Automatic Topic Classification of Test Cases Using Text Mining at an Android Smartphone Vendor

Junji Shimagaki
Kyushu University
Fukuoka, Japan
yiu31802@gmail.com

Yasutaka Kamei,
Naoyasu Ubayashi
Kyushu University
Fukuoka, Japan
{kamei,ubayashi}@ait.kyushu-u.ac.jp

Abram Hindle
University of Alberta
Edmonton, Canada
hindle1@ualberta.ca

ABSTRACT

Background: An Android smartphone is an ecosystem of applications, drivers, operating system components, and assets. The volume of the software is large and the number of test cases needed to cover the functionality of an Android system is substantial. Enormous effort has been already taken to properly quantify *"what features and apps were tested and verified?"*. This insight is provided by dashboards that summarize test coverage and results per feature. One method to achieve this is to manually tag or label test cases with the topic or function they cover, much like function points. At the studied Android smartphone vendor, tests are labelled with manually defined tags, so-called "*feature labels (FLs)*", and the FLs serve to categorize 100s to 1000s test cases into 10 to 50 groups.
Aim: Unfortunately for developers, manual assignment of FLs to 1000s of test cases is a time consuming task, leading to inaccurately labeled test cases, which will render the dashboard useless. We created an automated system that suggests tags/labels to the developers for their test cases rather than manual labeling.
Method: We use machine learning models to predict and label the functionality tested by 10,000 test cases developed at the company.
Results: Through the quantitative experiments, our models achieved acceptable F-1 performance of 0.3 to 0.88. Also through the qualitative studies with expert teams, we showed that the hierarchy and path of tests was a good predictor of a feature's label.
Conclusions: We find that this method can reduce tedious manual effort that software developers spent classifying test cases, while providing more accurate classification results.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

software testing, classification, dashboard

ACM Reference Format:
Junji Shimagaki, Yasutaka Kamei, Naoyasu Ubayashi, and Abram Hindle. 2018. Automatic Topic Classification of Test Cases Using Text Mining at an Android Smartphone Vendor. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239235.3268927

1 INTRODUCTION

Manufacturing an Android phone is software intensive. Android needs to be customized to fit the requirements of the organization, the hardware supported, and the target end-users. Thereby, Android is a large software ecosystem consisting of many layers composed of applications, components, drivers, operating system, and a kernel. To take charge of this complexity, organizations such as the studied company engage in rigorous and comprehensive testing of their own particular Android ecosystems. At the company[1], test cases covers a wide variety of functionality supported across the software stack by applications and even the kernel.

Given the complexity, the many features and functions, the tests must be comprehensive and numerous. The test cases provide feedback on the stability and state of the current level of development—but there are too many test cases to review. The company engages in using project dashboards that summarize test case coverage and test case status to give insight to stakeholders. Stakeholders include developers, project managers, and product managers, all of whom require different information and have different intents. Developers want to know how stable or finished components they rely upon are. Project managers want to understand their developers' current progress. Product managers want further overviews across the entire system to understand scheduling, blockers, and other issues that arise during development.

To enable project dashboards about testing, the company's developers have painfully hand annotated and labeled test cases with their functional purpose, called "*feature label (FL)*". The FLs are a small collection of labels, around 10s to 50s labels per feature, and are defined by the feature owner. Test cases gain summarized information of test content by the annotation of the label (See Section 2.4). These labels are aggregated by the project dashboard, allowing insight into each project, and an overview of the entire product especially from the quality assurance (QA) perspectives. For instance, it is now easy to list the areas of FLs where more testing resource is demanded. Examples of FLs include *11ac* and *WPS* for a wireless product. Unfortunately, if developers do not label tests then the insight is limited. In order to promote the labeling of tests we have built machine learning and NLP based systems to help predict and suggest the appropriate label/topic of the test case so that developers are not burdened by this task.

[1] We cannot disclose the company name because of confidentiality reasons.

This paper makes the following contributions:

(1) We propose a method of automatic labeling of test cases based on textual descriptions and file pathnames.
(2) We investigate the work/accuracy trade-off between manual effort labeling and automated labeling performance.
(3) We quantitatively and qualitatively validate the labeling approach with the company's practitioners—demonstrating that the deployed approach can reduce time and effort while achieving acceptable prediction accuracy.

2 TEST CASE MANAGEMENT AT A SMARTPHONE VENDOR

2.1 Application Lifecycle Management with ALMS

The *ALM*—Application Lifecycle Management—is a way to manage a set of pre-defined processes that exist to streamline business—such as software development—from start to finish, e.g., product releases or the termination of the product support [15]. In software development, ALM deals with requirements, source code configuration, build configuration, project and release management.

The *ALMS* (ALM Software tool)[2] provided by Micro Focus[3], formerly part of Hewlett Packard[4], is a software tool that helps manage the ALM process in an integrated way. The ALMS is shipped with applications that ease the necessary ALM processes such as testing, defect tracking/fixing, and requirements via a web interface or a dedicated Windows application.

Data Structure of Test Cases with ALMS. Figure 1 shows a screenshot of the actual ALMS use at the company. Note that the image is modified to hide some proprietary information. In the ALMS, test cases are structured in a similar way as in typical file system; it has directories, and directories can contain directories and test cases (2. in Figure 1). A test case must have a test case name (1. in Figure 1), which is one of unique identifiers apart from the automatically allocated integer ID. It also contains a test case description (4. in Figure 1) which explains more details about test case setup, test conditions, test steps, expected results, and so on. Finally, test cases are recommended to have a so called *feature label* (3. in Figure 1), which is the subject of this paper and we will explain it more in Section 2.4

2.2 Test Case Management with ALMS at the Studied Company

Our studied projects are developed by an Android smartphone vendor; it has been developing and releasing Android smartphone products for many years. Each time the company has a product development plan, it follows the same ALM processes including requirements and release management. Due to Android supporting many legacy features, each release typically comes with even more features. As the release version increases so does the customer's and market's high expectations in terms of quality thus quality assurance is a critical priority among all ALM processes at the company.

[2]https://software.microfocus.com/en-us/products/application-lifecycle-management/overview
[3]https://www.microfocus.com/about/
[4]https://www.hpe.com

The ALMS is suitable in such a situation where lots of test assets are reused over time, i.e., over different products or Android OS versions. For example, ALMS is capable of creating new test cases by copying existing test cases from old software projects and by slightly modifying the test case steps.

