A Mechanism for Automatically Summarizing Software Functionality from Source Code

Christos Psarras, Themistoklis Diamantopoulos and Andreas Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
cpsarrac@ece.auth.gr, thdiaman@issel.ee.auth.gr, asymeon@eng.auth.gr

***Abstract*—When developers search online to find software components to reuse, they usually first need to understand the container projects/libraries, and subsequently identify the required functionality. Several approaches identify and summarize the offerings of projects from their source code, however they often require that the developer has knowledge of the underlying topic modeling techniques; they do not provide a mechanism for tuning the number of topics, and they offer no control over the top terms for each topic. In this work, we use a vectorizer to extract information from variable/method names and comments, and apply Latent Dirichlet Allocation to cluster the source code files of a project into different semantic topics. The number of topics is optimized based on their purity with respect to project packages, while topic categories are constructed to provide further intuition and Stack Exchange tags are used to express the topics in more abstract terms.**

***Index Terms*—program understanding, topic modeling, reverse engineering, software reuse**

I. Introduction

Nowadays, software development requires efficiency and agility to cope with the ever increasing demand for new features. The evolution of the Internet and the introduction of online open-source repositories have led to the establishment of an effective component-based software engineering paradigm. In this context, developers are encouraged to reuse software components that cover parts of the required functionality, in order to reduce the time and effort required for software development. This process typically requires identifying the components to be reused, understanding their functionality and integrating them in their own source code.

The most commonly reused software components are found in libraries, which essentially are collections of components that specialize in a certain area of functionality. Libraries provide efficient implementations of algorithms, as well as frameworks to better assist the development process. Reusing libraries with well-defined APIs (*black-box reuse*) has been proven to improve the overall quality of the software product and reduce costs related to software development and maintenance [1]. On the other hand, the interest for integrating and extending library source code in a per-component basis (*white-box reuse*) has also lately increased [1], [2].

The first step towards effectively extending or reusing third-party source code involves understanding the offered functionality as well as the software architecture followed. However, current software projects (including libraries) are often not well-documented (if at all), thus hindering program comprehension both for reverse engineering and for reuse purposes. These challenges are also present after building a software product, as maintaining it may require considerable effort; notably, more than half of the maintenance effort is spent for program comprehension [3].

As a result, lately several research approaches aim to assist in program understanding using semantic clustering techniques [4]–[8]. These techniques use the source code of a software project as input and initially extract semantic information from source code elements, including variable names, comments, etc. After that, the extracted information is organized at package, class and/or method level, and clustering techniques are applied to identify categories (clusters) of components that are semantically similar.

Although these approaches can be effective under certain scenarios, they also have several drawbacks. At first, the underlying techniques depend on the proper selection of parameters, and tuning these parameters for each individual project or library can pose an important challenge to the developer. Furthermore, the identified categories are not clearly named and defined from a semantics perspective, therefore providing the developer with a source code categorization that lacks an abstract summarization for each category. Lastly, most approaches often produce a large number of topics, which may be hard to follow, while there are also multiple overlaps between the different topics, further perplexing the categorization.

In this paper, we create a system that analyzes the source code of a given library, extracts useful information from variable/method names and comments, and identifies semantic topics. Upon performing vectorization, our system employs clustering to produce a set of topics summarizing the source code. We employ a semantic clustering algorithm that is optimized based on the purity score of the extracted topics, thus alleviating the need for complex parameter assignment by the developer, while allowing him/her to easily tune the algorithm if desired. Furthermore, we aid the developer in this decision by employing postprocessing techniques to further merge the extracted topics into categories. Finally, our approach uses information from online sources to semantically enrich the topics, by annotating them with keywords that refer to a more abstract description of the functionality they represent.

The rest of this paper is organized as follows. Section II provides a literature review of the methodologies used for topic

extraction from source code and highlights the offerings of our system. The architecture of our system is presented in Section III along with an analysis of its modules. Section IV describes our evaluation framework and presents the results for our methodology as well as a case study for a software project. Finally, Section V summarizes our work and provides useful insight for future research.

II. Related Work

Program comprehension is one of the most important areas of Reverse Engineering and Software Reuse, as it constitutes the first step towards several actions of the developer. Obviously, a developer has to understand the offered functionality and the underlying semantics of a software project, as well as its structure, before being able to extend it, reuse it or even maintain it. For projects with minimal or no documentation, this can be quite a challenging task, as the developer has no option but to read the source code itself. Thus, lately several researchers have developed approaches for extracting semantic information from software engineering data in order to recover traceability links between documentation and source code, locate features in source code, and automatically label/group software components [4]. In the context of this work, we focus on approaches that extract topics from source code[1] in order to group software components from a functional perspective.

Topic$_{XP}$ [6] is an Eclipse plugin, aimed at assisting the developer in identifying the functionality provided by a given software project. It requires as input the source code of the project under analysis and extracts information from variable names, method names, and comments. Each Java class is considered as a document, following a bag of words representation. *Latent Dirichlet Allocation (LDA)* is then used on the documents in order to identify a set of topics that describe the different functionalities offered by the project. Java classes are associated with one or more topics, while the cohesion and separation values among classes are computed based on the similarity or dissimilarity of the topics they belong to. The main disadvantage of this approach is that its effectiveness relies heavily on the LDA parameters selected by the user, and especially the number of topics. Setting these parameters requires clear understanding of their effect and of the intrinsic features of the source code under analysis.

Linstead et al. [7] circumvented the parameter tuning step required by LDA by solving a broader type of problem. Instead of analyzing a single project or library, they created a system that receives as input a collection of software projects and applies LDA at class level. The system aims at identifying classes with similar functionality and relies on the size of the dataset to set the LDA parameters to fixed values, deemed consistent enough for a large collection of software components. While this approach alleviates the user from selecting the proper LDA parameters, it still lacks an effective tuning mechanism, thus restricting its applicability on different projects.

[1]See [5] for an extended review of approaches that use different types of information, including emails, logs, bug reports, etc.

