Research Article

Matej Madeja* and Jaroslav Porubän

Unit Under Test Identification Using Natural Language Processing Techniques

https://doi.org/10.1515/comp-2020-0150
Received Feb 24, 2020; accepted Apr 17, 2020

Abstract: Unit under test identification (UUT) is often difficult due to test smells, such as testing multiple UUTs in one test. Because the tests best reflect the current product specification they can be used to comprehend parts of the production code and the relationships between them. Because there is a similar vocabulary between the test and UUT, five NLP techniques were used on the source code of 5 popular Github projects in this paper. The collected results were compared with the manually identified UUTs. The tf-idf model achieved the best accuracy of 22% for a right UUT and 57% with a tolerance up to fifth place of manual identification. These results were obtained after preprocessing input documents with java keywords removal and word split. The tf-idf model achieved the best model training time and the index search takes within 1s per request, so it could be used in an Integrated Development Environment (IDE) as a support tool in the future. At the same time, it has been found that, for document preprocessing, word splitting improves accuracy best and removing java keywords has just a small improvement for tf-idf model results. Removing comments only slightly worsens the accuracy of Natural Language Processing (NLP) models. The best speed provided the word splitting with average 0.3s preprocessing time per all documents in a project.

Keywords: natural language processing, unit under test, program comprehension, automated identification, software maintenance

1 Introduction

In the source code the programmer expresses his or her thoughts that are based on the mental model [1]. When several programmers are working on a project they subconsciously share and interact with each other's mental model. The programmer uses mostly existing production code to understand the functionality and relationships between artifacts in the code, and then implements the necessary functionality based on this comprehension. The programmer thoughts are mainly expressed by following the naming conventions (e.g. Java Coding Style Guide [2]) which task is to simplify the representation of the code in the problem domain, e.g. by semantically correct naming of classes, methods, variables, etc. As Butler et al. mention in [3], names of identifiers have a significant impact on information mining from the source code and its comprehensiveness by the programmer.

The natural language processing (NLP) approaches [4] seek to reduce the barrier in computer-to-human communication. The NLP process involves the correct text analysis, then the determination of its semantics and the execution of the required action. Although the programmer is often constrained by the strict syntax of the programming language and the source code is a non-natural text, the best practices lead him or her to use parts of natural language conventions. A suitable preprocessing of the source code could be beneficial in using NLP techniques on the source code to identify related parts of the code.

Demeyer et al. [5] indicate that the best reflection of the source code are tests which must remain consistent with the source code during the whole product maintenance. Thanks to that developers can use tests as always up-to-date documentation to comprehend the production code. In our previous research [6] we focused on word frequency analysis between test and production classes in 5 popular Android projects on Github[1] and general testing practices. It was found that the words used by the programmers in the tests and the production code are from similar vocabu-

***Corresponding Author: Matej Madeja:** Department of computers and Informatics, Technical University of Košice, 042 00, Košice; Email: matej.madeja@tuke.sk
Jaroslav Porubän: Department of computers and Informatics, Technical University of Košice, 042 00, Košice; Email: jaroslav.poruban@tuke.sk

1 https://github.com/

lary set. Comparison of class and method names resulted in 49% inclusion of full name and even 76% of the partial name of a particular unit under test (UUT) in the test title. At the same time vocabulary used in test and production method bodies were very similar.

According to the above statements it is possible to assume that each source code file uses its own vocabulary depending on the problem domain and the functionality that it implements. Because the used words are not in the form of natural language sentences probably there don't exist semantic relations between particular words. On the other hand, each document contains a set of words that together characterize a whole, e.g. for Java it is common that a file represents a class.

McGlauflin [7] claims that in Java one production class should be tested by only one test class and the programmer is led to this convention also using an integrated development environment (IDE) tool. In this case it is possible to exactly identify UUT from the test and it can be assumed that the test and production classes will have similar vocabulary. However, this convention is often not respected in practice [6] so a general problem during the test code comprehension is the UUTs identification, especially when one test class tests multiple production classes.

This paper uses 5 natural language processing (NLP) models to help identify UUTs based on the vocabulary of the test, thereby simplifying program comprehension and preventing faults in the code. In this paper the following research questions are discussed:

RQ1: Are there large time differences between the NLP models when searching the index?
RQ2: Is there a general topics number for topic-based NLP models to process source code files without the need of searching it?
RQ3: How exactly can UUT be identified from the test class vocabulary?
RQ4: How to preprocess source code documents for model training to obtain the best results and what time is needed for the training?
RQ5: What time impact has different document preprocessing on the whole analysis process?

This research article is an extended version of a paper published in *2019 IEEE 15th International Scientific Conference on Informatics* [8] and extended with other NLP approaches on the same dataset. In Section 2 we briefly describe usage basics of selected NLP models to process natural text without mathematical details, since they are not important to the problem this paper deals with. Section 3 describes the programming language selection, used libraries and data preparation for processing. The results are described in Section 4 and at the end of the paper threats to validity, related work, conclusions and future work are discussed.