In reality, at the company, not all test cases are managed in the ALMS database as teams have freedom in coming up with the best solution to manage their test cases or assets in general, and the ALMS is just an option among many possible options (e.g., Excel, text files, etc.). Many teams completely manage everything in the ALMS, whereas other teams still have difficulty in managing tests and assets in the ALMS because of the rigid data structure (See Section 2.1). Teams responsible for many exploratory test cases tend not to prefer the ALMS because their test cases are designed to verify features in a cross-functional way and find it difficult to describe all possible scenarios while making reproducible testing reports. Teams with test cases who can describe their scenarios and results by a simple description tend to choose the ALMS.

2.3 Problems Faced during Evolution of ALMS Operations

For many teams, the ALMS outperforms conventional test case management with Excel, spreadsheets, or any other local files, because the database is centralized and engineers can create and share as many test cases as they want with minimal effort. However, after using many ALMS operations and because of the high flexibility in the ALMS, stakeholders started to face issues such as:

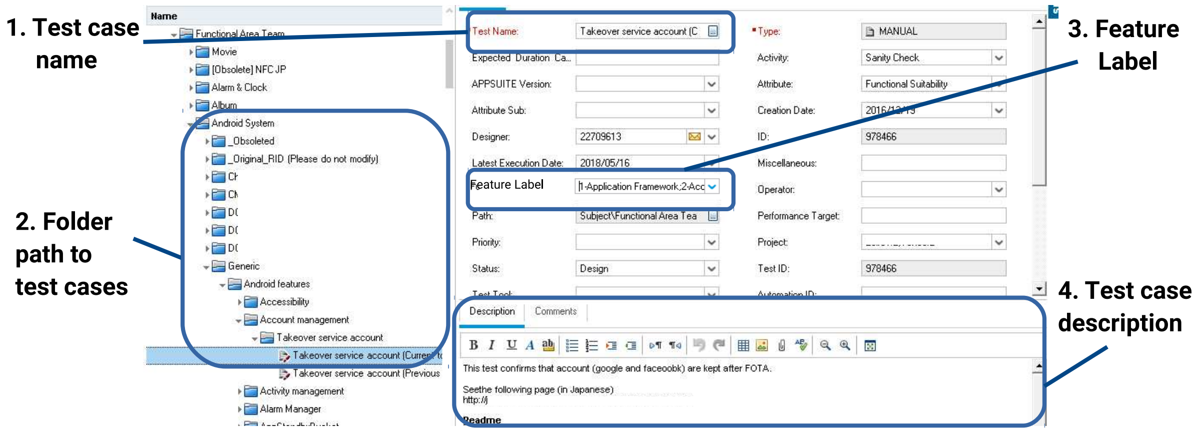
- Inconsistent size of test cases.
  Some teams create large test scenarios that check many application features in one test case, but other teams create many test case records of a single feature. The ALMS allows everybody to store their test cases, however, aggregation of those (e.g., to generate weekly verification progress reports) sometimes does not make sense because of the inconsistency, e.g., test case size.
- It is difficult to extract high level summaries of test cases.
  Test cases are primarily used internally within the team, thus both of the test case name or its description tend to contain many domain specific and implementation specific terms, that external stakeholders can no longer follow. The intention was that some summary information would be attached to individual test case records so that external test engineers can understand the purpose of the test without reading the source code.

2.4 "Feature Label" (FL) in Test Cases

To overcome the issues discussed in Section 2.3, ALMS users and administrators introduced a new field "*Feature Label*" (FL) in test case records that represents software features being tested in its test case. In other words, the FL is summarized information of test case name or test case description. Each team is responsible for coming up with labels; the number of them, the balance between simplicity and details, etc. Test cases are discouraged to have multiple FLs due to compatibility with internal dashboard presentation tool, thus most of test cases currently have only one FL.

Many stakeholders were satisfied by the introduction of FLs because of the following reasons:

- Test case sizes became similar with the FL.



Functional Area Team
- Movie
- [Obsolete] NFC JP
- Alarm & Clock
- Album
- Android System
  - _Obsoleted
  - _Original_RID (Please do not modify)
  - Ch
  - CN
  - DC
  - DC
  - DC
  - Generic
    - Android features
      - Accessibility
      - Account management
        - Takeover service account
          - Takeover service account (Current
          - Takeover service account (Previous
      - Activity management
      - Alarm Manager

Test Name:	Takeover service account (C	Type:	MANUAL
Expected Duration Ca...		Activity:	Sanity Check
APPSUITE Version:		Attribute:	Functional Suitability
Attribute Sub:		Creation Date:	2016/12/19
Designer:	22709613	ID:	978466
Latest Execution Date:	2018/05/16	Miscellaneous:	
Feature Label	1-Application Framework;2-Acc	Operator:	
Path:	Subject\Functional Area Tea	Performance Target:	
Priority:		Project:	
Status:	Design	Test ID:	978466
Test Tool:		Automation ID:	

This test confirms that account (google and faceoobk) are kept after FOTA.

Seethe following page (in Japanese)
http://j

Readme

Figure 1: A screenshot of the ALMS; browsing a test case from the tree.

The level of granularity in FL slowly became similar, even across different feature domains. Thus, external stakeholders., e.g., test leaders or product managers can extract test results of many teams to compare them.

- Dashboards can visualize the distribution of test cases by the FL. The FL has typically direct link to a small component of a feature, and the dashboard can provide an overview of testing status of a feature domain at a fine-grained level. Stakeholders can easily understand the detailed testing status without interpreting domain specific test case descriptions.

2.5 Overhead of Feature Labels

Yet from the test case provider side, test engineers began to claim the extra effort around the FLs.

The initial challenge when using the FLs is to come up with appropriate FLs for their team. The FLs ("*feature*" labels) are literally defined by a "*feature*" scope owner, which are equivalent to "*team*" at the company. The FLs are unique per team, and are not shared with another team, and this is quite practically reasonable rule because, for instance, the FLs of WLAN team may consist of [*11ac*, *WPS*, ...] and that of Android Framework may consist of [*ART/Runtime*, *AppWidget*, ...]; apparently those two sets of FLs have no relationship each other and their test cases should be quite different as well. The FL definitions must be agreed within the team, so before reaching the final form of FL definitions, it usually has a loop of review involving multiple people.

Even after reaching a good set of FLs, the developers still had 100s to 1000s test cases that they have to manually label based on the FLs they have defined. Labeling test cases requires domain knowledge and investigation; a typical workflow is to read the test case name, check the file path, and do further investigation such as checking the source code that is tested, previously found bugs by the test case, etc. The task is time consuming, much like other tagging tasks [1]. Test engineers started to feel uncomfortable with extra work caused by the FLs.

