Topic Modeling for Feature Location in Software Models: Studying both Code Generation and Interpreted Models

Francisca Pérez*, Raúl Lapeña, Ana C. Marcén, Carlos Cetina

Universidad San Jorge. SVIT Research Group
Autovía A-23 Zaragoza-Huesca Km.299, 50830, Zaragoza, Spain

Abstract

Context: In the last 20 years, the research community has increased its attention to the use of topic modeling for software maintenance and evolution tasks in code. Topic modeling is a popular and promising information retrieval technique that represents topics by word probabilities. Latent Dirichlet Allocation (LDA) is one of the most popular topic modeling methods. However, the use of topic modeling in model-driven software development has been largely neglected. Since software models have less noise (implementation details) than software code, software models might be well-suited for topic modeling.

Objective: This paper presents our LDA-guided evolutionary approach for feature location in software models. Specifically, we consider two types of software models: models for code generation and interpreted model.

Method: We evaluate our approach considering two real-world industrial case studies: code-generation models for train control software, and interpreted models for a commercial video game. To study the impact on the results, we compare our approach for feature location in models against random search and a baseline based on Latent Semantic Indexing, which is a popular information retrieval technique. In addition, we perform a statistical analysis of the results to show that this impact is significant. We also discuss the results in terms of the following aspects: data sparsity, implementation complexity, calibration, and stability.

Results: Our approach significantly outperforms the baseline in terms of recall, precision and F-measure when it comes to interpreted models. This is not the case for code-generation models.

Conclusions: Our analysis of the results uncovers a recommendation towards results improvement. We also show that calibration approaches can be transferred from code to models. The findings of our work with regards to the compensation of instability have the potential to help not only feature location in models, but also in code.

Keywords: Topic Modeling, Software Models, Search-based Software Engineering, Feature Location

1. Introduction

Latent Dirichlet Allocation (LDA) [1] is a popular and promising information retrieval technique, and one of the most popular topic modeling methods [2]. A recent topic modeling survey [2] shows that the research community has increased its attention to the use of LDA through several publications in various

*Corresponding author. Tel.: +34 976060100
Email addresses: mfperez@usj.es (Francisca Pérez), rlapena@usj.es (Raúl Lapeña), acmarcen@usj.es (Ana C. Marcén), ccetina@usj.es (Carlos Cetina)

fields such as software maintenance and evolution tasks. LDA represents topics by word probabilities. The words with the highest probabilities in each topic usually give a good idea of what the topic is about. Previous works have proposed LDA to support software maintenance and evolution tasks in source code such as feature location [3], bug localization [4], and traceability links recovery [5, 6].

Source code often suffers from the limitations of the lexicon from which source code text is drawn. The source code text that is embedded in identifiers, comments, and string literals tends to be sparse in nature. Source code text also lacks uniqueness and exhibits a large degree of repetitiveness compared to natural language text [7, 8].

In contrast to source code, software models have less noise (implementation details) [9]. Software models are high-level specifications of systems. Software models contain information that raises the abstraction level by using language that is closer to the problem domain, and that is less bound to the underlying implementation and technology [9]. In addition, the information contained in software models has a different granularity than that contained in other software artifacts: for instance, in code, FL approaches look for the most relevant method for the implementation of the feature; while in the case of software models, FL approaches look for the most relevant model fragment for the implementation of the feature. If a given feature description shares topic(s) with a model fragment of the population, the model fragment could be a strong candidate towards implementing the feature. In this context, the coarse-grained focus (topics) of LDA could enhance software maintenance tasks where software models are used.

However, software models have not been researched by any of the 74 works included in a recent topic modeling survey [2], even though they are a popular asset in software development [9, 10]. We acknowledge that models have not replaced source code as a means of software development, but they have nonetheless been reported as a successful paradigm to develop industrial software [9, 10].

In this paper, we assess the use of LDA in the context of software models. More precisely, we propose to use LDA as a fitness function for an evolutionary algorithm with the aim of enhancing Feature Location (FL) in software models. The goal of FL is to find the model elements (i.e., model fragment) in a system that implement a feature, where the term 'feature' refers to a specific functionality or characteristic of a product. FL is arguably one of the most frequently undertaken software maintenance tasks [11, 3, 12, 13].

We evaluate the performance of LDA considering two model-based industrial case studies. Each case study uses a different kind of software models: software models for code generation, and software models for interpretation. The first case study belongs to a worldwide leader in train manufacturing, *Construcciones y Auxiliar de Ferrocarriles* (CAF) [1], a company that formalizes the manufactured products in software models using a Domain-Specific Language (DSL). In CAF, software models are used to generate the firmware that controls their trains. The second case study belongs to a commercial video game, Kromaia, that uses software models to reason about the system, perform validations, and define game content such as bosses, worlds, and goals. In Kromaia, software models are used for interpretation. Thus, the content defined in the models is read and interpreted when the game is launched (without altering the source code of the video game). The video game has been released worldwide in two different platforms (PlayStation 4 and STEAM) and in 8 different languages.

To put the performance of LDA in perspective and to study the impact on the results, we compare our approach against a Latent Semantic Indexing (LSI) [14] baseline and Random Search. LSI is a popular information retrieval technique that has been proposed to support software engineering tasks such as FL in source code [3] and software models [15, 16]. Random Search is used as a sanity check that enables us to determine if LDA performs better than mere chance. We compare the results of each case study against those obtained by the LSI baseline and Random Search in terms of recall, precision, and F-measure. In addition, we perform a statistical anal-

[1] `www.caf.net/en`

ysis (Quade test) and effect size measures (Vargha and Delaney's $\hat{A}_{12}$ [17] and Cliff's delta [18, 19]) in order to provide quantitative evidence of the impact of the results (following the guidelines by Arcuri and Briand [20]) and to show that this impact is significant. Finally, we carried out a focus group interview with the aim of acquiring qualitative data and feedback from the engineers about the obtained results. Hence, the contributions of this paper are threefold:

- We investigate the use of LDA for FL in two kinds of software models (code generation and model interpretation). The software models belong to two industrial case studies. To the best of our knowledge, this is the first effort that addresses the challenge of FL in models for interpretation in the literature. This is relevant because FL is an essential task for software maintenance and evolution [11, 3].
- We experimentally demonstrate that LDA significantly outperforms the LSI baseline (by 16.33% in F-measure) when software models for interpretation are used. In contrast, the LSI baseline outperforms LDA (by 7.33% in F-measure) when software models for code generation are used. Both LDA and the LSI baseline outperform Random Search in all measurements in both case studies. In addition, through the focus group interview, we evidence that practitioners prefer the results of the LDA-based approach over the results of the LSI baseline.
- We provide a comprehensive discussion on the results considering four main limitations (data sparsity, implementation complexity, calibration, and stability). We learned that (1) LDA calibration approaches for code can also be transferred to models, (2) the usage of an evolutionary algorithm seems promising to compensate for LDA instability, and (3) code generation models can achieve better results if the model-to-code transformation process is checked. We provide insight into why the baseline outperforms LDA when models for code generation are used. We found that the model-to-code transformation process increases the information included in the models. This phenomenon affects the topics more than the similarity of terms because some topics change but there are still some terms in common. If we omit the cases in which this phenomenon occurs, the LDA fitness guide the approach to outperform the baseline when software models for generation are used. Hence, we recommend paying attention to the model-to-code transformation process in other code generation contexts.

The remainder of this paper is organized as follows. Section 2 provides background notions for feature location in software models and LDA. Section 3 describes our LDA-guided evolutionary approach for locating features in software models. Section 4 presents our evaluation in two real-world industrial case studies. Results are reported in Section 5 and discussed in Section 6. Section 7 discusses the threats to validity that could have affected our evaluation. Section 8 presents the related work. Finally, Section 9 concludes the paper and outlines directions for future work.

2. Background

This section overviews and motivates feature location in software models, and describes LDA in a nutshell.

2.1. Feature Location in software models

This section introduces FL in software models, using as a basis a model from one of our industrial case studies, in the domain of railway solutions. Train units are furnished with multiple pieces of equipment that carry out specific tasks for the train. Some examples of these devices are the pantograph that harvests power from overhead wires, or the circuit breaker that isolates and connects electrical circuits. The control software of the train unit makes all the equipment cooperate, achieving the desired functionality and guaranteeing compliance with regulations.