2 NLP models selection

All models used in this experiment are information retrieval (IR) algorithms that expect vectors as input, mainly because their nature is mathematical operations involving matrices. The input strings are therefore represented as vectors and this type of representation is called *Vector Space Model*. Based on these vectors a particular NLP model can make predictions. The aim of these algorithms is to train the model from the input data to minimize the occurrence of prediction errors.

There are different representations of text as a vector. The most straightforward representation is *bag-of-words* (BoW) and it is an orderless document representation, so only the counts of the words matter. This leads to loss of word order, syntactic relations, or morphology [9]. Despite that most IR algorithms the frequency of word occurrence is sufficient for calculation, in our case most of the input data except comments will not have the form of natural text, so usage of BoW and loosing the word order should not be critical. Therefore *bag-of-words* representation will be used to obtain term-document matrices for all models. In the following subsections selected models without mathematical details are described.

2.1 Latent Semantic Analysis

LSA is an indexing and IR method that uses *Singular Value Decomposition* (SVD) to identify relationships between words in an unstructured text. The model is based on the assumption that words used in the same context have a similar meaning [10]. By extracting terms from the document's body it seeks to create relationships between individual documents. It is important to choose a right number of topics to generate because if too many topics are requested for a short document the algorithm also returns words that should not determine the resulting topic of the document and vice versa.

2.2 Latent Dirichlet Allocation

The model is a generative probabilistic model of a corpus and considers each document as a set of topics which

characterize it. The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words [11]. Each topic consists of a set of words in a certain proportion. Based on the number of topics required the model attempts to rearrange the topics distribution within the documents to achieve the best composition. It is also very important to determine the right number of topics that the algorithm returns. Cvitanic et al. [12] discuss LSA and LDA model differences in more detail.

2.3 Term Frequency-Inverse Document Frequency

A slightly more sophisticated model is *tf-idf* which tries to encode two different kinds of information - term frequency and inverse document frequency [13]. It expresses a statistical measure used to evaluate how important a word is to a document in a collection or corpus. Term frequency (*tf*) is the number of times the word appears in a document. Since document length can differ a term would appear much more often in long documents than shorter ones. Therefore the term frequency is often divided by the document length as a form of normalization.

Inverse Document Frequency (*idf*) measures how important a term is. E.g. the word *class* can occur in documents representing a java file very often, so when evaluating similarity it is not so important as a word that occurred only in a small number of documents. Thus it is important to weigh down the frequent terms while scale up the rare ones. It is possible to extend this model, for example using topic models (like LSA or LDA).

2.4 Random Projections

RP model tries to reduce vector space dimensionality [14]. Using a fine randomness the approach approximates *tf-idf* distances between documents and thanks to reduced vector space dimensions it is a very memory- and CPU-friendly model. The model is similar to LSA but it can reduce the vector space model to lower dimensions and in this way reduce the cost of computing resources with similar accuracy. More about random indexing of text samples can be seen in the article [15] written by Kanerva et al.

2.5 Hierarchical Dirichlet Process

The HDP model is a non-parametric Bayesian method using the Dirichlet distribution in the same way as the LDA. For LDA it is necessary to correctly estimate the number of topics by document size which can greatly affect the results. In HDP during a document collection, posterior inference is used to determine the number of topics needed and to characterize their distributions. It was proven by Wang et al. that HDP improved performance over LDA topic model [16].

3 Method

The experiment was conducted on 5 popular Android projects from our previous research [6]. In order to know the success rate of particular NLP models in identifying a UUT from a test it is necessary to establish a link between the test and the production classes. Since we performed a manual analysis of 617 tests in [6] we can partially use the collected data for this experiment. We assume that manually created links are correct. The source codes of considered projects (see Table 1) are from February 2019 to preserve consistency with manually collected data. The projects were selected on the assumption that the most popular projects will include tests (see more in [6]).

Table 1 shows that in most cases the convention that one UUT (production class) is tested by one test class has been fulfilled, so in 125 cases we can clearly establish the expected connection between the test and production class. For tests that test multiple production classes the most tested production class will be considered as correct UUT.

3.1 Programming language and library

For projects' source code analysis we chose Python language which is great for processing computationally difficult tasks. Also the availability of *gensim* [17] library for this language was crucial. The library is very popular in the NLP field and according to its author Řehůřek [18] *gensim* is the most robust, efficient and hassle-free piece of software to realize unsupervised semantic modeling from plain text.