2.6 Motivating Automatic Classification of FLs

Gradually, developers started saying that *"Although FLs are suitable to overview our test case assets, it would be even greater if some of initial cost is eliminated"*. From the first author's observation, which is supported by a few of engineers worked on the FLs, FLs tend to have strong associations with file paths and the naming, because developers tend to create many folders to place similar test cases in one place. Still, the folder structure tends to be deep which makes trivial labeling impossible. For example, if test cases are collected in folders based on the network environments, test cases under *NW_Career_JPN* may cover a range of features with minor adjustment of the test scenarios based on the network environment.Test cases may be distributed in *NW_Career_JPN/SMS* and *NW_Career_JPN/Call* but labeling all test cases under *NW-_Career_JPN* as "*SMS*" would be incorrect. Thus, some advanced technique beyond simple labeling is needed to meet the practitioner's demand, and we believe the machine learning with NLP technique should definitely help.

3 QUANTITATIVE STUDY SETUP

In Section 2.4 to 2.5, we showed that a new attribute "FL" (Feature Label) improved the way test engineers work and provide insight to other stakeholder, but it is costly because of the manual effort required. Throughout our study, we try to understand: *"Is it possible to automatically assign a FL to all test case records using textual information?"* In this section, we show our experimental setups using text information retrieval technique, and statistical modelings.

We first introduce the datasets studied in this paper. Next, we explain the basic methodology of model construction and performance measurement. Finally, we explain the parameter tuning mechanism that is tightly integrated in the model construction, to find the best performing model out of numerous model parameters.

3.1 Our Studied Datasets

In our study, we pick test case data from 6 different teams out of over 50 teams at the company. We chose those teams because most of their test cases are managed in the ALMS database, and their test case records contain reliable FL values on most of their maintained records. Note that not all test cases are managed in the ALM database, as stated in Section 2.2. Thus, the number of test cases is by no means proportional to the actual testing effort and time. In Table 1, we show a brief summary of the test cases versus

Table 1: Summary of the studied datasets.

Team Alias	Team Domain	# Test Cases	# Missing Labels	# Unique Labels	# UT(*) in `Name`	# UT in `TC_Path`
A	Multimedia	711	422	15	886	69
B	Multimedia	571	375	10	519	149
C	Android OS and Linux Kernel	3,129	1,867	59	3,766	861
D	Android OS and Linux Kernel	107	38	9	246	57
E	Cellular and Connectivity	3,047	180	45	3,109	428
F	Cellular and Connectivity	1,824	0	62	1,353	278

(*) UT: Unique Tokens

teams:

- The number of test cases per team varies between 107 to 3,129, and the team domain has no relationship with it.
- There are many missing labels (i.e., test case records without manual labeling of FL), but practically many of those unlabeled records are usually obsolete test cases which are not maintained.
- The number of unique labels are roughly proportional to the number of test cases, i.e., teams implicitly tend to keep the number of test cases in one group not too high.
- The number of unique tokens in `Name` (Test Case Name) are always larger than that in `TC_Path` (Test Case Path) because test case names must be usually unique.

3.2 Model Constructions and Performance Measurement

Figure 2 shows the high level summary of data flow in our quantitative study experiments. In the following, we explain each of the operations in detail. The main workflow is we load the test cases, split into training and test, vectorize the texts used, tune the parameters of the model on the train set, train the final model on the full train set, then evaluate on the test set. In tuning we repeatedly split the training up into validation and train to find the best set of parameters given a limited range of configurations or a limited time to search.

3.2.1 Loading the Test Cases. We first load test cases from ALMS as CVS format (*a* in Figure 2). We have 6 datasets and separately treat them. We only load the data of the experiment's concerned team (*i1* in Figure 2). The raw data consists of 4 columns, which are subset columns from the ALMS database:

- Test Case ID, which is uniquely assigned in the ALMS database.
- Test Case Name (`TC Name`, which is defined by the users and usually in a free text format of 100–200 characters in length.
- Test Case Path (`Path`), which is the absolute path of the folder that contains the test case.
- Feature Label (FL), the expected label of the test case indicating the feature it tests. This is set by users whereas some users keep it blank when there is no input

We then select the column of use depending on the experiment setup; `TC Name` or `Path` (*i2* in Figure 2). We also select the feature label column, which works as our dependent variable in statistical modeling context. We finally filter out those rows where the feature label field is empty, because most of them are intentionally left unlabeled as discussed in Section 3.1.

3.2.2 Splitting the Test Cases into Training and Testing Datasets. We split the raw data into 2 parts; training and testing (*b, c* in Figure 2). The ratio depends on the experiment setup, but we split on all rows unless otherwise noted. In RQ1 and RQ2, we randomly split the raw data in half into training and testing datasets. In RQ3, we randomly split the raw data, but the number of test cases datasets varies in training and testing datasets.

3.2.3 Tuning Parameters. Before starting statistical model construction of the final models, we perform parameter tuning only on the training set. The purpose here is to find the best performing parameters (*d* in Figure 2) for a given training dataset. The set of parameters vary depending on the statistical model selection (*i3* in Figure 2), and whether the model is composed of *word count vectors* or *LDA topic vectors*. These vectors will be discussed in Section 3.2.4. The tuning logic will be discussed in Section 3.3.

3.2.4 Vectorization of Training texts. Having the column of text data, we convert the texts into bag-of-words representation, separating the words by splitting on characters such as `"-"`, `"_"` and `" "` (a space character). We remove words that consist of numeric characters only, because numeric characters are used alone in many places to sort directories. We then remove words that appear in almost every test cases (5% most common), and remove words that appear only once. Later we build a "corpus-dictionary" of vocabulary (*f* in Figure 2) that maps words to integers indices from training texts. The corpus-dictionary will be used to construct tokenized vectors for a test dataset in Section 3.2.6. Finally, we apply the TF-IDF transformation on the vectors to amplify the explanatory power of rare words. We use the Python `gensim` library [16] (3.4.0) to apply TF-IDF conversion (default parameters) to the entire set of word vectors.