Another quite interesting approach was proposed by Kuhn et al. [8]. The authors used *Latent Semantic Indexing (LSI)* to reduce the features (terms) describing each document and applied an average linkage hierarchical clustering algorithm on the resulting classes-documents. Each topic is described by the top occurring words of a collection of documents, which are clustered together. The result of the hierarchical clustering is then compared to the hierarchical package structure of the software project, so that each generated topic is evaluated for its distribution on one or more packages. The authors define distribution categories for topics spread over a single package (well-encapsulated), topics covering more than one packages either with or without a dominant package (cross-cutting or octopus-like respectively), and topics with very few classes (black sheep). Although postprocessing is one of the strong points of the approach, there is again no parameter tuning (e.g. for the hierarchical clustering), while the number of the final clusters is fixed. Furthermore, the topics are given along with the top words, and not further annotated as to the functionality they describe.

Lately several researchers have also proposed methods for improving the effectiveness of existing techniques. Nie and Zhang [9] attempted to optimize LDA-based feature location using topic cohesion and coupling as computed by a software dependency network, while Lawrie et al. [10] proposed enriching the vocabulary of the source code before extracting the topics, e.g. by expanding the acronyms in identifier names. These approaches, however, still do not support the developer for parameter tuning, while the topics are not postprocessed and thus do not output an intuitive semantic categorization.

Apart from the aforementioned approaches, several efforts on the area of topic modeling from source code have involved using human assistance. Indicatively, Maskeri et al. [11] focused on extracting business topics and implementation topics that can be subsequently refined manually, while Saeidi et al. [12] constructed ITMViz, a tool that accepts feedback from developers and architects in order to refine the extracted topics. There are also systems aimed at automatic software categorization, such as MUDABlue [13] or LACT [14], which use topic modeling not to distinguish within a project but to define the high-level category of the project/library as a whole. Though interesting, these approaches deviate from the scope of this work, since they confront different challenges and operate on different granularity levels (project-level instead of class/package-level). Finally, there are interesting works in the broad area of semantics for source code, including e.g. approaches for generating method names [15] or comments [16]. However, these are applied at method level and deviate from the topic modeling approaches analyzed in this work.

The reviewed approaches can be effective for certain cases. However most of them require some type of manual intervention by the developer, either for setting the parameters of the algorithms [6], [8] or for assessing and possibly refining the resulting topics [11], [12]. Furthermore, current approaches, with the exception of the approach of Kuhn et al. [8], do not perform sufficient postprocessing, resulting in a large unman-

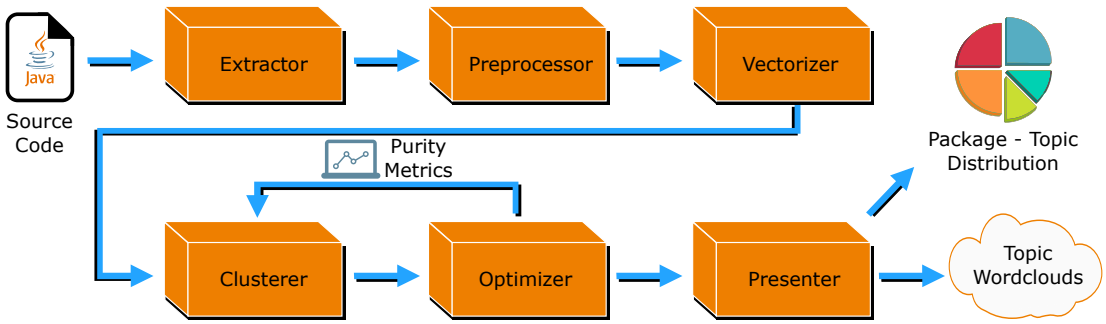

Fig. 1. System Architecture

ageable number of topics, for which no further information is given other than the top terms per topic.

In this paper we propose a system that overcomes the aforementioned drawbacks. We apply topic modeling for extracting semantic information from variable/method names and comments and propose determining the optimal number of topics using a purity metric, with ground truth information from the package structure of each project. Our methodology involves also postprocessing the derived topics, thus constructing a highly intuitive and comprehensive categorization. Finally, the topics are further enriched using online data, thus providing a more abstract summary of the represented functionality.

III. Methodology

The architecture of our system is shown in Figure 1. Our system comprises 6 modules: the *Extractor*, the *Preprocessor*, the *Vectorizer*, the *Clusterer*, the *Optimizer*, and the *Presenter*. As input we use the source code of a software project, which can be either stored locally or downloaded from an online repository (Sourceforge[2] and GitHub[3] connectors were implemented for this cause). The source code is subsequently parsed by the Extractor that extracts tokens from the names and types of methods and variables, the comments, and the javadoc blocks. Subsequently, the Preprocessor constructs a document (bag-of-words representation) for each class file and applies all required transformations (e.g. lowercase transformation).

After that, the documents are converted to vectors in a *Vector Space Model (VSM)* by the Vectorizer. The Clusterer continuously performs semantic clustering on the vectors to extract topics, while the Optimizer computes the purity for each set of topics given the packages of the source code and merges topics into categories. Upon determining the optimal value for purity (that can be overriden by the developer if required), the Presenter uses online information from Stack Exchange[4] in order to add abstract semantic tags to the topics. The output is the set of topics and their categories, which collectively describe the functionality of the project, and the distribution of topics over packages. The functionality of these modules is described in detail in the following paragraphs.

[2]https://sourceforge.net/
[3]https://github.com/
[4]https://stackexchange.com/

A. Extractor

The Extractor receives as input the source code of a software project, or the corresponding Sourceforge/GitHub repository. Our approach is language-agnostic, as applying it to projects of any programming language requires only providing a parser for the language. In this case, we focus on Java projects, thus all non-java files are discarded and java files are parsed using the ASTExtractor tool[5], which produces the *Abstract Syntax Tree (AST)* of each class in XML format. The AST contains all information present in the java file, including fields, method definitions and blocks, statements, etc., as well as comments and javadoc blocks. Given current literature [4]–[8], semantic information is typically contained in almost all of these types.