Table 1: General stats of manually analyzed data. [6]

Project	Production classes	Production methods	Test classes	Test methods
plaid	37	71	39	180
ExoPlayer	49	98	53	323
Android-Clean Architecture	17	22	17	29
shadowsocks-android	6	7	6	8
iosched	16	40	16	77
SUM	**125**	**238**	**131**	**617**

3.2 Documents preparation

All analyzed projects are built on the Android platform and implemented in Java and/or Kotlin. For each project we recursively searched for files having the `.java` or `.kt` extension. Kotlin is designed to interoperate fully with Java so they use similar programming conventions and it is also suitable for our analysis. Of course, only the files of the project were included in the analysis, i.e. without dependencies and platform software development kit files. We already knew the names of the test classes from [6] so we divided the particular files into test and production set. The content of production classes served for model training and content of test classes for searching similarity and UUT detection.

File preprocessing was the same for both test and production classes. From the content of each file new line characters have been removed and the result was saved in a project-depended training file. One line in this file represented one document for further processing.

Since the result is highly dependent on the quality of the input data and is governed by the idiom *"garbage in, garbage out"*, it is very difficult to assume in a non-natural text how it should be properly preprocessed. To find out how to prepare input documents derived from the source code in the best way (RQ3), we incrementally created 5 versions of document preprocessing:

1. *Full version* - original file version, removed only `\n` chars.
2. *Word split* - all `camelCase` or `snake_case` words has been split. Words out of base conventions, such as `ORMLite`, remained unsplit.
3. *Removed Java keywords* - all Java keywords have been discarded.
4. *Removed comments* - multi- and one-line comments discarded.
5. *Removed imports* - all Java imports removed.

Incremental preprocessing means that, for example in the 4^{th} iteration, both *Word split* and *Removed Java keywords* have been included. Another preprocessing of documents that applied to all iterations was the removal of frequently occurring English words using *nltk* library, such as *and*, *a*, *the*, etc. At the same time stemming over the documents has been executed, where inflected or sometimes derived words have been reduced to their word stem, e.g. *cars* to *car*. The last step was to remove words that occurred only once in the corpus of training documents to eliminate their negative impact on results.

3.3 Model training

To train the LSA model only the necessary parameters were supplied - number of topics, dictionary (BoW) and corpus (of vectors). For the LDA model training we also set the *alpha = auto* parameter which means the model learns asymmetric prior from provided corpus. From the *alpha* attribute LDA model computes *theta* that decides how the topic distribution is drawn. The last special parameter set for the LDA model was *passes = 20* which expresses the number of passes through the corpus during training. In terms of statistics, more training means statistically more accurate results.

For tf-idf, RP and HDP models corpus and BoW were provided. It was also possible to define the topics number for the RP model, but the default value was retained. For tf-idf the parameter *normalize = True* has been set to obtain better results from the model (explained in Section 2.3). Creating a *bag-of-words* representation for all models in the form of a dictionary (*id + word* pair) and creating a corpus of sparse vectors was relatively easy using the functions of *gensim*.

3.4 Estimating number of topics

Evaluation of the quality of LSA and LDA models, which is significant for results of both considered models, can be determined by topic coherence that is a measure used to evaluate topic models. The topic coherence is applied

Table 2: General document statistics in analyzed projects.

#	Project	Number of files		Mean search time in index (1 query)				
		prod.	tests	LSA	LDA	tf-idf	RP	HDP
1	plaid	679	39	49.73 ms	43.97 ms	280.85 ms	8919.26 ms	2710.2 ms
2	ExoPlayer	954	53	78.52 ms	88.33 ms	498.8 ms	13379.12 ms	5151.18 ms
3	Android-Clean Architecture	99	17	7.65 ms	3.53 ms	20.27 ms	1197.31 ms	283.69 ms
4	shadowsocks-android	157	6	11.67 ms	10.00 ms	68.7 ms	1924.79 ms	820.6 ms
5	iosched	332	16	18.89 ms	13.33 ms	78.58 ms	3986.66 ms	1205.59 ms

to the top N words from the topic and it is defined as the average/median of the pairwise word-similarity scores of the words in the topic [19]. A good model will generate coherent topics with high coherence scores. Good topics are those that can be described by a short label. Since the number of topics is dependent on the nature of the documents, we searched for the highest coherence value for each iteration and project.

Training a large number of models is a very time consuming task so in the early stages of the experiment we tried to obtain an approximate range to try in. After multiple tests we decided to calculate coherence values from 7 to 50 for each model, project and iteration, and the model with the highest value has been chosen for the analysis. Estimating the best number of topics was executed only for LSA and LDA models.

3.5 Evaluation of document similarity

When using different NLP models we focused on a single aspect of possible similarities, i.e. on apparent semantic relatedness of their texts (words), just a semantic extension over the boolean keyword match. Modern search engines also take into account random-walk static ranks, hyperlinks, etc. *Gensim* basically uses *cosine similarity* [20] to determine the similarity of two vectors and it is a standard measure in *Vector Space Modeling*.