If an experiment setup involves LDA (*i3 = true* in Figure 2), we convert the word count vectors into so-called *LDA topic vectors*, by training an LDA model to extract topics from our texts. LDA is an unsupervised algorithm that seeks to allocate documents with underlying topics. These topics are distributions of word counts. Effectively LDA can be used as a dimensionality reduction techniques to convert bags of words into associated topic allocations—vectors of length N where N is the number of LDA topics chosen [18]. LDA has numerous parameters, e.g., number of topics, α, and η are which we tune for (*d* in Figure 2). In our experiment, we choose these parameters as following:

$$\alpha = \eta = \frac{m}{N}$$

where m is the magnitude to the parameters, which are selected

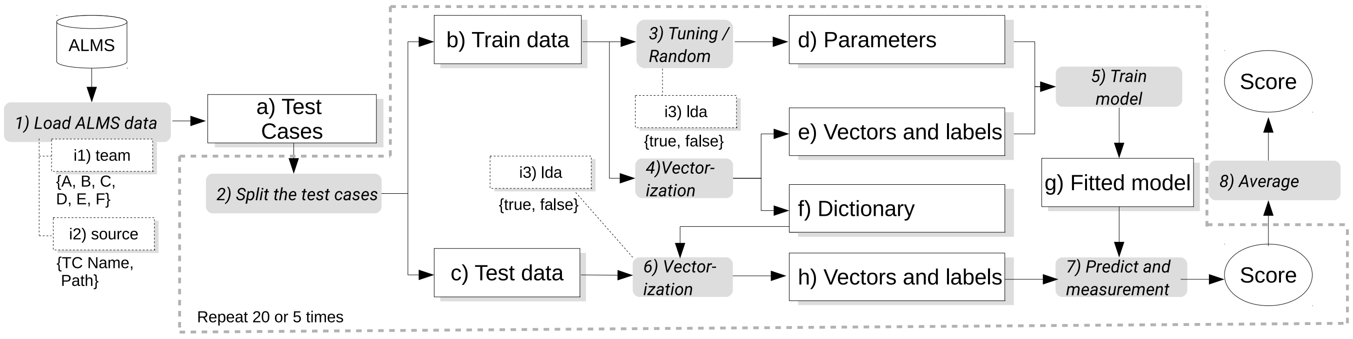


Figure 2: Flow-chart of how text data is processed

in the tuning from 0.1, 1.0, 10.0—thus α is always relative to the number of topics N.

We also convert textual label data into single label via one-hot encoding. The resulting vectors are noted as e in Figure 2.

3.2.5 Training a Model using the Best Parameters. In our study, we use a K-nearest neighbor model to infer the feature label by a given dependent word count or LDA vector. We use the Python `scikit-learn` library [17](0.19.1) on the model constructions. The K nearest neighbor model measures the Minkowski distance of a concerned vector against all vectors in the training dataset. And the K nearest vectors are assumed to have the most probable label, and it simply picks up the most voted label out of it. Minkowski distance is used because document sizes are relatively similar so the size normalization of cosine distance is not needed. In our study, we use the best performing K out of 1, 3, 5, 9, 15, which discovered via tuning (d in Figure 2).

3.2.6 Vectorization of a Test Dataset. Unlike the vectorization of a training dataset discussed in Section 3.2.4, the vectorization of a test dataset is not purely made up of word occurrence of its dataset; the word count vectors are built based on the corpus-dictionary of the training dataset (f in Figure 2), i.e., words that appear only in the test dataset are ignored, because the fitted model is only on the words that the model has ever seen. Additionally, if the experiment setup requires LDA vectors, i.e., *i3 = true* in Figure 2, the word count vectors are converted to LDA vector also by using the LDA inference on the trained LDA model (e.g., LDA topic conversion). The feature label is also encoded in as same value as in the training dataset, and the resulting vectors and labels are shown in h in Figure 2.

3.2.7 Predict and Performance Measurement. We explain how our model returns the predicted values, and how we measure the performance of the model so that we can compare multiple models.

Prediction. We use K-nearest neighbor model as our statistical model, and its classifier algorithm is to simply choose the most voted label as a predicted label. If votes spread over different labels equally leading to a tie, we output such case as "*unpredictable*", which does not contribute to performance gains, but also not to performance loss of certain metrics. For engineers practically, "*unpredictable*" is often better than predicting a wrong FL, because developers can efficiently provide help when machine learning is not confident.

Performance Measurement. All predicts are compared with the actual FLs, and if a predicted label e.g., *#a*, correctly points to the actual FL, i.e., *#a*, such predict is counted as "*true positive*" of *#a* in our sense, whereas if a predicted label wrongly points to another *#b*, such predict is marked as "*false negative*" of *#b*. "*unpredictable*" does neither add up true positive nor false negative.

Using the number of true positives and false negatives, we define following performance metrics: The "*true positive ratio (TPR)*" is defined by

$$TPR = \frac{\text{True Positives}}{\text{Number of Test Dataset Rows}}$$

whereby the TPR is calculated for the entire dataset. The "*F1*" of *label x* is defined by

$$F1 = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$

whereas the precision is the number of true positives of *label x* divided by the number of the rows of *label x* in the test dataset, the recall is the number of true positives of *label x* divided by the sum of true positives and false negatives of *label x*. The F1 is calculated for each label, and the overall performance metrics is a median value over the F1 scores of all labels. In RQ3, we also evaluate the return of investment, by calculating the *profit* (PRO), which is defined by

$$PRO = \frac{\text{True Positives} - \text{Number of Training Dataset Rows}}{\text{Number of Test Dataset Rows}}$$

In short, PRO is the *gains* (number of true positives, which our ML engine produce as values) deduced by the *cost* (number of training data rows, which testing engineers manually spend). PRO is normalized by the number of test data rows, so that it is comparable for different datasets. Same as the F1 scores, PRO is calculated for each label, and a median value represents the overall performance metric.

3.2.8 Result Summarization by Averaging. We repeat the process between 2 to 7 in Figure 2 for 20 times. We summarize the results by box-plots, point plots for visual representations, and by averaging them to get the model's performance.

3.3 Parameter Tuning

The main purpose of parameter tuning is to find the best performing parameters set out of numerous combinations of parameters

Table 2: Parameters to be tuned.

Name	Selections	Description
k	$1, 3, 5, 9, 15$	k of K-nearest neighbor model
n_{topic}	$20, 50, 100, 200$	Number of topics in LDA
α_{mag}	$0.1, 1.0, 10.0$	Order of magnitude of α in LDA

Table 3: Input variables in RQ1 to RQ3.