To implement such software, our industrial partner formalizes the products manufactured in software models using a DSL. The software models describe both the equipment interactions and the non-functional regulation aspects. The top-most part of

Figure 1 depicts an example model, taken from a real-world case study. It presents a scenario where two separate pantographs (high voltage equipment) collect energy from the overhead wires, sending it to their independent voltage converters through their respective circuit breakers (contactors). The converters power their assigned equipment: HVAC (air conditioning system), PA (public address system), and CCTV (television system). The model also presents an intermediate 'failure overload' contactor that connects the 'peer coverage' converter to an equipment assigned to the other converter.

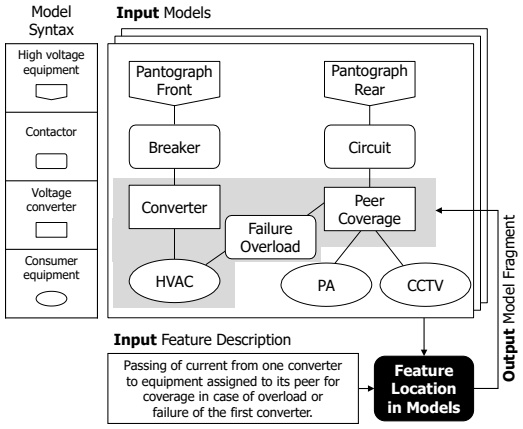

Figure 1: Feature Location in Software Models

FL in software models takes as input the models and the feature description. The output is the most relevant model fragment for the feature description. In the example of Figure 1, the feature is the 'converter assistance' feature, which allows the passing of current from one converter to equipment assigned to its peer for coverage in case of overload or failure, and the fragment is the set of model elements highlighted in light gray.

Although the example shown in Figure 1 may make FL in software models appear easy, in industrial scenarios, the complexity of the models in use and the number of elements in place render FL a time-consuming, error prone, and person-power intensive task. For instance, the data set provided by our industrial partner comprises 23 trains, with one software model per train, and 121 feature descriptions. All models comprise above 1200 model elements. The elements of a model implement its different included features (e.g., the model of Figure 1 includes the $CCTV$ consumer equipment element because this model includes the feature of the television system). In order to match a particular feature to a model fragment, a domain expert would need to decide which set of elements of the models are relevant to the feature. Assuming that the domain expert must spend around 5 seconds to take the decision with each element [21], creating a fragment would take around 100 minutes. Manually locating all the features would take slightly more than 200 hours (approximately, 26 full-time working days).

2.2. LDA in a nutshell

Latent Dirichlet Allocation (LDA) [1] is an unsupervised probabilistic technique for estimating a topic distribution over a text corpus. The corpus is made up of a set of documents, where each document is a set of terms. As a result, a probability distribution is obtained for each document, indicating the likelihood that it expresses each topic. In addition, a probability distribution is obtained for each topic identified by LDA, indicating the likelihood of a term from the corpus being assigned to the topic.

LDA inputs include the documents (D), the number of topics (K), and a set of hyper-parameters (i.e., a set of parameters that have a smoothing effect on the topic model generated as output). The hyper-parameters of any LDA implementation are:

- k, which is the number of topics that should be extracted from the data.
- α, which influences the topic distributions per document. Lower α value results in fewer topics per document.
- β, which affects the distribution of terms per topic. Lower β value results in fewer terms per topic, which in turn implies an increase in the number of topics needed to describe a particular document.

In this work, we use the Collapsed Gibbs Sampling (CGS) for LDA because it requires less computa-

tional time [22], and it was previously used for locating features in source code [23]. This implementation requires an additional hyper-parameter, σ, which denotes the number of sweeps to be made over the corpus. Finding the configuration of hyper-parameters that provides the best performance is not a trivial task [24, 23, 25]. Thus, the hyper-parameter values typically are set to either according to the facto standard values [23, 25] or calibrated [24].

LDA outputs include: ϕ, which is the matrix that contains the term-topic probability distribution, and θ, which is the matrix that contains the topic-document probability distribution.

3. Our approach

Our approach uses LDA to guide an evolutionary algorithm in the localization of features. The evolutionary algorithm explores the vast amount of model fragments (magnitudes around 10150 fragments for models of 500 elements), and LDA assesses the relevance of each model fragment to the feature description. Figure 2 presents an overview of our approach. The top part of the figure highlights the inputs of the approach (the input software models and feature description), the middle part shows the four steps of the evolutionary algorithm (text processing, model fragment initialization, fitness assessment, and genetic manipulation), and the bottom part presents the output of the approach (a ranking of model fragments that are relevant for the feature description). The four steps are presented in detail through the remainder of the section.

3.1. Text processing

The texts of the inputs are processed through the usage of Natural Language Processing (NLP) techniques, which homogenize the NL texts. The usage of NLP techniques is often regarded as beneficial and is a frequent practice for Software Engineering tasks in the field of Information Retrieval [26]. In order to process the NL texts, we use the following NLP techniques [27]: tokenization, syntactical analysis, stemming, and human NLP (inclusion of domain terms and removal of stopwords).

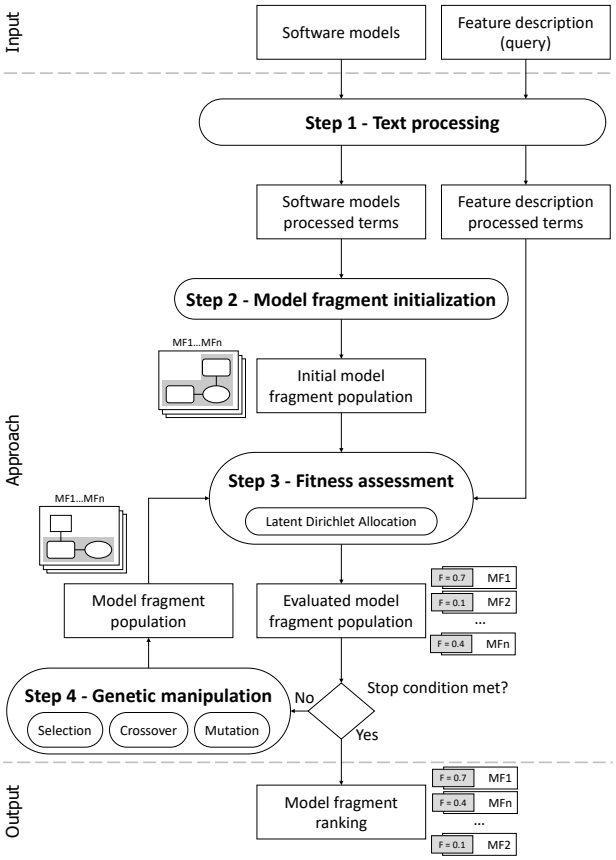

Figure 2: Approach Overview

As an example, consider the feature description and the model fragment of Figure 1. Through the application of the techniques described above, the language is homogenized: the feature description (*'passing of current from one converter to equipment assigned to its peer for coverage in case of overload or failure of the first converter'*) is transformed into the terms [*current, convert, hvac, peer, convert, coverag, overload, failur, convert*], and the text of the model fragment is transformed into the terms [*convert, hvac, failur, overload, peer, coverag*].

3.2. Model fragment initialization

This step generates an initial population of model fragments from the input set of models. To generate

the initial population of model fragments, model elements of a model are randomly selected. This random technique is the one commonly used in evolutionary algorithms [16].

To represent a model fragment and to be able to easily manipulate it, the model fragment is encoded in a bit string. The encoding is an array that contains as many positions as elements in the model. Each position in the bit string has two possible values: 0, if the element does not appear in the fragment; or 1, if the element does appear in the fragment. Figure 4 shows model fragments (highlighted in gray) and their corresponding bit strings. For example, elements 1 and 6-8 are set to '1' in the bit string of Parent 1 in Figure 4, so these model elements comprise the model fragment. The initial population of model fragments calculated through this step is used as input in the next step.

