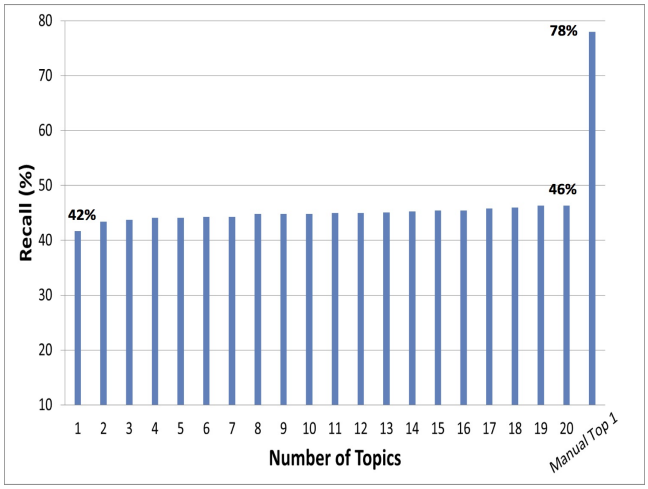

Figure 7: Recall for VSM+LDA combined approach.

With the standard pre-processing of tokenization, stop word removal (standard words in MALLET) and markup-removal, the best recall over the top-20 issues removed was 30%. We observed a increase of 3.01% with the additional pre-processing stages of URL removal, spam removal and stemming.

Following this, as described in Section 5, we systematically varied the *IDF* threshold to expand our list of stop words. As seen in the figure, the recall improved until $IDF = 4$, and started dropping off gradually. We choose $IDF = 4$ for the remainder of the analysis. However, with this expanded list of stop word,s we observed only a 1.57% increase in recall.

By manually sampling we observed that our true recall was 67%. Further, as noted during our manual sampling, many of the mismatches resulted from the appearance of the template issue statements in the issue description.

7.2.2 VSM

As anticipated, after removing frequent sentences, we observed a dramatic increase in recall. In Figure 4, we report the recall values for the top-k ranked retrieval (with $k = 20$) and our observation from manual sampling. We observe that our best recall achieved is 83%.

As we pick the top-20 similar issues, ranked by similarity, we are able to observe the variation in precision with recall and thereby report the mean average precision. We plot the PR-curve in Figure 5. We observe that the best precision is about 50% and for a 5% improvement in recall, there is a corresponding 20% drop in the best precision.

7.2.3 LDA

Similar to our observation in the case of VSM, we observe that the recall improves with the increase in k. Comparing against VSM, if only the most relevant issue was picked, LDA offers only a recall of 26% whereas the same was 49% in the case of VSM. Increasing the number of issues retrieved from 1 to 20, for LDA, we observe an improvement of recall of roughly 20%. Similar to the previous case, on sampling manually, we realize that the retrieval of similar issues is much better and the true recall is 54%. This is worse than what we observed for VSM.

7.2.4 LDA + VSM

Further, by combining LDA and VSM, we observe an improve-

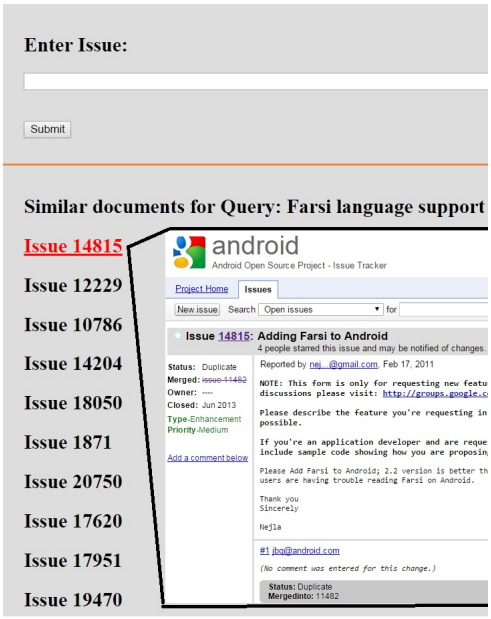

Figure 8: Screenshot from the demonstration portal.

Table 4: Summary of results. Note: The MAP values here are approximated using the area under the PR-curve.

	VSM	LDA	LDA+VSM
Best recall	83%	54%	78%
MAP	34.66%	6.46%	7.14%

ment over pure LDA, but the overall performance is still less than that of pure VSM. By selecting only the most similar issue, the recall is 42%. Contrary to the other two techniques, we do not observe any significant improvement in recall by increasing the number of issues (top-k) retrieved. With $k = 20$, the recall has only improved by 4%. However, as observed previously, while sampling manually, we observed that our true recall is much better at 78% which is still less than that of pure VSM.