RQ	Train:Test	Source-Stat.Model
RQ1	50%:50%	**(Name-WC, Name-LDA)**
RQ2	50%:50%	**(Name-WC, Path-WC, Path-LDA)**
RQ3	**(7.5%-50%)**:50%	Path-WC

within a limited amount of time. The parameters and their possible values are noted in Table 2; we have 5 combinations for word count vector model, and 60 combinations for LDA vector model. We randomly select 1 combination with replacement, and we will set the timeout length sufficient long so that at least 10 combinations are tried.

At each combination, we follow similar steps from the step 2 to 8) in Figure 2. The difference is we use a training dataset as input a) in Figure 2. Then we randomly split it in half as a new training dataset (b in Figure 2) and a validation dataset (c in Figure 2). Then, we perform the step 4) to step 8) with parameters at the combination. We repeat 5 iterations to mitigate the noise of randomness and calculate the averaged performance with the parameters.

When timeout is reached, we compare all of the averaged performance measurement values across parameters combinations, at least 10, and decide which parameters are the best among them.

4 QUANTITATIVE STUDY RESULTS

In this study, we conduct a numerical simulation of FL predictions while we change the input parameters depending on the objectives of the question. The summary of variables are put in Table 3, whereas target variables are highlighted with parentheses.

4.1 RQ1: Does LDA help to improve the performance?

Motivation. LDA has been known to be powerful unsupervised algorithm to transform large sized text data into much shorter vectors of associated topics. In our case, the vocabulary is not large compared with existing studies [4, 5, 8, 9]. Thus we want to know if LDA improves the overall model performance for inferring FLs.

Approach. We use the `TC Name` source of all of the 6 teams A to F. The training and test dataset split ratio is constantly set to 50% and 50%. Each of the datasets, we build 2 models; the KNN model using LDA vectors (Name-LDA) and the KNN model using word count vectors (Name-WC). We then measure the median F1 score and the total positive rate. This overall procedure is repeated for 20 times with a new sampling, and the resulted scores and rates are reported in a box-plot format to visually and numerically evaluate whether which of Name-LDA or Name-WC has better performance.

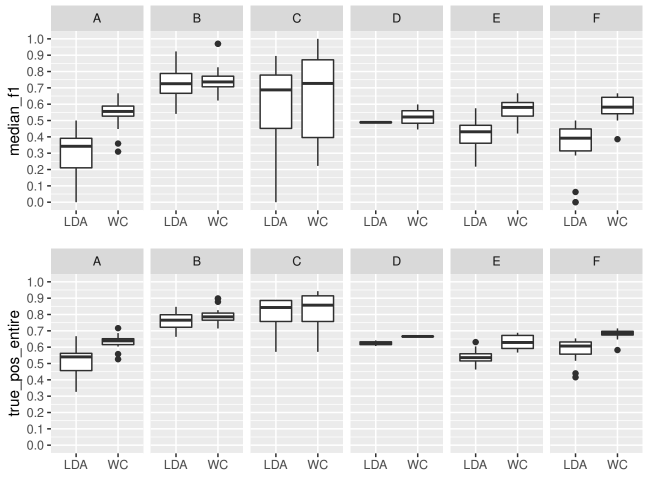


Figure 3: RQ1 — Median F1 (upper graph) and total true positive ratio (lower graph) comparisons (LDA and WC).

Results. Figure 3 (upper) shows the median F1 score distributions of the two models per team. We observe higher score distributions of Name-WC in teams A, D, E, F, suggesting that the simpler Name-WC is yet powerful to infer FLs. There is no clear difference between Name-WC and Name-LDA for teams B and C.

Figure 3 (lower) shows the total positive rate distributions. We can still see the same tendency with the F1 score distributions that (1) Name-WC model is slightly better with teams B, C, E, F and Name-WC is dominant for teams A, D, E, F and no much difference for teams B, C. When we examine the 95% confidence intervals, teams B and C indicate that WC and LDA perform similarly, while for teams A, D, E, and F, WC models perform better on average (95% CI: [0.0859,0.163] median F-1 difference between WC and LDA across all teams combined).

Word count vector model trends towards better performance, whereas 4 out of 6 teams have observable differences in F1 scores and true positive ratios.

4.2 RQ2: Which source of text predicts Feature Labels better?

Motivation. We hypothesize that test case names and folder paths are both related to the attached FL. Due to the nature of FLs, a feature label must be a brief representation of a group of similar test case names (`TC Name`). Also, as shown in Section 2.1, users generally try to manage test cases in an ordered hierarchical manner because these are many, so eventually the resulting folders tend to contain similar test cases in the same place. This means that due to the manual organization of test cases, a test case path (`Path`) should be related to FLs assigned to tests with similar paths.

Thus, both the test case name and folder paths should be a clear indicator for FLs, and we want to know which has even stronger predictive power so that we can achieve more accurate statistical models, and so that we can provide advice to test engineers about

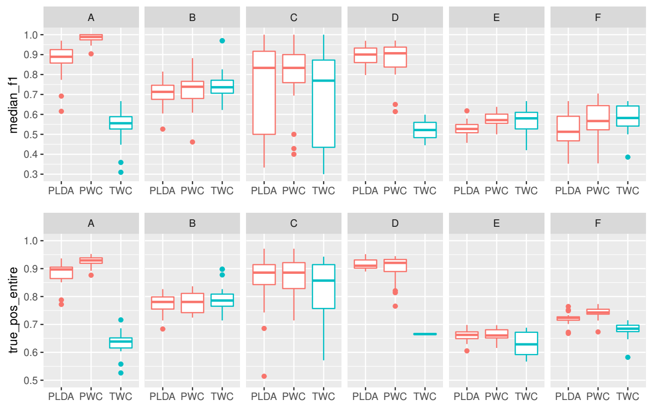


Figure 4: RQ2 — Median F1 score (upper) and true positive rates (lower) of `Path` (PLDA and PWC) and `TC Name` (TWC).

where their effort should be spent: description or organization. We do not mix the two sources at once for now, because our study is yet preliminary and we have project mandate to explain the fundamental theory behind the model performance to practitioners at the company, thus we need to clearly know which source of textual data is a better indicator for FLs.

Approach. The approach is the same as RQ1 except that our models this time are Path-LDA and Path-WC. In the resulting plots, Figures 4, we reuse the numerical results of Name-WC from RQ1 as a reference, because it performed slightly better than Name-LDA. We again plot the 20 repeated measurements for the models of 6 teams, whereby sampling, new model construction, and parameter tuning with at least 10 configurations.