Every single search query was made up of the content of a test class. We created *bag-of-words* for each document (test body) and converted it to the corresponding NLP model space. Subsequently, an index has been created from the trained model against which the query was evaluated. Similarities to all production classes were calculated and we obtained the result as $(document\_no, similarity\_value)$ pairs where $similarity\_value \in \langle -1, 1 \rangle$. The greater *similarity_value* the more similar document. Every *document_no* has been paired with the value stored in a relation database created during training and document preparation to identify a particular production class.

4 Results

Altogether we analyzed 2221 production and 168 test classes in five projects (see Table 2). In five iterations of document preprocessing a total of 2,742,100 similarity results between the tests and the production source code have been obtained.

4.1 Search speed

The time of searching/comparing a document similarity in the index is very important, especially when a realtime response is expected. NLP methods could be part of an integrated development environment (IDE), e.g. by displaying tooltips to navigate from test to related parts of the code or UUT, it is desirable that the search is fast enough. In the Table 2 statistics of mean search time in the prepared index are presented to help answer the **RQ1**.

As can be seen, the number of documents in the index has a big influence on the search speed, as the size of the matrix increases proportionally with the number of documents and thus the search response. It is interesting to observe an increase in search time for the tf-idf, RP and HDP models. For example, tf-idf search was 6 times and RP up to 185 times slower in average than LDA/LSA one. From the experiment the most powerful search model is LDA with an average time of 31.82 ms. For real-life usage the responses of the LSA and tf-idf models are also acceptable and could be used in the IDE where further research following this experiment should be directed. According to Tolia et al. [21] response times below 150 milliseconds do not impact user productivity and experience, but due to

the complexity of this issue, which is related to thousand of source code lines, times are sufficient (compared to full project search on indexed files in an IDE).

4.2 Optimal topic number

Model training lasted the longest time especially because of searching for the optimal topic number. The tests were performed on a 12-core *Intel Core i7-5820K* CPU with 12GB RAM and *Debian GNU/Linux 9* installed. Nevertheless, analysis was performed in one thread only to simulate the use in a real environment, e.g. in the background of an IDE. The average search times for the highest coherence value can be seen in Table 3.

To answer **RQ2** the Table 4 was created. As can be seen the mean value for LSA is relatively stable. Although the maximum deviation of the LSA value is 40, this situation occurred in only one case, i.e. it was just an exception that could be ignored. For the source code analyses using the LSA model it is therefore possible to use a relatively stable value of topic numbers in the range 7–10.

When using LDA the selected topic numbers were more diverse as the model is less stable. This can be obtained in multiple model training with the same data when the results vary slightly. That's why topic numbers are more diverse than in LSA. Most often the topic number for a project was in similar values and in the difference range of 10 units. However, the differences between the particular projects were large and based on this data it is not possible to determine the recommended topic number for LDA.

Table 3: Average times of searching best coherence values per project in minutes.

Project #	1	2	3	4	5
LSA	3.96m	4.46m	0.97m	2.03m	1.36m
LDA	21.31m	32.66m	2.26m	6.84m	7.18m

- to identify particular project pair with Table 2.

Table 4: Statistics of LDA and LSA topic models coherence value calculation.

Subject	Metric	Model	Project # 1	2	3	4	5
Number of topics	**mode**	**LSA**	7	7	7	10	10
		LDA	-	-	14	12	19
Number of topics difference	**min**	**LSA**	0	0	0	0	0
		LDA	1	1	0	0	0
	max	**LSA**	1	6	40	6	5
		LDA	11	20	25	17	23

- to identify particular project pair with Table 2.