3.3. Fitness assessment (LDA)

To assess the relevance of each model fragment with regard to the provided feature description (i.e., query), we use LDA as a fitness function. Given the terms for the query Q and the outputs of LDA (ϕ and θ), the conditional probability P of Q given a document D_i is computed as follows [23]:

$$Sim(Q, D_i) = P(Q|D_i) = \prod_{q_k \in Q} P(q_k|D_i)$$

where q_k is the k^{th} homogenized term in the query, and D_i is a document (i.e., a model fragment) of the population that is made of a set of homogenized terms.

Figure 3 shows an example of the fitness assessment for a population of model fragments, being each model fragment an individual of the population. Given a feature description (query), the figure depicts how the fitness is assigned to the n model fragments of the population by using LDA.

The left part of the figure shows an example of the outputs of LDA, which include ϕ and θ as described in Section 2.2. ϕ contains the term (K) to topic (T) probability distribution. Each cell can have a value from 0 to 1, indicating the likelihood of a term from the corpus being assigned to a particular topic. For instance, a value of $\phi[T_1, current] = 0.1$ indicates a 10% likelihood of the term *current* being assigned to topic T_1. θ contains the topic (T) to Model Fragment (MF) probability distribution. Each cell, with values that again range from 0 to 1, indicates the likelihood with which a model fragment from the population expresses a topic. For instance, a value of $\theta[MF_1, T_1] = 0.23$ indicates that model fragment MF_1 has a 23% likelihood of expressing topic T_1.

The right part of the figure shows an example of a provided feature description and its homogenized terms. The homogenized terms of the feature description, along with the LDA outputs, are used to calculate a matrix with dimensions K x n, of which an example is shown in the middle part of the figure. The rows of the matrix represent the homogenized terms of the feature description (q_t) and the columns of the matrix represent each of the model fragments in the population. Each cell in the matrix contains the conditional probability for each homogenized term given a particular model fragment (corresponding to $P(q_k|D_i)$ in the equation). The value of the conditional probability is obtained by applying the dot product operation between all the ϕ values associated with the homogenized term and all the θ values associated with the model fragment. The figure depicts an example of this operation, concerning the homogenized term *current* and model fragment MF_1. The operation is visually represented through a solid blue highlight of the ϕ and θ cells involved in the dot product, along with blue lines that point to the result of the operation, stored in the cell of the central matrix that is also highlighted in solid blue. The obtained value is the conditional probability for the homogenized term *current* given model fragment MF_1, that is, $P(current|MF_1) = 0.4$.

Once the matrix is calculated, the conditional probability values of the terms obtained for each model fragment are used to compute the conditional probability between the query and the different model fragments as described in the equation. The newly calculated values are the fitness values associated with each of the model fragments. In the figure, these values appear in the vector beneath the matrix. The visual representation of the operation

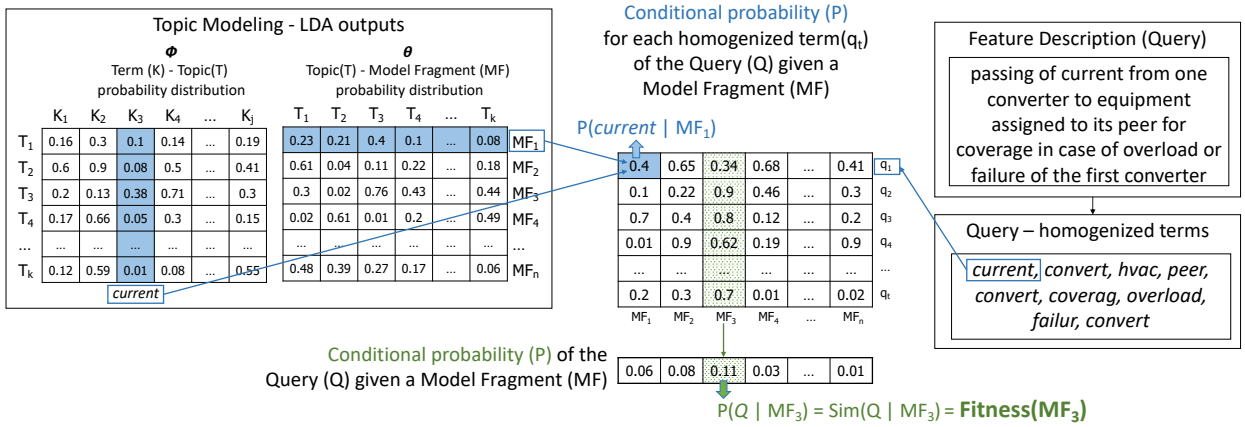

Figure 3: Fitness assessment example

is supported by an example, in the form of a light green highlight of the values obtained for model fragment MF_3, which lead to a conditional probability $P(Q|MF_3) = 0.11$ between the query and model fragment MF_3. The final outcome of the LDA fitness, $P(Q|MF_3)$, is stored as the fitness of MF_3.

The evolutionary algorithm will continue its execution by providing the evaluated model fragment population as input to Step 4 (genetic manipulation) until the stop condition is met. The stop condition can be a fixed number of generations, a trigger value for the fitness, or a time slot. The stop condition greatly depends on the domain and the problem being solved, and is therefore calibrated based on the output results. When the algorithm stops, it returns a model fragment ranking, sorted according to the fitness values obtained by LDA. Higher fitness scores entail higher degrees of similarity between query and model fragment, whereas lower fitness scores entail lower degrees of similarity between query and model fragment. As an example, if the stop condition of the evolutionary algorithm is met at the end of the iteration that yields the fitness values shown in the figure, the algorithm will order the model fragments into a ranking where MF_3 will be in the first position due to having the highest fitness score among all the model fragments in the population, with the rest of the model fragments ordered beneath it. The entire ranking of model fragments is associated as a solution for the query, with the model fragment in the first position of the ranking being considered as the most similar model fragment to the query.

Similarly to other works that retrieve text from an initial query using LDA or other information retrieval techniques, the results depend on the quality of the queries [23, 28], which is typically improved through an iterative refinement process [29]. If a query (i.e., feature description) explicitly mentions more properties and values of the model elements to be retrieved, the result will be closer to the objective. Therefore, even when irrelevant model fragments are obtained in the ranking, the results can be considered as a starting point for the iterative refinement process. From there, software engineers can either manually tweak the proposed solutions or modify the feature description to automatically obtain different solution model fragments. New inputs (feature description and software models where the feature must be located) will be necessary to execute the evolutionary algorithm again.

3.4. Genetic manipulation

As the fourth and final step, if the stop condition is not met, the evolutionary algorithm uses three genetic operations (selection, crossover, and mutation) to generate new model fragments based on existing ones.

The **selection operation** picks the best candidates from the population as input for the rest of operations. There are different methods that can be used to perform the selection of the parents, but one of the most spread choices is to follow the wheel selection mechanism [30], where each model fragment from the population has a probability of being selected proportional to their fitness score. The single-point **crossover operation** enables the creation of new individuals by combining the genetic material from two parent model fragments. The **mutation operation** is used to imitate the mutations that randomly occur in nature when new individuals are born. The operations are taken from [31] and [32] respectively, where their application to models is detailed.

Figure 4 shows an example of application of the genetic operations, following the example models introduced in Section 2.1. First, the selection operation is applied, and two model fragments from the population are chosen.

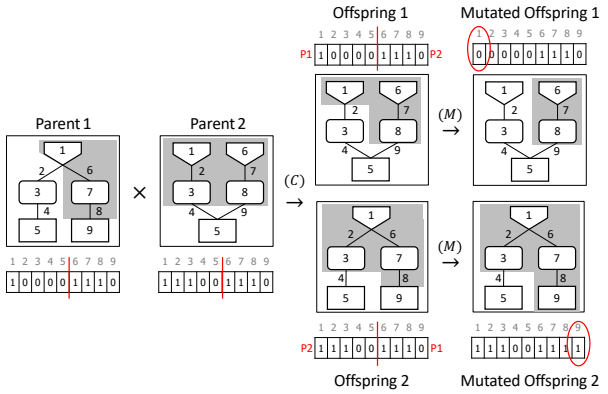

Figure 4: Genetic operations example

Afterwards, the crossover operation is applied. The two selected model fragments are taken as parents. The operation randomly selects a point on both parents and swaps the encoding based on the selected point, generating two offspring individuals in the process. The first offspring will hold the first half of the encoding from the first parent, and the second half of the encoding from the second parent. The second offspring will hold the first half of the encoding from the second parent, and the second half of the encoding from the first parent. Figure 4 shows an example of the single-point crossover operation over two model fragments.