B. Preprocessor

We retain the names and types of methods and all variables, including not only class fields, but also variables inside method blocks. Comments and javadoc blocks are also retained, while all other information (e.g. statement types) is discarded at this point. The Preprocessor receives all aforementioned data and uses the Python Natural Language Toolkit (NLTK) [17] to construct a bag-of-words representation for each class. The terms of each document/class undergo a series of transformations. CamelCase terms are split (e.g. `readData` is split into `read` and `data`), and all tokens are made lowercase. Javadoc blocks are cleaned using the regular expressions shown in Table I.

TABLE I
Regular Expressions for Javadoc Cleaning

Regular Expression	Description
`([a-zA-Z])/([a-zA-Z])`	Separate words split by /
`<pre>(.*?)</pre>`	Remove text of `<pre>` tags
`<code>(.*?)</code>`	Remove text of `<code>` tags
`\$(.*?)\$`	Remove text between $ chars
`((.*?)\@(.*?))`	Remove emails
`\((.*?)at(.*?)dot(.*?)\)`	Remove emails with `at` and `dot`
`((.*?)\))|(\((.*?))`	Remove parentheses
`<(.*?)>`	Remove text between </>

After that, apostrophe expressions are replaced or removed. In specific, `n't` is replaced by `not`, `'ll` by `will`, `'ve` by

[5]https://github.com/thdiaman/ASTExtractor

have, 're by are, 'd by would, and all other apostrophe expressions (e.g. 's) are removed. Thus, terms such as we'll and isn't are replaced by we will and is not respectively. The exceptions of won't and ain't are also transformed to will not and is not respectively.

Finally, tokenization is performed to remove any punctuation, while two types of stopwords are removed: stopwords of the English language using the list of NLTK [17] and Java stopwords (e.g. for, if, etc.) as defined by the language specification[6]. All remaining tokens are lemmatized using the NLTK lemmatizer (e.g. processes becomes process).

C. Vectorizer

The bag-of-words representation has to be transformed to a structure suitable for use by the semantic clustering algorithm. Thus, the Vectorizer receives as input the terms for each class and constructs a VSM representation, where each term corresponds to a dimension of the model and each list of tokens for a document-class corresponds to a vector of the model. The *frequency* of a term in a document denotes the value of the vector in the corresponding dimension. We used two different vectorization techniques, implemented in scikit-learn [18]: the *count vectorizer* and the *tf-idf vectorizer*. Count vectorization only takes into account the frequency of each term in a document, whereas tf-idf normalizes the frequency and uses also inverse document frequency to balance out the influence of very rare and very common words.

Consider for example two documents d_1 and d_2, where document d_1 has the terms [listener event bean update event] and document d_2 has the terms [tree node tree event tree gui]. The dictionary (dimensions) of the VSM has the terms [bean event gui update listener node tree] and the count vectors of d_1 and d_2 are [1 2 0 1 1 0 0] and [0 1 1 0 0 1 3] respectively.

For the tf-idf vectorizer, the frequency of each term t in a document d (term frequency - tf) is computed as follows:

$$tf(t,d) = 1 + log(f_{t,d}) \tag{1}$$

where $f_{t,d}$ is the absolute frequency of term t in document d. This normalization is performed to avoid any bias towards terms appearing very often in a document (e.g. variable names). After that, this value is also multiplied with the inverse document frequency (idf) of the term t, which is computed by the following equation:

$$idf(t,D) = log(\frac{|D|}{|d_t|}) \tag{2}$$

where $|D|$ is the total number of documents in the collection, and $|d_t|$ is the number of documents containing the term t. The idf ensures that very common words are given low weights, as they do not contain valuable information for a specific document. For the example of this subsection, the tf-idf vectors for documents d_1 and d_2 are [0.30 0 0 0 0.30 0 0] and [0 0 0.30 0 0 0.30 0.44] respectively.

[6]https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

D. Clusterer

Upon having constructed a document-term matrix with the frequencies of each term in each document, the Clusterer module produces a set of topics where the documents are categorized into. Our algorithm of choice is LDA [19], a generative statistical model where documents are viewed as mixture of topics and the task is to produce two probability distributions, one for the occurrence of terms in topics and one for the occurrence of topics in documents. The first probability function indicates the top words that describe each topic, whereas the second denotes how documents are clustered into topics. Both are assumed to follow a sparse Dirichlet prior distribution, which is described by the following equation:

$$Dir(\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{a_i - 1} \tag{3}$$

where K is the number of topics and $\boldsymbol{\alpha} = (a_1 \quad \dots \quad a_K)$ is a vector of the parameters of the distribution. The multivariate Beta function B functions as a normalizing constant, and is expressed using the Gamma function (Γ):

$$B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(a_i)}{\Gamma\left(\sum_{i=1}^{K} a_i\right)} \tag{4}$$

In our implementation, the parameter vectors for the topic-document and document-term distributions were initialized to 500 divided by the number of topics and to 0.001 respectively, as these values have been proven effective for multiple applications [19]. Finally, the algorithm requires as input the number of topics. We perform several runs for different topic counts, starting from 10 topics and gradually increasing until we reach an optimal point determined by the next module (Optimizer), which produces the final result of the algorithm.

E. Optimizer

The Optimizer module aids the developer to determine the optimal number of topics for the Clusterer. This is accomplished by computing a purity metric on the generated topics, using as ground truth the package decomposition of the software project. In specific, we assume that each package of the project corresponds to different functionality and, thus, different semantics. Although the package structure of a software project does not strictly translate into semantic partitioning, it provides a valid proof of concept for tuning the clustering algorithm, since it provides a measure of the effectiveness of topic modeling. Package separation offers not only technical but also semantic information about the project, thus using it for evaluating source code topic modeling tasks is typical [8].