4.3 Training times

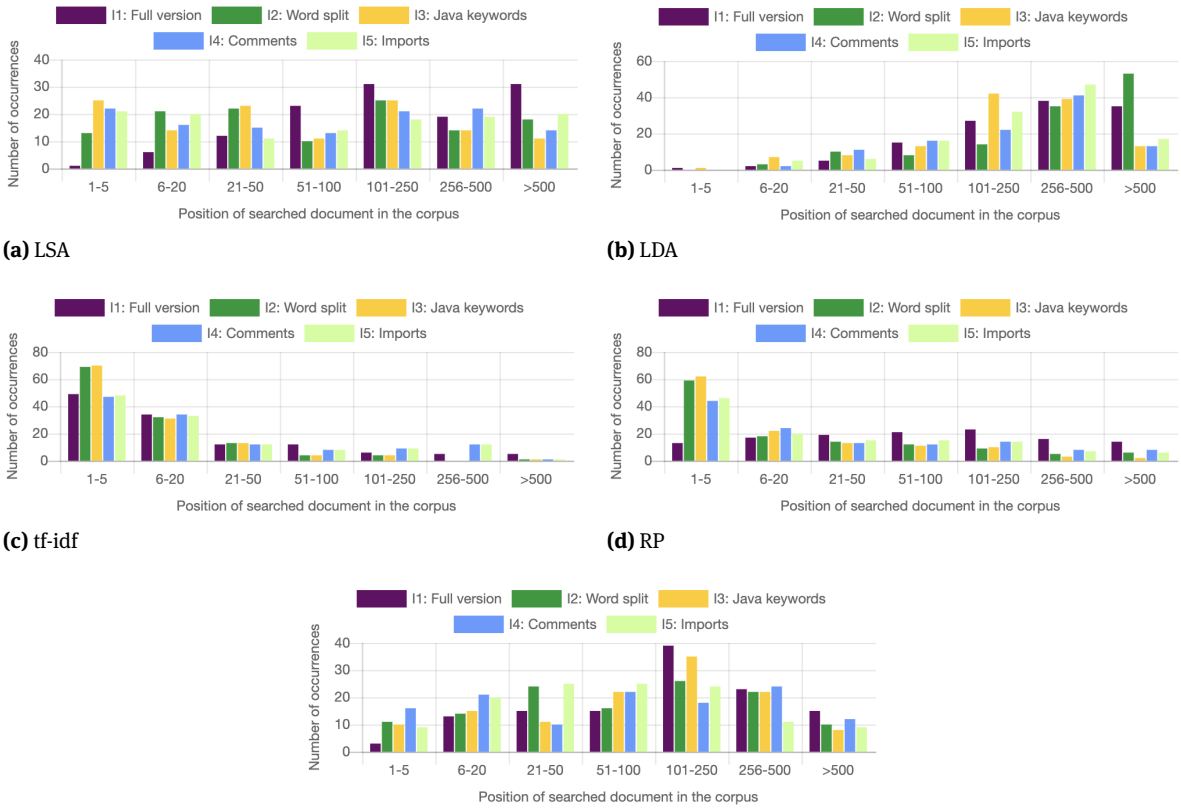
The differences between the models' training times for the different projects were minimal, i.e. a few milliseconds. Larger differences (in seconds) were noted for LDA and HDP models, indicating the complexity of using dirichlet allocation. The fastest average training time of 8ms was achieved by the tf-idf model. The greatest decrease of training time in a particular NLP model was recorded in the 4^{th} iteration when comments were removed. By removing them a lot of the training data have been lost so finding the best coherence value in the 4^{th} iteration was faster: 2.17 times for LSA, 2.83 times for LDA and 2.5 times for other models. Despite the increased speed a large amount of potentially natural text in the source code has been lost and as pointed out later (Section 4.5) the loss of this data negatively affects the results.

4.4 Accuracy of UUT identification

Since we assume that manually identified UUTs are correct it is possible to determine the accuracy of a particular model based on the order of production class in the search result. The Figure 1 shows the frequencies in the search queries for manually labeled production classes as UUT and the presented frequencies are generalized for all projects.

First let's discuss the results of LDA and LSA models which are similar from the viewpoint of finding the best topic number for the *gensim* library. As can be seen, LSA performed much better than LDA. The LSA is based on the frequency of words in the documents and as was found in [6], the words between the test and UUT are very similar which positively influenced the result. The accuracy of the LDA model was very low, in the first five results the correct UUT appeared only 2 times. Although the LSA achieved 82 correct UUTs in the first five results for all iterations it is still only 13% success rate which is considerably inadequate! and only 5% of UUTs were marked absolutely correctly in all iterations and solely by the LSA method.

The if-idf model was the most accurate because it marked 46% of UUTs classes on the top five positions and 22% on the first position in average, independently of preprocessing iteration. This result can be considered as excellent compared to the original results presented in [8]. Taking into account also the preprocessing of the input data, it can be seen that in 3^{rd} iteration up to 40 UUTs have been accurately labeled, i.e. 33% success rate of rightly marked UUTs. Similarly good results were achieved by the RP model with a success rate of 36% for the first 5 positions



(a) LSA **(b)** LDA

(c) tf-idf **(d)** RP

(e) HDP

Figure 1: Position frequency of manually identified UUT for all models.

Table 5: Train times of all models per particular project.

Subject	Metric	Model	Project # 1	2	3	4	5
Final model train time (s)	min	LSA	0.094	0.140	0.016	0.027	0.028
		LDA	8.921	11.273	1.001	2.877	3.825
		tf-idf	0.008	0.014	0.001	0.002	0.002
		RP	0.022	0.043	0.003	0.005	0.004
		HDP	1.598	2.100	0.197	0.368	0.641
	avg	LSA	0.140	0.192	0.018	0.038	0.054
		LDA	12.351	24.306	1.292	4.265	5.298
		tf-idf	0.012	0.022	0.002	0.004	0.003
		RP	0.031	0.062	0.003	0.007	0.006
		HDP	1.970	2.989	0.234	0.436	0.820
	max	LSA	0.202	0.302	0.020	0.050	0.079
		LDA	17.856	44.739	1.503	8.367	7.234
		tf-idf	0.018	0.035	0.003	0.007	0.005
		RP	0.050	0.117	0.004	0.011	0.010
		HDP	2.266	3.910	0.282	0.517	0.937

- to identify particular project pair with Table 2.