Finally, the mutation operation is applied to add or remove elements from the offspring model fragments. In the example of Figure 4, the mutation operation takes the offspring fragments produced through the crossover operation and removes or adds one element from them. The model fragments resulting from the mutated offspring fragments are new candidates in the population.

The model fragment population generated through the genetic operations is then assessed through the fitness function. The last two steps of the approach (fitness assessment and genetic manipulation) are repeated in this manner until the solution converges to a certain stop condition.

4. Evaluation

This section describes the evaluation of our work: the research questions we aim to answer, the two model-based industrial case studies that comprise both the models for code generation and the models for interpretation, the sanity check and the baseline to put the performance of our work in perspective, and the experimental setup we use to answer the research questions.

4.1. Research questions

We aim to answer the following research questions:

RQ$_1$ What is the performance of topic modeling for locating features in the two kinds of software models?

RQ$_2$ How much is topic modeling influencing the performance in the two kinds of software models?

Answering RQ$_1$ allows us to discover the performance results (in terms of recall, precision and the F-measure) of topic modeling, the sanity check and the baseline in two model-based industrial case studies (which use different kind of models: the models for code generation and the models for interpretation). Answering RQ$_2$ allows us to properly compare

the performance results using statistical methods in order to determine whether the differences in the results are significant and if so, to determine by how much (the magnitude of improvement).

4.2. Case studies

Our first case study has been provided by one of our industrial partners in the railway domain, CAF. In our first case study, the models are used for code generation purposes. The data is made up of 23 trains where, on average, each product model[2] is composed of more than 1200 model elements. Each model element has about 15 properties that include terms, which are used to differentiate among model elements. Specifically, our industrial partner provided the following documentation: 121 feature descriptions, the 23 product models where the model fragments should be located, and the approved feature realization (that is, the model fragment that corresponds to each feature) that will be considered to be the ground truth (oracle). The model fragments that correspond to each feature have between 5 and 20 model elements, with an average of 13.55 model elements and a median of 14 model elements.

Our second case study has been provided by one of our industrial partners in the video games development domain. This industrial partner has provided us with the models and documentation for one of their commercial video games, Kromaia, which was released worldwide in both physical and digital versions for PlayStation 4 and STEAM, and translated into 8 different languages. In this second case study, the models are built with interpretative purposes in mind, or in other words, to formalize the system and capture its particularities. These models are used to reason about the system, perform validations, or transform them into run-time objects. Our industrial partner provided us with 15 product models, 106 feature descriptions, and the approved feature realizations. Each product model is composed of more than 800 model elements. The model fragments that correspond to each feature have between 7 and 18 model elements, with an average of 12.42 model elements and a median of 12 model elements. Each model element has about 16 properties that include terms.

The upper part of Figure 5 shows an excerpt of a product model in Kromaia where different model elements are included to specify content in the video game such as a boss. A boss is a powerful enemy that the player must defeat at the end of a level. The model elements that are included in the figure show an excerpt of *hull* and *link*. A *hull* is a module of solid bodies that are connected through configurable *links*. The lower part of Figure 5 shows an example of boss during the execution of the video game as a result of interpreting the configuration of the product model[3].

4.3. Baseline

For each case study, we set as baseline a variant of the evolutionary algorithm that is described in Section 3 in which the fitness function (Step 3) is replaced with Latent Semantic Indexing (LSI) [14], which analyzes relationships between *queries* and *documents* (bodies of text). LSI is a popular information retrieval technique [23] that has been used in software engineering tasks such as feature location in source code [3]. Furthermore, previous works evaluated different fitness functions to calculate the similarity between the feature description and each model fragment: LSI [15], Formal Concept Analysis [32], learning to rank [33], understandability [34], timing [35], and combinations of these [15]. Considering all of the works mentioned above, LSI is the fitness function that achieves the best results. LSI is also the most common fitness used for feature location in models [32, 16, 36, 15, 21]. Consequently, we have used LSI as a baseline for this work.

LSI produces a *term-by-document co-occurrence matrix*. Rows in the matrix stand for the *terms* to be found, in our case, the homogenized terms that appear in the feature description and the model fragments. Columns in the matrix stand for each of the

[2]Learn more about the models for code generation of CAF at `https://youtu.be/Ypcl2evEQB8`

[3]Learn more about the interpreted models of Kromaia at `https://youtu.be/Vp3Zt4qXkoY`

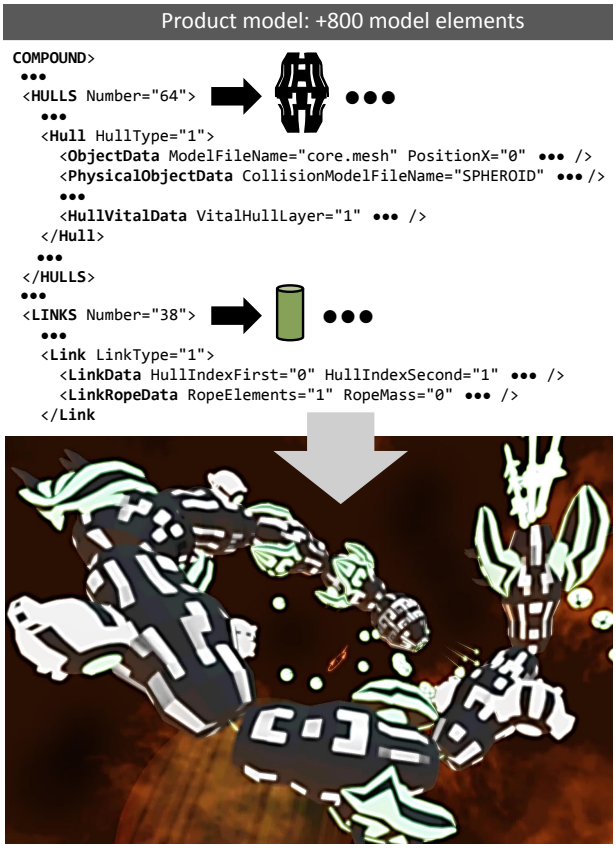

Figure 5: Example of model and interpreted game content

search *documents*, in our case, the model fragments in the generated population. The final column stands for the *query*, in our case, the input feature description. Each cell in the matrix contains the frequency of each *term* in each *document*.

Vector representations of the *documents* and the *query* are obtained by normalizing and decomposing the *term-by-document co-occurrence matrix* using a matrix factorization technique called *Singular Value Decomposition* (SVD) [14]. Afterwards, the similarity degree between the *query* and the *documents* is measured by calculating the cosine between the *query* vector and the *document* vectors. Cosine values closer to one denote a high degree of similarity, and cosine values closer to minus one denote a low degree of similarity. Through this measurement, the model fragments are ordered according to their similarity degree to the query.

Besides the LSI baseline, we compare our work against a standard Random Search approach that is used as a sanity check. Using Random Search enables us to determine if topic modeling performs better than mere chance. Random Search starts with a random initial model fragment (considered as the best candidate for the first iteration). Then, a new random model fragment is generated (candidate). The best candidate is updated if the fitness value of the new candidate is better than that of the current best candidate. The fitness value is calculated using LSI as described for the baseline. This loop is repeated until the stop condition is met.