Results. Figure 4 (upper) shows the distribution of F1 score for each of the teams, and Path-WC and Path-LDA are colored with red, Name-WC is colored with green. There is clear advantage for Path-(WC/LDA) models in Team A and D. The rest of the teams have almost similar performances. The team C here again has high standard deviations due to its small number of samples. The performance difference between WC and LDA vectors may be minor.

Figure 4 (lower) also shows the distribution of the entire positive rates for each of the teams. The results are almost equal to that of F1 score except that the standard deviations are slightly lower, and making the tendency of Path apparent to additional teams A and D too. Teams B,C,E, and F did not exhibit significant differences.

The vocabulary sizes of `Path`, as previously shown in Table 1, are smaller than that of `TC Name` by 3.5 times at smallest (Team B), 12.8 times at largest (Team A), and 6.7 times on average.

Globally for WC models, across teams, the 95% confidence interval of the differences in median F-1 measures between `TC Name` and `Path` was negative $(-0.162, -0.0857)$, showing that globally `Path` is dominant in performance. Although if we look at confidence intervals per team, most teams have similar performance between `TC Name` and `Path`, with only teams A and D showing `Path` dominating `TC Name` in F-1 performance.

Overall speaking, the results suggest that the text source `Path`, which has much lower amount of textual information, is better suited for inferring the associated FLs in most of the studied datasets.

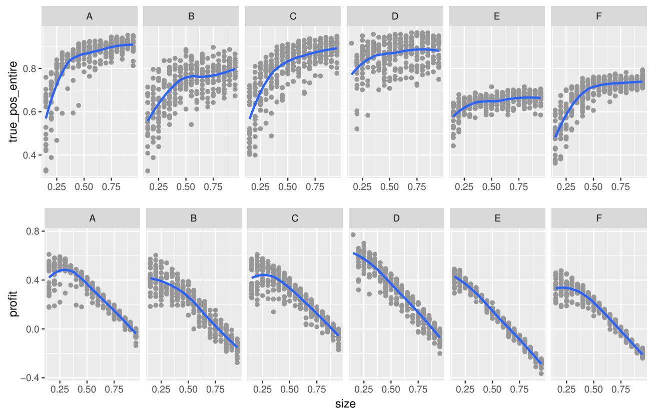


Figure 5: RQ3 — True positive rates (upper) and profit rates (lower) versus size using Path-WC model.

Folder path achieved higher predictive power of FLs globally, although its word space is substantially smaller than that of test case name.

4.3 RQ3: How small can a training dataset be?

Motivation. In Section 4.2, we saw that Models using `Path` (folder path to test case) have better performance compared to the another text data `TC Name` (test case name). In that experiment, we used a constant ratio of 0.5 for test and training data split. Practically in the future, we want to achieve the good accuracy without having rich amount of the training dataset so that we can produce accurate FL inference only by feeding a few manually classified test cases into our classification engine.

Thus, in this RQ, we study the effect of reducing the size of the training dataset. We also consider the ROI (*Return of Investment*) by calculating the *profit* (PRO) and we seek to optimize accuracy versus cost, which is important from a business perspective to stakeholders, especially test engineers. Test engineers want accurately predicted labels without much manual labeling. They do not want to spend extra effort training machine learners.

Approach. We use Path-WC for all of the 6 datasets we have, based on the performance comparisons in RQ1 and RQ2. In this experiment, we vary the ratio of the use of a training dataset; we first split the entire dataset into a test dataset and the rest by half. Then from the rest, we take 15% of that amount as a training dataset (i.e., 7.5% of the whole), and do the measurements of the entire true positive rate and the profit rate (PRO). We repeat the test and training samplings with the set ratio for 20 times, and then we increase the size of the training dataset by 5%. We continue this until the training set ratio reaches 100% (i.e., 50% of the whole). By increasing the training dataset size, we try to understand how the performance changes, and how the corresponding profit changes.

Results. We show the training dataset ratio versus TPR in Figure 5 (upper). As expected, constant increase in TPR is observed at the beginning, e.g., where the training dataset ratio is incremented from 10% to 40%. Especially, the dataset of teams with smaller dataset size, e.g., Team A and B have steep increase. Overall speaking, the increased performance rate looks to be gradually saturated, and we observe almost milder increase in TPR around 50% to 70%, implying that increased amount of training dataset does no longer improve the performance if the model already has rich amount of data to train.

In Figure 5 (lower), we show the PRO (profit ratio) of the same dataset. Team A, C and F have clear tendencies that PRO is gradually increasing at the beginning, i.e., gain is greater than cost. However, after around 30% (i.e., 15% of the whole), PRO started to decrease implying that the performance improvement by additional amount of the training dataset becomes low. The rest of the teams have constant decrease in PRO from the beginning, however, this implies that even a small amount of the training dataset may yield contribution to performance improvement, i.e., profit exists.

From cost balance perspectives, it is most effective to have 10% to 20% of the entire dataset for training to maximize the return of cost investment.

5 INTERVIEWS AND DISCUSSIONS

In this section, we qualitatively investigate our quantitative results with stakeholders involved in the projects that engage in feature labels. We investigated the potential underlying software engineering reasons for our results. To do so, we reached out to engineers from each team under study, who contributed to the manual labeling of FLs, and held open discussions by showing our results. We were able to get 3 engineers from 3 different teams to participate (50% response rate). In each of the discussions, we touched upon the questions we prepared:

- Why does `Path` often work significantly better than `TC Name` when inferring FLs?
- In which case does `Path` fail to produce accurate inference?
- How could this auto classification be beneficial for you?

Interviewees. The 3 engineers that we interviewed are all test engineers in the company, and each belongs to Team C, D, F with 5, 10+, 10+ years of experiences in the domain respectively. In the past, all of them participated in manual labeling activities, and still maintain the FLs if there is a need to update them. In the following, we denote each of the members as Int-C,D,F.

Interviewer and Setup. The first author conducted the interview with each of the members while showing the results of our Path-WC model. Each interview took 20 minutes, and was conducted in a meeting room. Interviews for C and F were in Japanese and translated to English, Team D's interview was in English.

5.1 "Why does `Path` often work significantly better than `TC Name` when inferring FLs?"

Motivation. Typically more features lead to be better machine results. Often larger training sets achieve better model prediction performance, but we achieved generally better performance with smaller texts such as `Path`, which has less unique tokens than `TC Name` (See Table 1) and generally performs better as in Section 4.2. We want to clarify why.