Table 6: Position frequency for first 5 positions of analyzed models.

	Model	Position in the search	Iteration I1	I2	I3	I4	I5	Σ
Position frequency in the search	LSA	1		4	10	9	9	32
		2	1	1	8	6	6	22
		3		3	2	4	2	11
		4		5	4	1	3	13
		5			1	2	1	4
		Σ	1	13	25	22	21	
	LDA	1						0
		2						0
		3	1		1			2
		4						0
		5						0
		Σ	1	0	1	0	0	
	tf-idf	1	18	39	40	19	19	135
		2	12	13	12	14	14	65
		3	11	13	10	9	9	52
		4	3	2	5	4	4	18
		5	5	2	3	1	2	13
		Σ	49	69	70	47	48	
	RP	1	2	34	28	17	15	96
		2	7	10	9	16	9	51
		3	2	7	14	3	6	32
		4	2	6	6	2	9	25
		5		2	5	6	7	20
		Σ	13	59	62	44	46	
	HDP	1	2	5	4	8	5	24
		2			1	5	3	9
		3		2	2	3		7
		4	1	3	2			6
		5		1	1		1	3
		Σ	3	11	10	16	9	

and exact identification of 16%, but it is relatively inaccurate compared to tf-idf. The HDP model achieved only 8% detection in the first 5 places and 4% in the exact identification of UUT, but this result is also influenced by the fact that HDP is still rough around its academic edges.

In response to **RQ3**, our results indicate that the most accurate model for use on the source code is tf-idf one. As mentioned in the previous section, the search speed in the index is sufficient, so this model seems to be the most suitable for real use. In the results it is necessary to take into account the fact that for the 6 test classes, which tested multiple production classes at once, the most tested class was chosen as the correct UUT (discussed in Section 3).

4.5 Data preparation

A more detailed look at the best search results in each iteration is needed to respond to the **RQ4** (see Table 6). As can be seen, *word split* and *removal of java keywords* (I2 + I3) has the greatest impact on the accuracy of the results through all used NLP models. Our expectation was that when comments are removed the results will get worse because there is a potential for sole natural language in the comments. In Table 6 it can be seen that removing comments (I4) and imports (I5) had a significant impact on accuracy only for the tf-idf model.

It also shows that the meaning in the code is most often expressed directly in the names of the identifiers, i.e. class, methods and variables names. On the other hand,

Table 7: Average times of document preprocessing for all 5 projects per particular iteration.

Iteration	I1	I2	I3	I4	I5
Time (s)	0.719	1.548	24.465	194.432	193.137

the removal of imports (I5) did not have a big influence on the accuracy of the results, except in some rare cases. The assumption was that the test and production class would also use similar imports and this similarity could help identify UUT, but this was not shown to be crucial in this experiment. Using word splitting (I3) generally gave rise to the biggest increase in precision of methods and removing the java keywords (I4) didn't have so much impact on accuracy, compared to I3. At the same time 3rd iteration was one of the fastest ones in terms of finding the best coherence value and searching in the index.

RQ5 deals with a very important aspect for real IDE usage and that is the duration of document preprocessing (see Table 7). As can be seen, there is a significant difference in preprocessing duration between I2 and I3. For the most accurate model tf-idf we have already achieved very good results during I2 and preprocessing time was remarkably short (note that the times in the table are for all 5 projects). In terms of the accuracy, training times, speed of preprocessing and searching the index the 2nd iteration with the tf-idf model is the best combination.

4.6 Detected errors of manual identification

In the experiment, we unexpectedly found that 8 classes manually labeled as UUT were not in the corpus. Manual testing was done in *Android Studio* IDE where we used references created directly by IDE. After a more detailed analysis we found that production classes that were not in the corpus were incorrectly labeled. Incorrect identification was due to references to generated source code that were not present in the file system of the project without run. The method can be therefore used to prevent such errors.

5 Threats to validity

Comparison the accuracy of LSA and LDA was relied on the fact that the manual identification of UUTs was performed correctly. If an error in manual identification happened this could have a negative effect on the results reported in this paper. The analysis was performed on only 5 popular Android projects and no other projects were included, e.g. less popular, proprietary, etc. At the same time, projects in other languages have not been analyzed and a particular language can affect the corpus of words by using different naming conventions or language syntax.

While the LDA method is more accurate than the LSA (a claim from the official description of the method, not from our results), this method also shows slight differences in document comparisons when training the LDA model multiple times with the same data, indicating some inaccuracy still statistically friendly.

The preprocessing of documents (the *garbage in, garbage out* idiom) has also a huge impact on the results. How to prepare source codes for such analysis was also one of the research questions of this paper. There exists a threat to validity because not all possible document preparation could be tried out.