As the result of the Clusterer is a topic-class distribution, computing the purity metric requires aggregating it at package level. Thus, the purity of each topic depends on the classes and packages to which it spans. For the i-th topic, purity is computed by the following equation:

$$Purity(i) = \frac{max_j\{|c_{ij}|\}}{\sum_j |c_{ij}|} \tag{5}$$

where $c_i j$ is the set of classes that belong in package j and are assigned to topic i. Similarly, the total purity for all topics is computed using the following equation:

$$Purity_{Total} = \frac{\sum_i max_j\{|c_{ij}|\}}{\sum_i \sum_j |c_{ij}|} \quad (6)$$

To provide a purity calculation example, we use a distribution of 3 packages and 3 topics for a project shown in Table II. For Topic 1 the package with the maximum number of classes is Package 3, while the total number of classes of all packages for this topic is $10+10+40 = 60$. Thus, the purity for this topic is $40/60 = 0.667$. Similarly, the purity values for the other two topics are 0.857 and 0.909, respectively. The total purity is equal to the sum of the number of classes for the optimal package of each topic, i.e. $40+30+50 = 120$, divided by the total number of classes of all packages (150), which finally amounts to $120/150 = 0.8$.

TABLE II
EXAMPLE TOPICS AND PACKAGES FOR A PROJECT

	Topic 1	Topic 2	Topic 3
Package 1	10	30	0
Package 2	10	0	50
Package 3	40	5	5

As with every purity-like metric, our metric ranges from 0 to 1. The metric is based on the assumption that, in general, every package of a software project should focus on particular functionality, which can be described using one or more topics. Thus, ideally, each topic should for the most part include several classes that belong to the same package. As a result, we may consider purity as a measure of topic separation; in this context, if a topic includes more than one package in high levels of participation, this means that it could be replaced by more (sub)topics describing separate functionality.

As noted in the previous subsection, the number of topics is increased by 1 each time, the clustering algorithm is executed, and the Optimizer calculates the total purity for all topics and shows a graph to the user, containing all values. The developer can terminate this process whenever he/she is satisfied by the level of total purity or wait until the algorithm fully splits the topics into packages. After that, the configuration (number of topics) with the maximum purity value is selected, while the developer is allowed to override this value and thus tune the algorithm according to his/her preference.

Finally, to provide further intuition to the developer and to enhance the manageability of the extracted topics, the Optimizer also constructs higher-level categories for the topics. The categories are actually supersets of topics and are constructed based on the similarity degrees among topics. At first, for each topic we keep the top 15 terms. After that, the topics are given different frequencies/weights according to their position, i.e. the top term is given frequency equal to 15, the second to top is given 14, etc. The topics are then vectorized using tf-idf vectorization (see equations (1) and (2)). Finally, complete linkage hierarchical clustering is applied on the data to map the topics into different clusters/topic categories. The distance function of the clustering was set to the cosine distance, computed as follows for two topic terms vectors $\boldsymbol{t_A}$ and $\boldsymbol{t_B}$:

$$CosineDistance(\boldsymbol{t_A}, \boldsymbol{t_B}) = 1 - \frac{\boldsymbol{t_A}\boldsymbol{t_B}^T}{||\boldsymbol{t_A}||\,||\boldsymbol{t_B}||} \quad (7)$$

where $\boldsymbol{t_A}\boldsymbol{t_B}^T$ is the inner product of the vectors, while $||\boldsymbol{t_A}||$ and $||\boldsymbol{t_B}||$ are their corresponding magnitudes. The cutoff for constructing the clusters was set to 0.4, thus any topics with similarity larger than this value were considered equal.

F. Presenter

Upon having determined the optimal number of topics for the clustering algorithm, the Presenter applies postprocessing to the generated topics and presents the results of the analysis.

First, the topics are enriched using semantic information from Stack Exchange (which has been proven effective for similar challenges [20]). The semantics of each topic are represented by its top 5 terms (as extracted by LDA), along with a term for the name of the software project. These terms form a query that is sent to the Stack Exchange website, using the Google Custom Search API[7]. The query returns relevant question posts, which are subsequently parsed to extract their tags and finally for each topic save them ordered according to the number of questions they appear. Thus, Stack Exchange is used as a link between programming jargon, found in variable names and comments, and abstract field descriptors, such as tags. This way topics are annotated in a more general and abstract way, which allows developers to easier determine the functionality that the topics focus on.

Finally, the output for a software project involves the distribution of packages over topics, the top terms for each topic and the tags extracted from Stack Exchange visualized as word clouds, and the topic categories and purity statistics as computed by the Optimizer.

IV. EVALUATION

In this section we evaluate our work using a dataset of popular software libraries and illustrate its applicability using a case study on the Weka machine learning library.

A. Evaluation Framework

Table III presents information about the dataset used for the evaluation of our system.

TABLE III
DATASET INFORMATION

Name	Packages	Classes	#Vectorizer Tokens Count	Tf-idf
Weka	144	2600	13880	262
SystemML	112	1500	9739	185
DL4J	284	1151	7222	156
Mahout	154	1197	7091	132
Neuroph	155	903	4314	164
Spark	97	682	4561	154

[7]https://developers.google.com/custom-search/

TABLE IV
LDA RESULTS USING COUNT AND TF-IDF VECTORIZER

	Count			Tf-idf		
Name	Topics	Purity	Categories	Topics	Purity	Categories
Weka	130	0.70	90	140	0.60	95
SystemML	110	0.85	60	130	0.75	85
DL4J	150	0.64	100	200	0.52	120
Mahout	150	0.74	100	160	0.55	100
Neuroph	60	0.95	50	90	0.95	60
Spark	40	0.95	40	50	0.90	40

The dataset consists of 6 machine learning libraries, which were downloaded from their GitHub repositories. These libraries were selected since they are quite popular and at the same time quite diverse. In specific, the number of extracted tokens and the size of the libraries vary, which in turn is expected to affect the semantic information contained in each class. Furthermore, libraries with different sizes are expected to influence the performance of our methodology, since the LDA may encounter more noise when analyzing larger projects.