Discussion. Int-C mentioned the following:

"I know that `TC Name` *is hard to define, and sometime come up with naming randomly without consideration. Thus, we do not really trust it to find the right FL. How to find the right FL? Well, you will need to survey different sources, such as test case steps, the related source code, previously found bugs from the cases, ... etc."*

Thus `TC Name` was not the major indicator of FL for them in the context of the company. Int-D and Int-F also mentioned,

"`TC Name` *can be set by multiple persons and there is no clear naming rule for it.* `Path` *is more ordered because we try to put similar test cases in the same place and we have some guideline of maintaining the folders."*

They pointed out the issue of test case naming involves multiple people, that can easily lead to naming rule inconsistency. Keeping the consistency of many test case names can be done by strict naming rule. They seem to believe that `Path` is easier to maintain even with multiple people, because people tend to scan through existing folder structure before creating new folders.

Overall speaking, all of the interviewees show strong agreement that our approach should use `Path` instead of `TC Name`.

`Path` *is maintained carefully, and is quickly browseable, thus it is more ordered whereas* `TC Name` *has no rules and is maintained by many different people, which make the textual information less orderedly and harder to browse.*

5.2 "In which case does `Path` fail to produce accurate inference?"

Motivation. Our best model still failed to infer correct labels in some cases. For future improvement, we want to know where classification failures occur so we can improve our text processing and model construction to achieve even higher accuracy in the future.

Discussion. We identified 2 situations when inference failures could occur. **First,** we and Int-C, D firstly observed that there exists a pattern of mistake by inferring FLs of some teams: For example, there is a term `management` and there was a case where our model linked that with *"Power Management"* FL. So each time words such as *"process management and handling"* appear in `Path`, the inference could not correctly return *"Unix Process"* but it would return *"Power Management"* due to the word `management`.

Second, another situation dealt especially with Int-F of Team F, where numeric characters have special meaning, such as `2.4`, `5.0`, `802` in the wireless protocol areas—but in Section 3.2.4 we stated that words consisting of numeric characters were removed. These numbers are used in many places for ordering the folder names, and we believe such tokens can produce unnecessary noise. We need to find a way to intelligently distinguish numeric that has meaning so that it is included in our word vectors. We could use a similar approach to Aggarwal et al. [1] who suggest adding labels based on domain knowledge, such domain knowledge could be used to replace words that were protocols or version numbers with a token that will survive text transformation.

Errors arise from biases towards unique terms and inappropriate token splitting. Some non-important tokens happen to obtain strong predictive power of certain FLs. A domain specific text processing approach should be attempted such that domain specific tokens are retained to reduce classification error.

5.3 "How could this auto classification be beneficial for you?"

Motivation. In our quantitative study, we showed that it is possible to build an auto classification engine of FLs using historical data. On the other hand, our studied teams have higher input rate of FL input, e.g., Table 1 shows that Team E and F have 3,047 and 1,824 test cases, however, they only have 180 of those with missing labels for Team E, and 0 for Team F. Thus, we wonder whether there is room where automation can contribute. If the cost is not negligible, then it is difficult to apply the techniques to industrial projects. In this discussion question, we explore what could be the benefit for them if we have it as a deployment system.

Discussion. Firstly, they all agreed that the manual task on putting the FL was very time consuming. Int-D explained;

> *"At first, we spent days and weeks to come up with the current set of feature label definitions. I scanned the all of the test case assets we have, roughly classify similar test cases, made an initial proposal feature label definitions, asked a senior engineer to review them, and slowly we reached the current feature label definitions."*

As mentioned in Section 2.4, coming up with the right definitions of FLs is initially a tough problem to tackle—this is discussed later in Section 8. Int-F also explained his team's experiences:

> *"As you know, we have close to 2,000 test cases, but we had strong managerial pressure that we needed to put the FLs. We were able to manage it somehow, but we spent almost 3 months with multiple engineers, including proposal making and reviewing."*

Int-F also supplemented his opinion saying this tedious task should not be repeated in other teams. Int-F also makes it clear that the cost of labeling is not just 30 seconds reading the text of a test, it often involved communications between multiple engineers, exacerbating the cost of labeling. Int-C mentioned that

> *"Although our test cases already have manually assigned FLs mostly, we are still adding new test cases from time to time. We are often too busy to find an appropriate FL so we did not yet input the FL. It would be great if those are automatically updated; our existing test cases with FLs should definitely work as a training dataset. "*

Int-D confirmed that their test cases are still evolving and automatic classification should definitely help them in the future. They also showed understanding to our study's accuracy as well:

> *"We are happy to have this studied result deployed. We can happily accept the error ratio of e.g., 30% cause it is much smaller portion compared to that of what we have to work on now."*

By making use of our proven success with Team A to F, we can apply our technique to new test cases and more teams at the studied company, not to repeat the tedious manual labeling work.

6 RELATED WORK

Developers and stakeholders have information needs, as is discussed by Buse et al. [2]. Often this information can be aggregated within dashboards, as suggested by Truede et al. [13], such as the dashboards used by the studied company, and other companies [10] in this study. Fundamentally software artifact analysis must be funneled into dashboards, which is the aim of this study.

Traceability. Traceability is a term used in software engineering to discussing the linking of software artifacts [6]. In this study we care about linking test-cases to features automatically. Keenan et al. [6] describe Tracelab, a kind of benchmark suite for traceability tasks, algorithms, and data—it does not focus on test cases.

Features and Test Cases. This proposed method is meant to leverage test-cases to characterize functions and features of a system. This approach is similar to function points [3], a unit of measurement of software that is manually estimated in order to suggest functionality and size of functionality that is completed.

Topic analysis. We leveraged LDA in classification models. Panichella et al. [9] argue, like Blei et al. [18], that LDA should be tuned. They demonstrate the value of genetic algorithms for tuning LDA on SE tasks as an alternative approach to Blei's tuning on information content and topic quality. They apply LDA to traceability link recovery, feature location, and artifact labeling, much like this study. We engaged in constrained random search for tuning as we experimented with different kinds of models.

Hindle et al. [5] studied the LDA topics generated from numerous software requirement documents at Microsoft by surveying the actual practitioners' perceptions about those generated topics. They found that some topics make sense and practitioners were able to name them though, there are other topics which are hard to interpret by human. We apply a similar methodology but we do not expose the actual LDA topics to the end-users themselves.