4.4. Experimental setup

Figure 6 shows an overview of the experimental setup to answer each research question, which is described as follows:

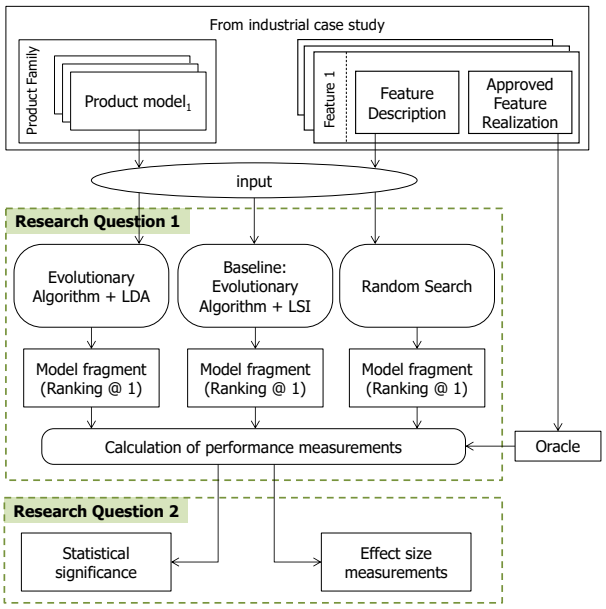

Figure 6: Experimental setup

Answering RQ$_1$: The performance of the approach is assessed in each case study separately. To

that extent, the evaluation starts by taking the inputs from the case study, shown in the upper part of Figure 6 (the product models and feature descriptions with their corresponding approved feature realization, which conforms the evaluation oracle). Next, our LDA-based approach, the LSI-based baseline and random search are executed, as the middle part of Figure 6 shows. We executed 30 independent runs (as suggested by Arcuri and Fraser [37]) for each feature in both the approach and the baseline: 121 (features) x 3 (approach, baseline and random search) x 30 repetitions in Case Study 1, plus 106 (features) x 3 (approach, baseline and random search) x 30 repetitions in Case Study 2, for a total of 20430 independent runs.

From the ranking of model fragments that is obtained as a result of each run, the first model fragment (i.e., the model fragment with the highest fitness value) is compared against the oracle, which is considered to be the ground truth. Once the comparison is performed, a confusion matrix is calculated.

A confusion matrix is a table that is often used to describe the performance of a classification model on a set of data (the best solution) for which the true values are known (from the oracle). In our case, each outputted solution is a model fragment. Since the granularity is at the level of model elements, the presence or absence of each model element is considered as a classification. The confusion matrix arranges the results of the comparison between the model fragment from the oracle and the solution into four categories of values: (1) True Positive (TP) values, model elements that are present in the model fragments of both the solution and the oracle, (2) False Positive (FP) values, model elements that are present in the solution but absent in the oracle, (3) True Negative (TN) values, model elements that are absent in both the solution and the oracle, and (4) False Negative (FN) values, model elements that are absent in the solution but present in the oracle.

The confusion matrix holds the results of the comparison between the results of the execution and the oracle. From the values in the matrix, it is possible to extract measurements that evaluate the performance of the approach and the baseline for each case study. Specifically, we derive three performance measurements which are widely accepted in the software engineering research community [38, 5, 39]: recall, precision, and F-measure.

Recall measures the number of elements of the oracle that are correctly retrieved by the proposed solution, and is defined as:

$$Recall = \frac{TP}{TP + FN}$$

Precision measures the number of elements from the solution that are correct according to the oracle, and is defined as:

$$Precision = \frac{TP}{TP + FP}$$

Finally, the F-measure corresponds to the harmonic mean of precision and recall:

$$F - measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Recall values can range between 0% (which means that no single model element from the oracle is present in the solution model fragment) to 100% (which means that all the model elements from the oracle are present in the solution model fragment). Precision values can range between 0% (which means that no model elements from the solution model fragment appear in the oracle) to 100% (which means that all the model elements from the solution model fragment appear in the oracle). A solution with values of 100% in both precision and recall implies that the solution and the oracle are the same.

Answering RQ_2: To determine whether the differences between our approach and the baseline, and between our approach and random search are significant in the two case studies, the results must be properly compared and analyzed using statistical methods. To do this, we follow the guidelines presented in [20] with the aim to provide formal evidence that the differences do in fact have an impact on the comparison measurements (or in other words, that the differences in the results were not obtained by mere chance).

A statistical test should then be run to assess whether there is enough empirical evidence to claim

that there are differences between our approach and the baseline, between our approach and random search, and between the baseline and random search. It is accepted by the research community that a *p-value* under 0.05 implies statistical significance [20].

The statistical test that must be followed depends on the properties of the data. Since our data does not follow a normal distribution, our analysis requires the usage of non-parametric techniques. There are several tests for analyzing this kind of data. However, the Quade test is more powerful than other tests when working with real data [40], and according to [41], has shown better results than other tests when the number of algorithms is low (no more than 4 or 5 algorithms). The Quade test was also used by previous FL approaches [16, 15]. We record a *p-value* for each performance measure (recall, precision and F-measure) in each case study.

To determine how much the performance is influenced by using our approach, the baseline and random search, it is important to assess (through effect size measures) whether our approach is statistically better than the baseline, and if so, measuring how much the solution obtained by our approach improves the quality of the solution obtained by the baseline (the magnitude of the improvement). For non-parametric effect size measurements, we use Vargha and Delaney's $\hat{A}_{12}$[17] and Cliff's delta [18, 19].

$\hat{A}_{12}$ measures the probability that running one approach yields higher values than running another approach. With the $\hat{A}_{12}$ statistic, the approaches are compared in pairs (*treatment* vs *control*). If the $\hat{A}_{12}$ statistic obtains a value greater than 0.5, the comparison will be in favor of the treatment. If the $\hat{A}_{12}$ statistic obtains a value lesser than 0.5, the comparison will be in favor of the control and 1-$\hat{A}_{12}$ will be used to interpret the magnitude of effect. According to the guidelines for interpreting $\hat{A}_{12}$ values [17], the $\hat{A}_{12}$ value of 0.5 means that the two approaches are equivalent (no effect). The $\hat{A}_{12}$ value of 0.56 means a small effect in the magnitude of improvement, 0.64 means a medium effect, and 0.71 means a big effect. For example, a value of $\hat{A}_{12} = 0.56$ means that on 56% of the runs, the treatment would obtain better results than the control and that the effect in the magnitude of improvement is small. A value of $\hat{A}_{12} = 0.24$ means that on 76% of the runs, the control would obtain better results than the treatment, and that the effect in the magnitude of improvement is large.

Cliff's delta is an ordinal statistic that describes the frequency with which an observation from one group is higher than an observation from another group compared to the reverse situation. It can be interpreted as the degree to which two distributions overlap, with values ranging from -1 to 1. For instance, when comparing distributions of the treatment and the control, a value of 0 means no difference between the two distributions, a value of -1 means that all samples in distribution of the treatment are lower than all samples in distribution the control, and a value of 1 means the opposite (all samples in the treatment are higher than all samples in the control). In addition, threshold values can be defined [42] for the interpretation of Cliff's delta effect size as "negligible" ($|d| < 0.147$), "small" ($|d| < 0.33$), "medium" ($|d| < 0.474$), and "large" ($|d| \geq 0.474$).

We record an $\hat{A}_{12}$ value and a Cliff's delta value for each pair-wise comparison between our approach and the baseline, between our approach and random search and between the baseline and random search for each performance indicator (recall, precision and F-measure) in each case study.

4.5. Implementation

To perform a fair comparison between the approach and the baseline in the two case studies, it is necessary to properly calibrate the parameters of the evolutionary algorithm, LDA, and LSI.

Regarding the evolutionary algorithm, we have chosen the parameter settings that are commonly used in the literature [43, 44, 15, 16, 21] for evolutionary algorithms and LSI. The settings are shown in Table 1. The top part of the table shows the parameters related to the evolutionary algorithm such as the population size, the number of parents, or the crossover and mutation probabilities.

Table 1 also shows the LDA and LSI hyper-parameters. To calibrate the hyper-parameters, we follow the recommendations of Biggers et al. [23], which were designed for dealing with code software artifacts. In addition, as recommended by Panichella

Table 1: Parameter settings

	Parameter description	Value
Evolutionary algorithm	$Size$: Population Size	100
	r: Solutions replaced at population size	2
	μ: Number of Parents	2
	λ: Number of offspring from μ parents	2
	$p_{crossover}$: Crossover probability	0.9
	$p_{mutation}$: Mutation probability	0.1
LDA	k: Number of topics	300
	α: Topic distributions per document	1.0
	β: Distribution terms per topic	0.5
	σ: Number of sweeps over the corpus	300
LSI	k: Number of dimensions	100

et al. [24], we perform a trial-and-error procedure by changing the values of the hyper-parameters for other recommended values [23] in order to ensure that the chosen calibration leads to a balance between performance in the solution quality and computational cost.