B. Evaluation Results

Table IV presents the results of the semantic clustering for the two different vectorization algorithms. For each library, we present the maximum purity value, which is also used to select the number of topics and subsequently the number of categories.

At first, for both vectorization techniques, we notice that the number of generated topics has some relation to the size of each library. For instance, using the count vectorizer, Weka is described by 130 topics for 2600 classes, whereas the 682 classes of Spark are distributed over only 40 topics. However, this relation is not exactly proportional, as DL4J and Mahout appear to have more topics than Weka for fewer number of classes. This indicates that these two libraries have a sparse distribution of functionalities, with many packages each containing fewer classes. The generated topics however are similar enough to be mapped into categories, e.g. for Mahout at least 1/3 of the topics can be clustered with other topics, for both vectorizer implementations.

The purity scores also seem to follow interesting trends. Smaller libraries, such as Neuroph or Spark, seem to have very high purity scores. This indicates that the semantics of these libraries are properly distributed over packages, which intuitively should be easier for libraries with smaller size. Larger libraries, such as Weka or SystemML, are probably more difficult to organize, given that they have multiple functionalities, possibly exposed through different APIs. Interestingly, DL4J and Mahout seem to have quite low purity values, indicating once again that each of their packages may contain many different topics.

The number of topics and the purity score for each library seem to have similar trends for the two vectorizers. Comparing the two techniques indicates that the count vectorizer achieves higher purity scores than the tf-idf. This is not totally unexpected as the LDA algorithm is effective enough even with noisy datasets (e.g. very frequent or very rare terms). In contrast, the tf-idf vectorizer might reduce the significance of certain terms that contain important semantic information. In any case, these results indicate that both algorithms they can be effectively applied to libraries of varying sizes. The selection process of the optimal number of topics and the quality of the generated topics are further assessed in the following subsection, which includes a case study on the Weka library.

C. Case Study

Weka was selected as the library of our case study, since it is a very popular machine learning library that involves several different types of algorithms, each with its own semantics. Furthermore, the library offers varying functionality [21], including a GUI, a CLI, file readers, converters between file formats, visualizers, etc. Thus, its analysis is expected to reveal interesting information about the semantics of the library, pinpoint well-defined packages and packages that may require refactoring. The analysis is similar for the other five libraries.

To apply our methodology on Weka, we first extract the AST from its classes and create documents containing semantic information as a bag-of-words representation. After that, the documents are vectorized using the count and the tf-idf vectorizers, as defined in subsection III-C. The system proceeds to cluster the documents initially using 10 topics and gradually increasing them by 1 until 300 document topics are reached. The developer is shown a graph (by the Optimizer) with the overall purity of the topics with respect to packages and the merged topic categories (see subsection III-E).

The graphs for the two implementations are shown in Figure 2, where the blue line shows the actual purity value (or number of categories) for the corresponding number of topics, while the red line is a quadratic approximation of the curve. Note that the developer is able, at any time, to stop the clustering procedure and select the optimal number of topics. In this case, we have set the maximum number of topics to 300, and then select a point with maximum purity value[8].

Concerning the trends of the purity curves, it seems that the value of purity is initially low, then gradually increases as the number of topics gets up to a point, and after that point it

[8]A possible option would be to set the maximum number of topics to be relative to the number of packages of the software project under analysis.

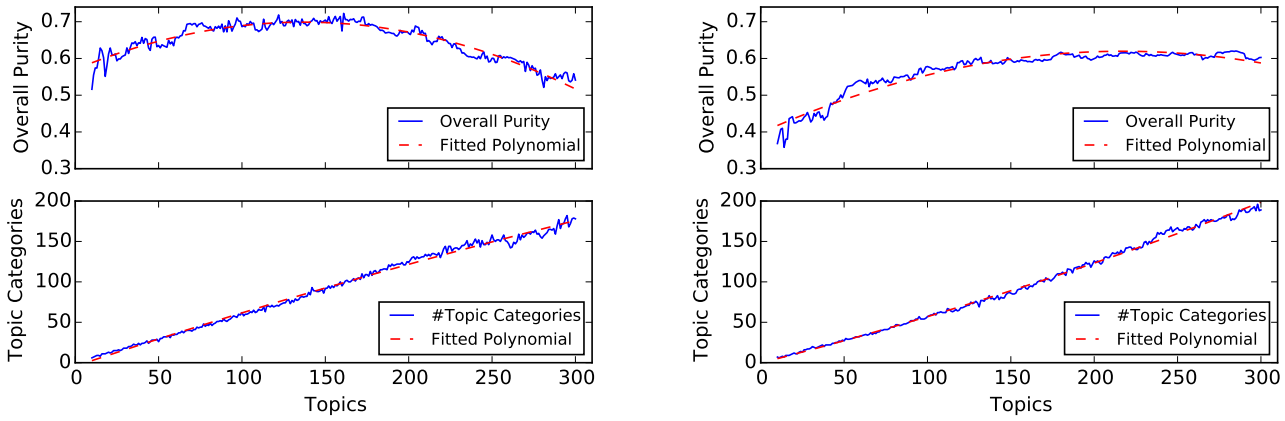

Fig. 2. Purity and Topic Categories for (a) Count Vectorizer and (b) Tf-idf Vectorizer

decreases again. This trend can be interpreted using intuition from the extreme values for the number of topics. Given e.g. 10 topics, all classes would practically be assigned to these 10 topics, thus each of the formed topics would involve multiple packages that would result in very low purity value. On the other hand, having a very large number of topics would force each topic to include very few classes. Theoretically, having an extreme scenario, e.g. with one topic per class, could seem to optimize the purity metric, however note that purity is assessed on package level and not on class level, thus having too many topics would again decrease the purity of the clustering.