In Table 4, we summarize the best observed recall values (from manual sampling) and the calculated *MAP* over exact matches against the set of original issues (G).

7.3 Demonstration Portal

To demonstrate the duplicate issue detection capability, we designed a proof-of-concept web portal where users can enter an issue, and the portal will use VSM to retrieve the top-20 issues related to the query, ranked in descending order of similarity. We use only a small subset of the issues (about 500 issues) for this purpose. This is due to the fact that our implementation of VSM is not optimized for speed and scale which results in long processing times.

Hence, for demonstration purposes, we limit ourselves to this subset of issues. In Figure 8, the screen shot of the demo portal is shown where the user issued query was “Farsi language support", and the retrieved top-20 issues are listed. In this example, the most similar issue found was “Issue 14815" and the actual issue is overlaid in the graphic for the reader’s understanding.

The portal is live and available for public access[9].

[9]https://is.gd/aospduplicates

8. CONCLUSION AND FUTURE WORK

We have presented our first steps toward automating detection of duplicated bugs for AOSP. We show some examples for the good, bad and ambiguous cases of issues picked as duplicate using our proposed approach, as shown in Figure 9. Figure 9(a) illustrates a good case example where the original issue, the issue marked as duplicate and the one picked by our approach were very similar. Figure 9(b) is a bad case example, as the topic of sending messages in Android 2.2, was picked as issues about sending mails in Gmail. Figure 9(c) shows an ambiguous example.

In summary, our findings indicate the use of VSM to work sufficiently well in detecting duplicate issue for AOSP, compared to other techniques like LDA, and LDA + VSM. We designed a proof-of-concept web portal where VSM technique is used to retrieve the top-20 issues related to a user-defined query. We demonstrated that classification technique using VSM achieves higher precision (of 83% from manual pick and 67% from Top 20 retrievals). These results informed us of a system that could benefit active users of AOSP in finding bug reports more efficiently.

In the future, we seek to extend our implementation to provide for other language development support tool. Second, we will expand our study to evaluate larger data sets, especially since we disregarded comments, code and traces in this study. Technical terminologies like class names, methods, and errors, particularly, make common query keywords among expert users. We also intend to continuously stream the AOSP issue tracker and update the dataset periodically. Finally, we will expand our system to exploit the package hierarchies in Android API, which we believe could achieve better inference rules.

9. ACKNOWLEDGEMENTS

This work was supported by Singapore Ministry of Education Academic Research Fund Tier 2 under research grant MOE2011-T2-1001 and by the National Research Foundation, Prime Minister’s Office, Singapore under its IDM Futures Funding Initiative. Kasthuri Jayarajah is supported by an A*STAR Graduate Scholarship. All findings and recommendations are those of the authors and do not necessarily reflect the views of the granting agency, or Singapore Management University.

10. REFERENCES

[1] *Getting started with topic modeling and mallet. http://programminghistorian.org/lessons/topic-modeling-and-mallet/ last accessed: June 10, 2016.*

[2] *Idc 2015 android market share predictions. http://www.idc.com/prodserv/smartphone-os-market-share.jsp last accessed: June 18 2016.*

[3] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, *Detecting duplicate bug reports with software engineering domain knowledge*, In Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER’15), IEEE, 2015, pp. 211–220.

[4] Anahita Alipour, Abram Hindle, and Eleni Stroulia, *A contextual approach towards more accurate duplicate bug report detection*, In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR’13), 2013, pp. 183–192.

[5] Nicholas Jalbert and Westley Weimer, *Automated duplicate detection for bug tracking systems*, IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN’08), IEEE, 2008, pp. 52–61.

(a) Good (b) Bad (c) Ambiguous

Figure 9: Issues Picked by our approach

[6] Nguyen T. T. Nguyen T. N. Lo D. & Sun C. i Nguyen, A. T., *Duplicate bug report detection with a combination of information retrieval and topic modeling*, In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE’12), ACM, 2012.

[7] Nawagata Nilambari, Vintee Chaudhary, and Shivani Gautam, *A survey on automated duplicate detection in a bug repository*, International Journal of Engineering Research & Technology (IJERT) **3** (2014).

[8] Gerard Salton, Anita Wong, and Chung-Shu Yang, *A vector space model for automatic indexing*, Communications of the ACM **18** (1975), no. 11, 613–620.

[9] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang, *Towards more accurate retrieval of duplicate bug reports*, In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE’11), IEEE Computer Society, 2011, pp. 253–262.

[10] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo, *A discriminative model approach for accurate duplicate bug report retrieval*, In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, 2010, pp. 45–54.