As mentioned in Section 2 the choice of topics number also has a big impact on the accuracy of the methods. Despite finding the best value for this parameter a search range of 7 to 50 may not be sufficient. Also the use of coherence value may not be reliable at all times and there is no general recommendation on how to accurately determine this parameter, so there is no guarantee that the best values have been chosen with respect to the input data.

6 Related work

The most similar research on improving program comprehension was done by Maletic et al. [22, 23]. In their conclusions they argue that the LSA model can assist in supporting some of the activities of the program's comprehension process. However, they only analyzed one project in C, in our case a larger sample and Java and/or Kotlin languages are considered. They analyzed 269 files, we analyzed a sample of 2221 files in all projects without 168 test files used as queries on index. They created code clusters of similar files trying to make it easier for the programmer to find related parts of the program. In our case we focused on the relationship between the test and the production class which can even be written in another language (e.g. tests in Java, production code in Kotlin; see *plaid* project) and assumed that the UUT test will have more common vocabulary as 2 different classes.

Another type of research was performed by Thomas et al. [24, 25] in 2014 who mined software repositories using topic models to simplify the understanding of software changes during software evolution, especially for stakeholders. Although their experiments have not been veri-

fied on real stakeholders the results show that extraction of topics is sufficient and should therefore have a positive impact on simplification of understanding. In our research we focused more on developers, analyzing the source code and relationships within it.

Asuncion et al. [26] proposed an automated technique that combines traceability with topic modeling. They record traceability links during the software development process and learn a probabilistic topic model over different artifacts. From collected data they are able to categorize artifacts and create topical visualisations of a particular system. They implemented several tools that support data collecting during software evolution. In our case, we are still in the early stages, so we found out whether it is beneficial to look for similarities between the test and the production code using NLP techniques.

7 Conclusions and future work

This paper presented use of 5 NLP techniques on source code of 5 popular Android Github projects. As previous research has shown there is similarity between production and testing code by looking at the vocabulary used in these software artifacts. Since the programmer expresses his thoughts in the code and is natural for him/her to think in natural language, it is assumed that even in the source code he/she will use the identifiers and comments to express the meaning in a natural way. Using NLP techniques it could theoretically be possible to clearly identify UUTs based on the similarity between the test and the production class, and in the future to enrich the source code to support program comprehension.

A particular model index was created from the source code of production classes, which was compared to the similarity of the test class content. A total of 2,742,100 similarity results between tests and production source code have been obtained and the analysis included 131 test and 2221 production classes. The main objective was to find out whether the use of NLP techniques on the project source code can accurately identify the UUT and whether it is possible to achieve better results by appropriate preparation of input documents.

The documents used to train the models were preprocessed in five different iterations and the following NLP methods were used for each iteration: Latent Semantic Analysis, Latent Dirichlet Allocation, Term Frequency-Inverse Document Frequency, Random Projections and Hierarchical Dirichlet Process. It was found that LSA and LDA search times were at least 6 times faster than other methods. The reason of this issue has not been investigated, but using DP and HDP methods often took several seconds to execute the search. During experimenting the best topic numbers were obtained for the LSA and LDA models. For LSA it was possible to identify recommended topics number in range 7-10, for LDA it was not possible.

Another area of interest was how long it takes to train models for different projects. The fastest average training time of 8ms was achieved by the tf-idf model and the second fastest time was achieved by the LSA model. Also identified was the complexity of the dirichlet distribution calculation in LDA and HDP, where the training times were mostly in seconds. The accuracy of the UUT identification by an NLP approach was evaluated on the basis of previous research in which UUTs were manually labeled for all analyzed projects. The most accurate results were achieved by the normalized tf-idf model with a 22% average accuracy of determining the correct UUT regardless of document preprocessing. Taking into account also the preprocessing of input documents the tf-idf model achieved the best results in the 3rd iteration, i.e. after word splitting and removing java keywords. Accuracy of the I3 was up to 33% which is a very significant result compared to other models and previous experiments. Accepting a small deviation tolerance tf-idf model found 57% of UUTs in the I3 with accuracy up to first 5 places. The model has proven the most reliable in terms of accuracy, search and training speed. Its combination with other methods could help the support program comprehension in the future.

At the same time, the paper focused on methods of suitable preprocessing of documents to achieve the best results. Words split (I2) had the greatest impact on the accuracy of all methods but the best results were achieved after removing java keywords (I3) in general. This shows the negative impact of frequently occurred words in NLP methods that don't use normalization and only take into account the frequency of words, and of course, the occurrence of java keywords was high in the source code files. On the other hand, it was observed that removing the comments slightly worsened the results. Initially the higher impact of comments removal was expected, as the comments contain the most natural part of the code. It turned out that comments do not have as much impact on accuracy as originally expected by the authors. Taking into account the time needed to preprocess the data, it seems 2nd iteration is most suitable for use with the tf-idf model with an average preprocessing time of 0.31s per project.