As a result, and given the graphs of Figure 2, the optimal number of topics lies somewhere between 130 and 170 topics for both implementations. In this range, both implementations achieve expected maximum purity values, considering Weka is a large library with various functionalities distributed over multiple packages and subpackages. As noted also in subsection IV-B, the count vectorizer seems to achieve higher purity values than the tf-idf vectorizer, due possibly to the loss of information by performing tf-idf. As LDA seems capable of handling the noise of common and rare terms, the two approaches are practically equivalent. Thus, we select the count vectorizer approach for the rest of this case study. We also set the number of topics to 130 as this is the lowest value for which purity is maximum and is expected to produce a result that will be manageable and comprehensive.

Given this configuration, an histogram of the participation (total number of classes) of the 130 topics is shown in Figure 3, where the topics are sorted in descending participation. As shown in this Figure, the top 20 topics contain more than 60% of the semantic information. Hence, for the needs of visualization, we retain these top 20 topics (in terms of absolute participation). Furthermore, for each package we remove any topics with participation lower than 1%. In a different scenario, our system could even be used for identifying these types of classes and present them to the developer so that they are refactored.

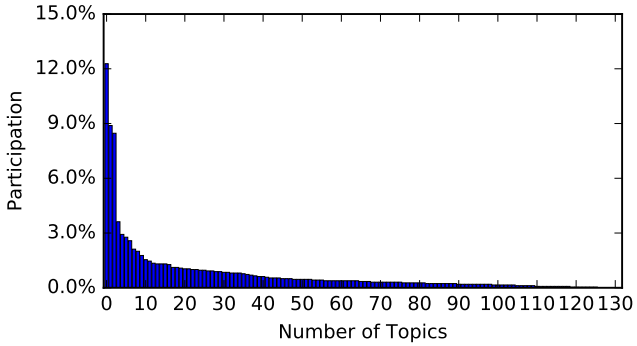

Fig. 3. Distribution of the Weka packages over the Extracted Topics

Finally, the distribution of the packages of Weka over the top 20 extracted topics is visualized in Figure 4. The distribution is quite diverse; certain packages span across multiple topics, while others are more specific. In any case, most topics seem well-defined. For instance, topic 57 clearly refers to testing, since it has several relevant keywords (e.g. test, junit), while at the same time describes a large part of the functionality defined in the test package. This topic also spans across multiple packages, which is expected given that tests are often distributed along with the corresponding components. Other interesting examples include topics 18 and 99 that refer to gui functions (and indeed describe a large part of the functionality of the gui package), topic 49 that refers to file handling, etc.

Concerning the distribution within packages, the results depend on the specifics of each package. Thus, for example, the main topic of the associations package is topic 83, which clearly refers to association rule analysis (as its terms include item, rule, support, etc.). Further examples include the filters package or the python/server packages, which involve topics 23 and 92 respectively. The top 5 terms for package 23 may indeed represent filter operations (e.g. input, batch), while topic 92 clearly refers to external operations for connecting to a python server.

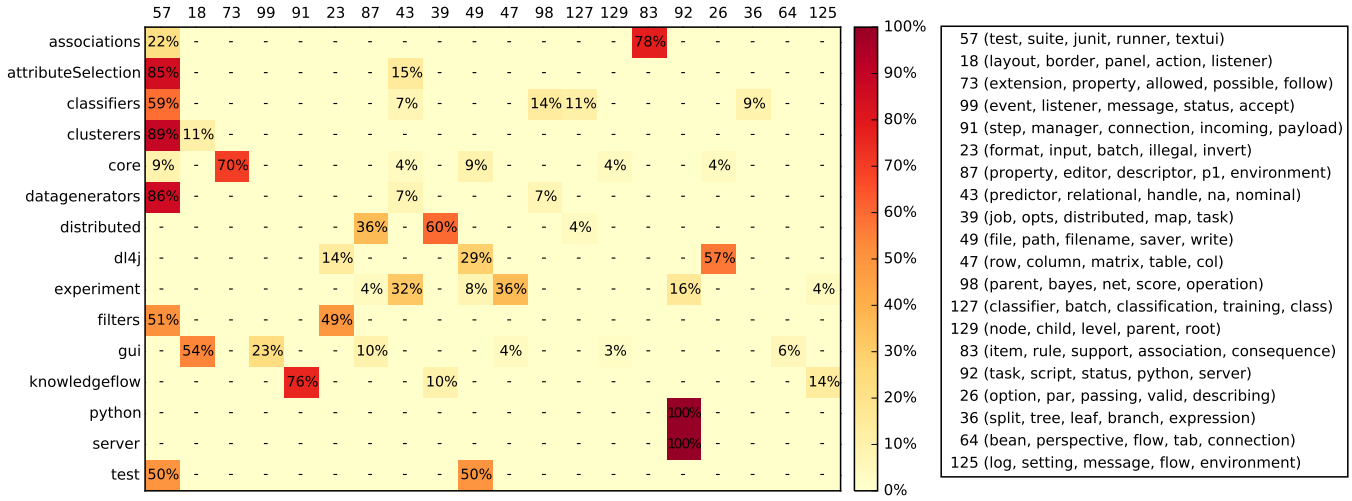

Fig. 4. Distribution of the Weka packages over the Extracted Topics

On the other hand, there are also several packages that span over multiple topics. This quite expected as they contain different functionalities. For instance, package core certainly involves generic functions described better by the generic topics 73 and 26, while it also contains functions for testing (topic 57), for data and file handling (topics 43 and 49 respectively), and for tree/graph operations (topic 129). This is also shown in the classifiers package, since it involves different types of algorithms. Finally, we may also analyze each package into its subpackages in order to further decouple its offered functionality. For example, the subpackage-topic distribution for the subpackages of gui is shown in Figure 5.

For the gui subpackages, the topics with participation higher or equal to 1% are 14. Obviously, the dominant topic over all subpackages is topic 18 that refers to user interface functions. Other than that, we may note certain subpackages that involve also large participation from different topics. For example, data handling subpackages, such as arffviewer or sql, are better described by topic 47 that has terms referring to data manipulation (row, column, matrix, etc.). Other examples include the treevisualizer and graphvisualizer subpackages, for which topic 129 has high participation, as it represents graph/tree handling functionality. Finally, there are also subpackages that offer multiple different functionalities, such as the beans subpackage, which involves events (topic 99), UI environments (topic 87), jobs (topic 39), and java beans (topic 64).