We used the Eclipse Modeling Framework [45] to manipulate the software models in the implementation of our approach and the baseline. The techniques used to process the NL were implemented using OpenNLP [46] for the POS-Tagger, and the English (Porter2) algorithm for stemming [47]. The LDA fitness was implemented using JGibbLDA [22]. LSI was implemented using the Efficient Java Matrix Library (EJML) [48]. The genetic operations are built upon the Watchmaker Framework for Evolutionary Computation [49]. An independent run of the baseline comprises more than 1406 lines of code, whereas an independent run of our approach comprises more than 2488 lines of code. Just the implementation of the LDA fitness comprises more than 1082 lines of code.

In this work, we put the focus on performance measurements and optimizing the quality of the solution (i.e., obtaining a solution that is more similar to the one from the oracle in terms of precision and recall) instead of the algorithm speed (or search effort). After running some prior tests for our approach and the baseline in the two case studies to determine the algorithm convergence time (and adding a margin to ensure convergence), we allocated a fixed amount of wall clock time (80 seconds) to stop the execution of the evolutionary algorithm. The execution was performed using a Mac Pro computer with an Intel Xeon E5-2697 V2 processor (clock speeds 2.7 GHz and 12 cores) and 64 GB of RAM. The computer was running macOS Catalina (10.15.4) as the hosting Operative System and the Java(TM) SE Runtime Environment (build 1.8.0_77).

Due to confidentiality agreements with our industrial partners, the data set and part of the implementation are limited. The CSV files used as input in the statistical analysis as well as an open-source implementation of the LDA fitness function and the baselines are available here: `https://bitbucket.org/svitusj/tm-fitness`

5. Results

5.1. Measurements report

Table 2 shows the mean values and standard deviations of recall, precision, and F-measure for our LDA-guided approach, the LSI-guided baseline, and Random Search. In addition, the distribution of the obtained results is depicted in the box-plots that can be seen in Figure 7.

On the first case study (models for code generation) the LDA fitness guides the approach to average values of 28.18%±14.17% recall, 26.68%±14.88% precision, and 23.34%±10.82% F-measure, and the LSI baseline guides the approach to average values of 34.34%±15.13% recall, 33.06%±12.47% precision, and 30.67%±10.82% F-measure.

On the second case study (models for interpretation), the LDA fitness guides the approach to average values of 65.51%±13.22% recall, 59.21%±13.97% precision, and 60.63%±9.68% F-measure, and the LSI baseline guides the approach to average values of 45.72%±16.59% recall, 49.86%±14.63% precision, and 44.30%±11.45% F-measure.

RQ_1 answer: The results reveal that the baseline outperforms LDA in the three performance measurements in the first case study (models for code generation). In contrast, LDA outperforms the baseline in the three performance measures in the second case study (models for interpretation).

5.2. Statistical significance and effect size

The following paragraphs outline the statistical significance and effect size of the obtained results.

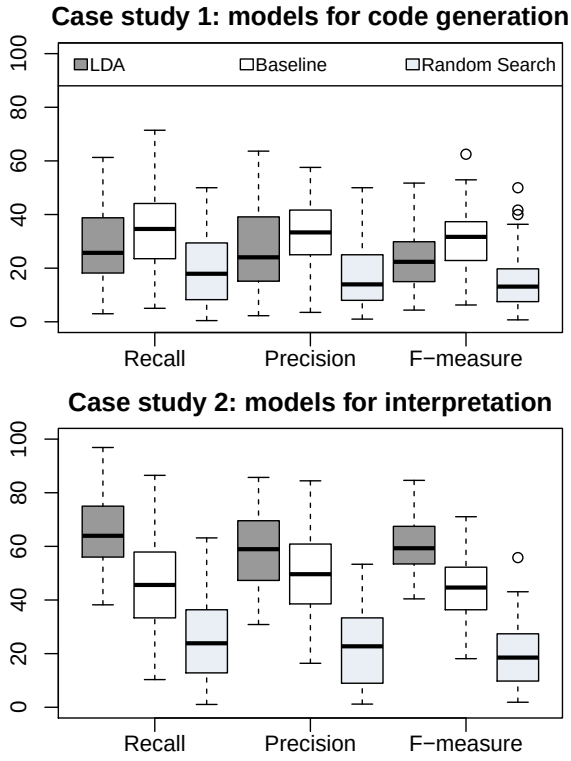

Figure 7: Performance achieved for each case study

Table 2: Mean values and standard deviations for recall, precision, and the F-measure for each case study

	Case study 1: models for code generation		
	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)
LDA	28.18 ± 14.17	26.68 ± 14.88	23.34 ± 10.82
Baseline	34.34 ± 15.13	33.06 ± 12.47	30.67 ± 10.82
Random Search	19.71 ± 13.30	16.84 ± 11.91	14.55 ± 9.78
	Case study 2: models for interpretation		
	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)
LDA	65.51 ± 13.22	59.21 ± 13.97	60.63 ± 9.68
Baseline	45.72 ± 16.59	49.86 ± 14.63	44.30 ± 11.45
Random Search	24.46 ± 14.13	22.23 ± 13.80	19.53 ± 11.60

Regarding statistical significance, the *p-values* obtained through the Quade test for our LDA-guided approach, the LSI-guided baseline, and Random Search in the first case study (models for code generation) are $1.472x10^{-12}$ for recall, $8.27x10^{-16}$ for precision, and $< 2.2^{-16}$ for the F-measure.

In the second case study (models for interpretation), the Quade test obtains *p-values* of $< 2.2^{-16}$ for recall, precision, and the F-measure.

Regarding the effect size, Table 3 shows the obtained values for the $\hat{A}_{12}$ and Cliff's delta measurements associated with the three reported performance measurements obtained by our LDA-guided approach, the LSI-guided baseline, and Random Search in both case studies.

In the first case study (models for code generation), the $\hat{A}_{12}$ measurement obtains values of 0.3833 for recall, 0.3645 for precision, and 0.3103 for the F-measure, implying that the baseline obtains better values than our approach in 61.66% of the runs for recall, 63.55% of the runs for precision, and 68.97% of the runs for the F-measure. In addition, the Cliff's delta measurement indicates a small effect size for recall, a small effect size for precision, and a medium effect size for the F-measure. When our approach is compared against Random Search, the $\hat{A}_{12}$ measurement shows that our approach obtains better values than Random Search in all performance measurements. Specifically, our approach obtains better values than Random Search in 66.96% of the runs for recall, 69.45% of the runs for precision, and 73.68% of the runs for the F-measure. The Cliff's delta measurement indicates a medium effect size in all performance measurements. When the baseline is compared to Random Search, the baseline obtains better results in all performance measurements, and the Cliff's delta measurement indicates a large effect size.

In the second case study (models for code generation), the $\hat{A}_{12}$ measurement obtains values of 0.8223 for recall, 0.6705 for precision, and 0.8610 for the F-measure, implying that the LDA approach obtains better values than the baseline in 82.23% of the runs for recall, 67.05% of the runs for precision, and 86.10% of the runs for the F-measure. In addition, the Cliff's delta measurement indicates a large effect size for recall, a medium effect size for precision, and a large effect size for the F-measure. LDA also outperforms Random Search, which shows a large effect size according to the Cliff's delta measurement. In this case study, the baseline also outperforms Ran-

Table 3: Effect size measures for comparing recall, precision and F-measure in the two case studies

	Case study 1: models for code generation LDA vs Baseline		
	Recall	Precision	F-measure
$\hat{A}_{12}$	0.3833	0.3645	0.3103
Cliff's Delta	-0.2334 (small)	-0.2710 (small)	-0.3793 (medium)
	LDA vs Random Search		
$\hat{A}_{12}$	0.6696	0.6945	0.7368
Cliff's Delta	0.3391 (medium)	0.3889 (medium)	0.4736 (medium)
	Baseline vs Random Search		
$\hat{A}_{12}$	0.7651	0.8232	0.8647
Cliff's Delta	0.5302 (large)	0.6464 (large)	0.7294 (large)
	Case study 2: models for interpretation LDA vs Baseline		
	Recall	Precision	F-measure
$\hat{A}_{12}$	0.8223	0.6705	0.8610
Cliff's Delta	0.6446 (large)	0.3409 (medium)	0.7220 (large)
	LDA vs Random Search		
$\hat{A}_{12}$	0.9858	0.9697	0.9960
Cliff's Delta	0.9717 (large)	0.9394 (large)	0.9920 (large)
	Baseline vs Random Search		
$\hat{A}_{12}$	0.8295	0.9093	0.9288
Cliff's Delta	0.6590 (large)	0.8185 (large)	0.8576 (large)

dom Search by a large effect size.