The results of the paper show that usage of NLP methods has potential to support program comprehension. Although the maximum accuracy of 33% is not very reliable, by combining it with other methods (e.g. static code anal-

ysis or co-evolution observation) it will likely be possible to determine the UUT exactly, which will be the subject of our further research. This experiment was performed only on a sample of Android projects, so the source code could be influenced by the platform. In the future, it would be useful to verify accuracy for other programming languages and platforms and compare the accuracy of these methods in order to generalize the results as much as possible.It would also be advisable to perform accuracy tests for other known techniques of UUT identification, compare them and try to design a suitable and less time-consuming combination to exactly identify UUT.

Acknowledgement: This work was supported by project VEGA No. 1/0762/19: Interactive pattern-driven language development.

References

[1] Détienne F., What model(s) for program understanding?, 2007
[2] Reddy A., et al., Java™ coding style guide, Sun MicroSystems, 2000
[3] Butler S., Wermelinger M., Yu Y., Sharp H., Mining java class naming conventions, in 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, 93–102, 10.1109/ICSM.2011.6080776
[4] Manning C.D., Manning C.D., Schütze H., Foundations of statistical natural language processing, MIT press, 1999
[5] Beck K., Gamma E., Test infected: Programmers love writing tests, Java Report, 3(7), 1998, 37–50
[6] Madeja M., Porubän J., Tracing naming semantics in unit tests of popular GitHub android projects, volume 74, 2019, 10.4230/OASIcs.SLATE.2019.3
[7] McGlauflin B., Java Unit Testing Best Practices: How to Get the Most Out of Your Test Automation, DZone Technical Library, 2019
[8] Madeja M., Porubän J., Accuracy of Unit Under Test Identification Using Latent Semantic Analysis and Latent Dirichlet Allocation, in A.S. Valerie Novitzká Štefan Korečko, ed., Informatics 2019, Institute of Electrical and Electronics Engineers, 2019, 248 – 253
[9] Croft W.B., Metzler D., Strohman T., Search engines: Information retrieval in practice, volume 520, Addison-Wesley Reading, 2010
[10] Deerwester S., Dumais S.T., Furnas G.W., Landauer T.K., Harshman R., Indexing by latent semantic analysis, Journal of the American society for information science, 41(6), 1990, 391–407
[11] Blei D.M., Ng A.Y., Jordan M.I., Latent dirichlet allocation, Journal of machine Learning research, 3(Jan), 2003, 993–1022
[12] Cvitanic T., Lee B., Song H.I., Fu K., Rosen D., Lda v. lsa: A comparison of two computational text analysis tools for the functional categorization of patents, in International Conference on Case-Based Reasoning, 2016
[13] Hiemstra D., A probabilistic justification for using tf×idf term weighting in information retrieval, International Journal on Digital Libraries, 3(2), 2000, 131–139, 10.1007/s007999900025
[14] Bingham E., Mannila H., Random Projection in Dimensionality Reduction: Applications to Image and Text Data, in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, Association for Computing Machinery, New York, NY, USA, 2001, 245–250, 10.1145/502512.502546
[15] Kanerva P., Kristoferson J., Holst A., Random indexing of text samples for latent semantic analysis, in Proceedings of the Annual Meeting of the Cognitive Science Society, volume 22, 2000
[16] Wang C., Paisley J., Blei D., Online variational inference for the hierarchical Dirichlet process, in Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 2011, 752–760
[17] Řehůřek R., Sojka P., Software Framework for Topic Modelling with Large Corpora, in Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, Valletta, Malta, 2010, 45–50, http://is.muni.cz/publication/884893/en
[18] Řehůřek R., About Gensim, 2019
[19] Lau J.H., Newman D., Baldwin T., Machine reading tea leaves: Automatically evaluating topic coherence and topic model quality, in Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, 2014, 530–539
[20] Huang A., Similarity measures for text document clustering, in Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand, volume 4, 2008, 9–56
[21] Tolia N., Andersen D.G., Satyanarayanan M., Quantifying interactive user experience on thin clients, Computer, 39(3), 2006, 46–52
[22] Maletic J.I., Marcus A., Using latent semantic analysis to identify similarities in source code to support program understanding, in Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000, 2000, 46–53, 10.1109/TAI.2000.889845
[23] Maletic J.I., Valluri N., Automatic software clustering via latent semantic analysis, in 14th IEEE International Conference on Automated Software Engineering, IEEE, 1999, 251–254
[24] Thomas S.W., Adams B., Hassan A.E., Blostein D., Studying software evolution using topic models, Science of Computer Programming, 80, 2014, 457–479
[25] Thomas S.W., Mining software repositories using topic models, in Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, 1138–1139
[26] Asuncion H.U., Asuncion A.U., Taylor R.N., Software traceability with topic modeling, in 2010 ACM/IEEE 32nd International Conference on Software Engineering, volume 1, IEEE, 2010, 95–104