As discussed in the above paragraphs, most extracted topics are well-defined, given that the developer may be able to determine the functionality they cover using the top terms of each topic. However, at times, when the terms are very specific, domain knowledge may be required to understand the functionality of a topic. For instance, given the term "listener", an experienced developer may be able to understand that is should refer to a GUI object. On the other hand, less experienced developers or simply developers that are not familiar with GUI design may find this task harder. As part of our methodology, we confront this challenge by searching in Stack Exchange (see subsection III-F). Upon issuing queries for the top 20 extracted topics of our case study, the top 5 tags for each topic are presented in Table V.

TABLE V
TAGS FOR THE TOPICS OF THE WEKA LIBRARY

N^o	Tags
57	junit, junit4, test-suite, main, jar
18	swing, eclipse, unsupported-class-version, swt, svm
73	jar, machine-learning, visualize, svm, smooks
99	unsupported-class-version, jar, jboss, spring-mvc, spring
91	xml, substring, reflection, invoke, outofboundsexception
23	simpledateformat, reflection, netbeans, libsvm, iso8601
87	servletexception, noclassdeffounderror, netbeans, jstl, jsp
43	linear-regression, r, machine-learning, python, regression
39	emr, hadoop, apache-spark, amazon-emr, hdfs
49	csv, arff, text-mining, text-classification, result
47	r, decision-tree, classification, worksheet-function, rows
98	bayesian-networks, machine-learning, matlab, python, scikit-learn
129	machine-learning, decision-tree, c++, tree, text
83	out-of-memory, heap-memory, garbage-collection, data-mining, associations
92	python, machine-learning, nlp, data-mining, classification
26	utf-8, search, scala, reflection, r-faq
64	translate3d, time, stl, scope, scheduler
125	machine-learning, jar, classification, data-mining, c#

The tags of the topics are indeed relevant to the top terms and they provide a higher level of abstraction. For example, for topic 18 we may see the terms swing and swt, which refer to GUI frameworks, and thus describe the topic better than the specific keywords, such as panel or listener. Other high quality examples include topic 43 that refers to regression, topic 47 that refers to classification, etc. There are also certain low quality mappings, such as topic 125, which refers to logging that is relevant to several topics, therefore the Stack Exchange questions involved very generic tags, such as machine-learning and data-mining. Additionally, topics 36 and 127 did not return any matching questions. In any case, even though these tags cannot directly provide topic names, the developer can use them along with the top terms for each topic in order to

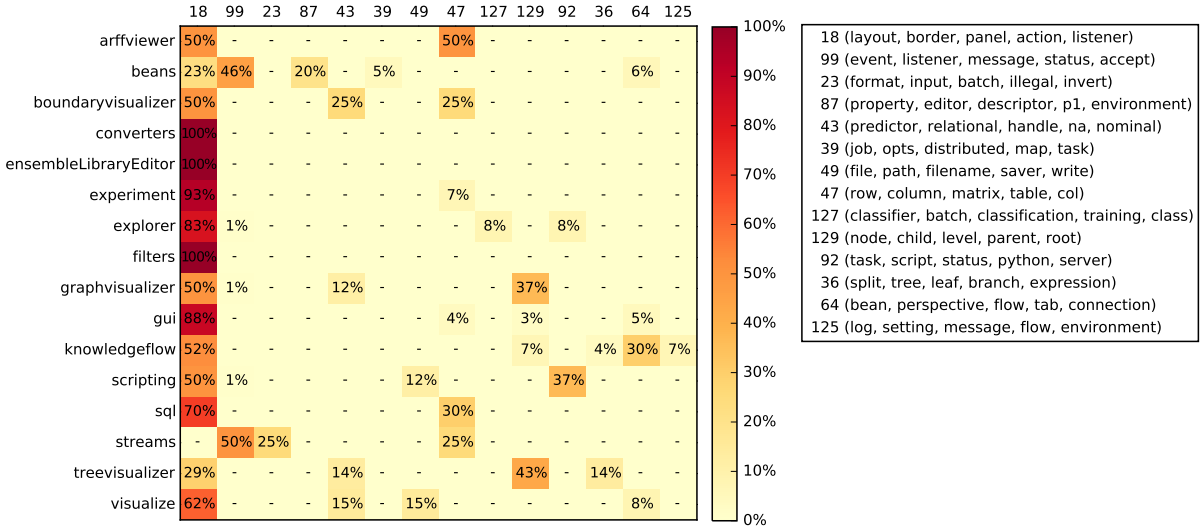

Fig. 5. Distribution of the gui subpackages of Weka over the Extracted Topics

better understand the functionality described by each topic. An indicative visualization that can aid the developer in this task is provided Figure 6 for four topics.

Each word cloud involves the top 5 terms extracted from LDA (in blue) and the top 5 tags from Stack Exchange (in red). The frequency of both types of terms was normalized to $[0, 1]$, and thus the size of each term reflects its frequency. The main functionality of the topics can be derived intuitively. For example, topic 57 clearly refers to testing as it includes terms such as test or junit with high frequencies. Similarly, the term swing and the terms border, layout, and panel indicate that topic 18 refers to GUI operations. This representation is also useful for more complex topics, such as topic 49 or topic 47.