RQ$_2$ answer: Since the Quade test *p-values* are smaller than the 0.05 statistical significance threshold for all performance indicators in both case studies, we can state that there are significant differences in performance among the LDA approach, the LSI baseline, and Random Search when locating features in the two kinds of software models from the case studies under research.

In addition, from the effect size analysis, we can conclude how much the performance is influenced by using LDA compared to the baseline in the two kinds of models. In the first case study, the baseline obtains better results than the LDA approach in over 60% of the runs for the three measurements, albeit the effect size ranges from small to medium. On the other hand, in the second case study, the LDA approach is the one that obtains better results in the majority of the runs, and the effect size ranges from medium to large.

6. Discussion

In order to structure the discussion, we are going to consider the four main limitations that have been studied in the literature for the last 20 years of application of LDA to code problems [25].

6.1. Data sparsity

Source code tends to suffer from sparsity of data. In particular, the vocabulary in use for source code text is limited. Therefore, approaches that deal with source code often tune the parameters of LDA, even more so than approaches that deal with natural language [7, 8]. From an analytical perspective, data sparsity raises major concerns about the feasibility of using approaches such as LDA or LSI to generate semantically coherent topics in source code [50, 51, 52, 53].

The main difference between code and software models is that the latter use terms that are closer to the problem domain. Even though this is true both for code generation models and interpreted models, the results are significantly distinct. The lexicon in use for code generation models is more limited than that of models for interpretation. This is due to the fact that the majority of features present terms that do not have a correspondence with the terms that appear in code generation models. These terms only appear in the model-to-code transformation process.

We reviewed the model-to-code transformation process and in most cases, we found that the model-to-code transformation process increases the information included in the models (around 70%). This coincides with the main idea of model-driven software development. Models abstract from implementation details and the model-to-code transformation process adds those details. However, in the rest of the cases, the engineers from CAF have embedded domain information in the model-to-code transformation process. For example, when the concept compressor appears in the model, the model-to-code transformation process has hardcoded the code that generates the state machine to regulate the compressor. In a way, this is an abuse of the model-to-code transformation process. This is like programming in imper-

ative despite using an object-oriented programming language.

When we asked the engineers why they do it this way, they stated that is because that functionality has not changed so far but they would include the information in the models (instead of hardcoding the functionality in the model-to-code transformation process) if the functionality changes. This phenomenon affects more the topics than the similarity of terms. This is because some topics change but there are still some terms in common. We checked that this phenomenon does not occur in models for interpretation. If we omit the cases in which this phenomenon occurs in the results of the first case study (models for code generation), the LDA fitness guide the approach to average values of 40.61% in recall, 37.94% in precision and 36.94% in F-measure. The Quade test obtains *p-values* smaller than the 0.05 statistical significance threshold for all performance indicators, so we can state that there are statistical differences in performance. The $\hat{A}_{12}$ measurement when the LDA-guided approach is compared with the LSI-guided baseline obtains values of 0.6503 for recall, 0.6252 for precision, and 0.6939 for the F-measure, implying that the LDA approach obtains better values than baseline when software models for generation are used.

These results lead us to recommend that the model-to-code transformation process should be checked in other code generation contexts. If the engineers of our industrial partner embedded domain information in the model-to-code transformation process, this could happen in other contexts.

6.2. Implementation complexity

The mathematical structure of generative topic modeling techniques is not intuitive or easy to grasp and understand [1]. The added theoretical complexity is the main reason why topic modeling is often used as a black box with little to no customization in order to deal with special corpora coming from software systems [51, 50]. In our work, we have also used topic modeling as a black box. We have not introduced any modification so as to take into account the particularities of the software models in use.

6.3. Calibration

Most of the topic modeling techniques from the literature require an exhaustive calibration process. During this process, several parameters are calibrated until the desired output is reached [1, 54, 55]. Such parameters (for instance, α, β, K, and the number of iterations) often need to be simultaneously calibrated until the best configuration settings for the task under research are identified. Often, researchers overcome the problem by relying on heuristics stemming from experimental settings [56, 57, 50]. Other times, the topic modeling algorithm is run multiple times, and then the average (or best) performance of the runs is taken as a result. However, such heuristics do not necessarily guarantee the success of the techniques in all experimental settings. In addition, there is no objective way to determine the optimal amount of algorithm runs, nor an objective criterion for choosing the best model. Even with the support of automated calibration strategies [58, 24, 59], the calibration process is still computationally expensive and not guaranteed to find an optimal solution.

Regarding the calibration of our work, we have used calibration guidelines originally designed for dealing with code software artifacts. Even though it is not possible to ensure that the calibration is optimal, the calibration has yielded better results than those reached without calibration. This issue suggests that the efforts in calibration with code artifacts can be transferred to calibration for model artifacts.

6.4. Stability

Topic modeling techniques rely on intractable stochastic inference strategies in order to generate topics. Hence, it cannot be guaranteed that different runs of topic modeling techniques will generate similar distributions of topics [60, 1]. This issue raises major practicality concerns, since solutions might vary depending on the starting inferred conditions.

A major difference between FL approaches for code and FL approaches for software models is the granularity of the artifacts: regarding code, FL approaches look for the most relevant method for the implementation of the feature, and in the case of software models, FL approaches look for the most relevant model

fragment for the implementation of the feature. The sheer amount of potential solution model fragments leads to the usage of evolutionary algorithms as a means of efficiently exploring the large search space. With each new iteration of the evolutionary algorithm, a new run of LDA classifies the new model fragment population. The evaluation of a same model fragment by LDA in the context of different model fragment populations mitigates the stochastic inference of LDA. In any case, more experiments in this line of research are required to study the stability of the technique.

6.5. Other issues

Outside of the prior categories, the analysis of the results has led to the identification of other issues, common to FL approaches and scenarios, that are negatively affecting the results of the approach and baseline: vocabulary mismatch and tacit knowledge.

Vocabulary mismatch happens when the language in use in the feature descriptions and the software artifacts is not strictly the same, or in other words, when different terms are used to describe the same actions and components in the natural language of the descriptions and in the language in use in the models. This issue happens due to the fact that the artifacts are created with different purposes in mind, and when the artifacts are created by different software engineers. For example, both circuit breaker and HSBC refer to an on-off switch.

Tacit knowledge happens when the software engineers make assumptions about the domain knowledge of other software engineers, based on their own knowledge and expertise. Hence, when feature descriptions are written, parts of the domain knowledge are assumed by the domain experts, and not portrayed in the writing of the descriptions, but nonetheless used to build the final software artifacts. This leads to yet another textual mismatch between the descriptions, which contain only part of the domain knowledge, and the models, which contain the entirety of the domain knowledge. For example, software engineers use the term doors in feature descriptions, but they do not clarify whether the doors belong to one side or another of the train, or if the doors refer to cabin doors, car doors, or both.

6.6. Focus group interview

To obtain qualitative data from practitioners, we ran a focus group with four engineers from the partner companies (two for each case study). The engineers from the first case study have spent 7 and 8 years developing software. The engineers from the second case study have spent 4 and 15 years developing software. The aim of the questions was to acquire feedback from the engineers about the results that were obtained with topic modeling (LDA) and the baseline (LSI). Specifically, the focus group was composed of the following open questions: (1) How do you feel about the results of each approach? (2) What challenges did you have in understanding the results of each approach? and (3) Why would you choose the results of one approach over the results of the other approach?