Han et al. [4] used LDA to analyze bug reports of similar products produced by different companies and found that the bug reports had different vocabularies for the same topic, implying that organizations tended to be consistent with their own internal vocabulary for the same topics. McIlroy et al. [7] studied user review messages in mobile appstores, and described a method to assign multiple topic labels to the review by textual retrieval technique.

Test case prioritization. The studied company's underlying motivation of having the FL is to have a high level overview of the distribution of a large number of test cases, and take a necessary action if needed, e.g., manual test case prioritization. Test case prioritization is a mature field and numerous surveys have been written about test-case prioritization research [11, 14]. The intent of test case prioritization is to reduce redundancy in testing and run the most important test first to get feedback about current development. Some prioritization schemes seek test code that will be called recent commits to the software. Our work does not focus specifically on test-case prioritization but can leverage methods from it. Thomas et al. [12] employ topic models (LDA) for test case prioritization.

7 THREATS TO VALIDITY

Construct validity. To build the model that infers a feature label, we used two data resources: test case name and folder paths. We could have explored other attributes as well. If we use more data resources (e.g., test case description), we may be able to capture more characteristics of FLs for test cases. While we already achieve good performance inferring feature labels, future work will be to improve performance using other available resources.

We assumed that FLs that were manually labeled by the company's developers are true ones. Therefore, the FLs may be biased. However, the developers that labeled the FLs belong to the project and have more than at least 5 years experience as the company's developers. We believe that the bias may not be too large.

Internal validity. We used LDA to extract topics from word count vectors and KNN generated the model that infers a feature label. As shown in Table 2, we need to choose parameters for LDA and KNN. Although we apply parameter tuning in model construction, other parameters may yield different results.

External validity. We analyze six industrial proprietary projects. Our selection of subjects may introduce bias. To mitigate the risk, we select projects of various size, with different team domains. Our findings have been useful to help the company to put into action a plan to improve test case.

Our studied projects are based on the Android OS, whereby source code is clearly divided into hundreds of repositories given its gigantic scale. The directory structure of the source code enables organization and allows tests organization to mimick already existing paths. Software that is organized in a similar hierarchical manner would benefit from these techniques—but not all software is so organized.

8 FUTURE WORK

Feature Labels, their definition, and agreement on labels is a multi-person task and laborious. Future research should address the collaboration needs for discussing, defining, and deciding on feature labels. Therefore the perceptual quality of suggestions of labels needs to be evaluated, we propose instrumenting the existing UI with feedback buttons that allow users to rate the suggestion.

Reinforcement learning and online learning could be explored to help reduce the developer effort in labeling test cases. Such systems could request developers learn from particularly important examples rather than any example. Online learning would enable developers to update the model as they labeled test cases.

Domain specific specialization in the NLP and IR pipeline could improve both machine learning performance and run-time performance. Words like 802.11 and H264 should be part of the domain knowledge and not split apart by a naive tokenizer.

9 CONCLUSIONS

In conclusion we have presented a method of labeling test cases based on meta-data about the test case such as its organization (its path) and its short description (its name). This task is important as it provides dashboards with categorization used to infer high level insights about project state and health. We found that we could infer the correct feature being tested from these textual inputs 30-85% of the time.

We investigated multiple models and found that the path of the test case, once tokenized and TF-IDF processed, fared quite well at annotating test cases with feature labels. The textual description was less effective and interviews with developers confirmed it was less structured and less consistent than the path field.

We investigated how such a system could be deployed at an Android smartphone vendor and how much work developers would have to invest to make a working system. We found that by labeling 10% to 20% of test cases one could have acceptable feature labeling performance.

Acknowledgment. This research was partially supported by JSPS KAKENHI Grant Numbers JP18H03222 and an "JSPS Invitational Fellowship". Dr. Hindle is supported by an NSERC Discovery Grant. We thank the anonymous stakeholders at the studied company who participated in the survey. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of the company and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of the studied company's products.

REFERENCES

[1] Karan Aggarwal, , Finbarr Timbers, , Tanner Rutgers, , Abram Hindle, , Eleni Stroulia, , and Russell Greiner. 2017. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process* 29 (2017). Issue 3. e1821 smr.1821.

[2] Raymond P. L. Buse and Thomas Zimmermann. 2012. Information Needs for Software Development Analytics. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 10.

[3] C. Ebert and H. Soubra. 2014. Functional Size Estimation Technologies for Software Maintenance. *IEEE Software* 31, 6 (Nov 2014).

[4] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. 2012. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *2012 19th Working Conference on Reverse Engineering*.

[5] Abram Hindle, Christian Bird, Thomas Zimmermann, , and Nachiappan Nagappan. 2012. Relating Requirements to Implementation via Topic Analysis: Do Topics Extracted from Requirements Make Sense to Managers and Developers?. In *International Conference on Software Maintenance (ICSM 2012)*. IEEE.

[6] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. 2012. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *2012 34th International Conference on Software Engineering (ICSE)*.

[7] Stuart McIlroy, Nasir Ali, Hammad Khalid, and Ahmed E. Hassan. 2016. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering* 21, 3 (01 Jun 2016).

[8] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. 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*. ACM.

[9] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA.

[10] Miroslaw Staron, Wilhelm Meding, JÃűrgen Hansson, Christoffer HÃűglund, Kent Niesel, and Vilhelm Bergmann. 2014. Chapter 8 - Dashboards for Continuous Monitoring of Quality for Software Product under Development. In *Relating System Quality and Software Architecture*, Ivan Mistrik, Rami Bahsoon, Peter Eeles, Roshanak Roshandel, and Michael Stal (Eds.). Morgan Kaufmann, Boston.

[11] D. Suleiman, M. Alian, and A. Hudaib. 2017. A survey on prioritization regression testing test case. In *2017 8th International Conference on Information Technology (ICIT)*.

[12] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Empirical Software Engineering* 19, 1 (2014).

[13] C. Treude and M. A. Storey. 2010. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1.

[14] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012).

[15] Kovair Marketing, "White Paper - ALM and Integrated ALM", https://www.kovair.com/What-are-ALM-and-Integrated-ALM.pdf

[16] Radim Řehůřek and Petr Sojka, "Software Framework for Topic Modeling with Large Corpora", Proc. of the LREC Workshop, 45–50, 2010

[17] Pedregosa, F. et. al., "Scikit-learn: Machine Learning in Python", J. Mach. Learn. Res., 12, 2825–2830, 2011

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation", J. Mach. Learn. Res., 3, 993–1022, 2003