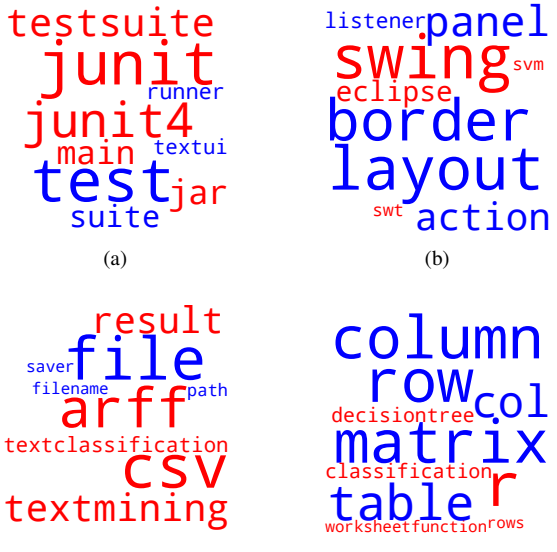

Fig. 6. Word clouds containing topic terms (in blue) and retrieved tags (in red) for (a) topic 57, (b) topic 18, (c) topic 49, and (d) topic 47

For topic 49, the high-frequency terms file, arff, and csv indicate that it refers to file I/O operations, while the rest of the terms (e.g. textmining, textclassification) imply also the use of text mining. Topic 47 most probably involves data manipulation, as terms column, row, matrix, etc. have quite high frequencies, while the rest of the terms (e.g. r, decisiontree, classification) suggest also classification algorithms however with generally lower frequencies/probability.

D. Threats to Validity

The main threats to validity and limitations of our approach involve the choice of evaluation metrics and the lack of comparison with other approaches. Concerning the metrics, the purity of the extracted topics with respect to the package structure of the project may not always be an effective criterion, as the packages may not be well defined. However, as already noted and supported by current literature [8], package separation is suitable for assessing topic modeling techniques as it usually contains useful semantic information about the project. Further manual inspection on the extracted topics with respect to their purity has revealed that the metric is indeed indicative of the quality of the extracted topics.

Concerning the comparison with current approaches, this has proven troublesome both for the qualitative nature of the results and for the unavailability of tools relevant to these approaches. Given the results of the analysis, i.e. the extracted topic distribution, an effective evaluation method would have to rely on some objective ground truth, which is unavailable. This challenge could be overcome by performing a user study or by using human-annotated topics, which are interesting ideas that we consider as future work. Finally, as the unavailability of the source code (or executable) of current approaches have made any comparison difficult, we also choose to upload our source code and findings online (https://github.com/AuthEceSoftEng/CodeSummarizer) to facilitate future researchers that may face similar challenges.

V. Conclusion

Nowadays, as software development requires efficiency to confront the increasing demand for new features, reusing and extending software projects and libraries has become a norm. This gave rise to the need for effective analysis tools to enhance software understanding. However, current tools in topic modeling for software projects require expert knowledge of the underlying technologies as well as the system under analysis. As a result, in this work we implemented a tool that automates the parameter tuning phase, with emphasis on topic number selection. Our methodology involves the calculation of a purity metric that models the distribution of topics with respect to the packages of the software project under analysis.

After selecting an optimal number of topics, either automatically or by overriding the selected value, the developer is presented with the distribution of packages over the extracted topics. Furthermore, the topics are enhanced using semantic tags from Stack Exchange to provide more abstract information about their described functionality. Given the evaluation of our system and the designed case study, it is clear that the extracted topics are indeed representative of the functionalities provided by the software project under analysis. Furthermore, the results indicate that the tags can be helpful in some cases in order to better understand the topics.

Future work includes experimenting with different topic modeling approaches, such as lda2vec, and/or integrating the LDA parameters in the optimization process, based on the purity metric. Additionally, as the purity metric may rely on the package structure dictated by the developer of the project, we plan to further extend our ground truth using also information from the derived topic categories, instead of only presenting them to aid the developer in the selection of the number of topics. The semantic enhancement of topics using tags can also be further improved by constructing queries with more terms (e.g. including also package names) and/or issuing them in different services. Finally, our methodology can be adapted to identify packages that could be merged together and packages that should be divided into subpackages, by analyzing the topic distribution of each package.

Acknowledgment

This work has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-02296).

References

[1] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the Extent and Nature of Software Reuse in Open Source Java Projects," in *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ser. ICSR'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 207–222.

[2] W. Schwittek and S. Eicker, "A Study on Third Party Component Reuse in Java Enterprise Open Source Software," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '13. New York, NY, USA: ACM, 2013, pp. 75–80.

[3] P. Bourque and R. E. Fairley, Eds., *SWEBOK: Guide to the Software Engineering Body of Knowledge*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society, 2014.

[4] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 522–531.

[5] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A Survey on the Use of Topic Models when Mining Software Repositories," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 1843–1919, Oct. 2016.

[6] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "TopicXP: Exploring Topics in Source Code Using Latent Dirichlet Allocation," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–6.

[7] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining Concepts from Code with Probabilistic Topic Models," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 461–464.

[8] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic Clustering: Identifying Topics in Source Code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[9] K. Nie and L. Zhang, "Software Feature Location Based on Topic Models," in *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01*, ser. APSEC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 547–552.

[10] D. Lawrie, C. Uehlinger, and D. Binkley, "Vocabulary Normalization Improves IR-based Concept Location," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 588–591.

[11] G. Maskeri, S. Sarkar, and K. Heafield, "Mining Business Topics in Source Code Using Latent Dirichlet Allocation," in *Proceedings of the 1st India Software Engineering Conference*, ser. ISEC '08. New York, NY, USA: ACM, 2008, pp. 113–120.

[12] A. M. Saeidi, J. Hage, R. Khadka, and S. Jansen, "ITMViz: Interactive Topic Modeling for Source Code Analysis," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 295–298.

[13] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An Automatic Categorization System for Open Source Repositories," *J. Syst. Softw.*, vol. 79, no. 7, pp. 939–953, Jul. 2006.

[14] K. Tian, M. Revelle, and D. Poshyvanyk, "Using Latent Dirichlet Allocation for Automatic Categorization of Software," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 163–166.

[15] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33rd International Conference on Machine Learning*, ser. ICML '16, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2091–2100.

[16] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 200–210.

[17] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2009.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

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

[20] D. Wu, X.-Y. Jing, H. Chen, X. Zhu, H. Zhang, M. Zuo, L. Zi, and C. Zhu, "Automatically answering api-related questions," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 270–271.

[21] E. Frank, M. A. Hall, and I. H. Witten, "The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition," 2016.