All engineers stated that they preferred the results of topic modeling rather than the results of LSI, indicating that topic modeling seemed to understand the context better than LSI. As an example, feature descriptions and model fragments with the terms *Pantograph* and *Circuit Breaker* in some cases refer to model elements that are organized for the ignition system of the train, whereas in other cases these same elements are organized for emergency systems of the train, such as fire control. In all cases, the majority of terms are *Pantograph* and *Circuit Breaker*, whereas the terms that are related to the ignition system or the emergency system are the minority. While LSI would be influenced by the majority of the terms without distinguishing that the terms are from different contexts, topic modeling can identify different topics for the ignition system and the emergency system. Thus, topic modeling is able to distinguish model fragments and feature descriptions that belong to each of these two contexts.

The other aspect that the engineers highlighted as an advantage of topic modeling over LSI is that, in LSI, it is possible to count terms but not to group them in topics as topic modeling does. Not only engineers found this grouping in topics useful, but in addition, they recognized that some of the topics uncovered by topic modeling challenged their intuition: some of the topics identified by topic modeling would not have been naturally inferred or proposed by the

engineers, but were recognized by the engineers as relevant topics for the case studies.

7. Threats to validity

In this section, we use the classification of threats to validity suggested by De Oliveira et al. [61] to acknowledge the threats to validity of our work.

Conclusion validity: We considered random variation with 30 independent runs [37] for each feature description in our approach and the baseline in the two case studies. We used measurements (recall, precision and F-measure) that are widely accepted in the software engineering research community [37] to analyze the obtained confusion matrix. We also used statistical and effect size measurements (Quade test, $\hat{A}_{12}$, and Cliff's delta) following accepted guidelines [37].

Internal Validity: We followed the same evaluation process for our approach and the baseline in the two case studies. We addressed the poor parameter settings threat using values from the literature in our approach and the baseline. Default values are good enough to measure the performance of location techniques as suggested by Arcuri and Fraser [37]. With regard to the time needed to produce a solution (stop condition), we used 80 seconds since it was the time needed to converge in the two industrial case studies. Nevertheless, we have not yet researched how this time scales in other industrial case studies with a larger search space size or a larger solution size. Moreover, we addressed the threat related to the lack of real problem instances by applying the evaluation of this paper to two industrial case studies.

Construct validity: We performed our evaluation around three widespread measurements (recall, precision, and F-measure) in the software engineering research community [37] to address the identified threat of the lack of assessing the validity of cost measures. Moreover, we performed a fair comparison between our approach and the baseline in the two case studies.

External Validity: We evaluated our approach in two real-world industrial case studies to mitigate the threat of the lack of a clear object selection strategy. With regard to the extent to which it is possible to generalize the results, the results depend on the quality of the queries as occurs in other works [23, 28]. Poor feature descriptions lead to the selection of irrelevant model fragments. It is also worth noting that the domain-specific language used for the model elements and feature descriptions must use the same terminology to rank relevant model fragments with LDA and LSI. To narrow the gap between the model elements and the feature descriptions, different NLP techniques (i.e., tokenizers, stemming, and POS tagging techniques) are applied. Also with regard to the mitigation of the generalization threat, our approach has been evaluated in two model-based case studies from two different domains. Our approach can be applied to any model that conforms to MOF (the OMG metalanguage for defining modeling languages), and the text elements that are associated with the models are extracted automatically by using the reflective methods provided by the Eclipse Modeling Framework. The requisites to apply our approach are that the set of models must conform to MOF, and the feature description must be provided in natural language. Nevertheless, our approach should be replicated with case studies from other domains before assuring its generalization.

8. Related work

Regarding Feature Location, some works [62, 63] propose automated approaches for the identification of features within a set of existing software variants. The features are then used to generate Software Product Lines (SPLs). Martínez et. al., [63] evaluate their approach through two different real-world systems, comparing the number of model elements of the original system with the model elements of the blocks that can be identified in the generated SPL. In [62], Assunçao et. al. present an automated approach for SPL generation based on an evolutionary algorithm that leverages existing UML class diagrams and a list of well defined features to generate a Feature Model. The authors of [62] evaluate their approach through 10 different case study applications. These studies use UML models as input, and are focused on generating Feature Models for SPLs. In our work, we do not aim to generate Feature Models for SPLs, but we

rather put the focus of our Feature Location efforts in locating model fragments that implement features (which description is provided) by using LDA as a fitness function for an evolutionary algorithm. In addition, our work is built on two different kinds of software models (models for code generation and models for interpretation) distinct from UML.

Other of our previous works deal with Feature Location in industrial software models. Some of these works generate and rank model fragments that are relevant for the location of a feature, guiding the process through the usage of different approaches such as clustering [32] and empirical learning [33], or through the evaluation of a combination of similitude, understandability, and timing measurements [15]. Another work [16] explores different search strategies with a fixed fitness function based on similitude. The authors of [64] propose expanding the available information for the Feature Location task through the usage of models at run-time, and the authors of [21] introduce collaboration in Feature Location for complex location tasks that exceed the knowledge of individual software engineers. In [65], the Feature Location process is guided through the analysis of the sustainability of long-living software systems. Finally, the study in [36] provides measurements to describe the model fragments in use for Feature Location in software models. While these studies deal with Feature Location in industrial models, they do not study Feature Location for different kinds of software models (models for code generation and models for interpretation) as this paper does. In addition, none of these works study the application of LDA as a fitness function to guide the evolutionary algorithm.

Regarding Topic Modeling, it soon became popular within the software engineering community [2, 66]. A recent survey [2] investigated previous works that are related to topic modeling based on LDA to perform source code analysis. Lukins et al. [4] present a Feature Location technique that uses LDA to identify methods affected by a bug. In [67], LDA was used to extract topics from source code and to perform visualization of software similarity. There has also been research on the appropriate configuration of LDA in source code. Grant et al. [57] were concerned about K (the number of topics). Panichella et al. [24] proposed LDA-GA, a genetic algorithm approach that searches for the appropriate LDA hyperparameters for software engineering specific tasks. Biggers et al [23] studied different configurations of LDA to retrieve features from open source Java systems.

Although topic models based on LDA have an important role within the software engineering community, previous works have been focused on source code as the main software engineering artifact. Unlike previous works, we study the use of LDA in a different software engineering artifact, software models. Specifically, we study the use of LDA for feature location in both software models for generation and software models for interpretation. In addition, we do not research the configuration of LDA, but rather use it as a means of guiding the evolutionary algorithm in our approach.

9. Conclusion and future work

LDA has not been studied in the literature when software models are used as the main software engineering artifact even though models are a popular asset in software development. In this paper, we have filled this research gap by proposing an approach that uses LDA to guide an evolutionary algorithm that locates features in two kinds of software models (for code generation and for interpretation). The evaluation has been performed in two real-world industrial case studies. Our results have shown that LDA significantly outperforms the LSI-based baseline when software models for interpretation are used. In contrast, LDA does not outperform the baseline when software models for code generation are used. We have found that the model-to-code transformation process increases the information included in the models. This phenomenon affects the topics more (LDA) than the similarity of terms (LSI).

We have also discussed our results considering the following aspects: data sparsity, implementation complexity, calibration, and stability. Our findings about transferring calibration approaches from code to models can help other researchers to not dismiss those approaches even if they were originally designed

with code artifacts in mind. Our findings about compensating instability with the evolutionary algorithm can also help researchers working in both models and code. Instability is an open issue when LDA is applied to code. Finally, our findings on checking the model-to-code transformation process lead us to recommend that the model-to-code transformation process should be checked in other code generation contexts and domains when LDA and models for code generation are used.

It is important to highlight that the models taken as input in this work are used in code generation and in model interpretation contexts. For this reason, these models are richer in terms than other kinds of models such as sketches for analysis or models at runtime (which are partially connected to the run-time implementation). We cannot generalize the outcomes of LDA when meeting these other types of models. Hence, as future work, we also plan on researching these other types of models.

Acknowledgements

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the Project ALPS (RTI2018-096411-B-I00).

References

[1] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.

[2] H. Jelodar, Y. Wang, C. Yuan, X. Feng, Latent dirichlet allocation (LDA) and topic modeling: models, applications, a survey, Multimedia Tools and Applications 78 (2019) 15169–15211.

[3] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process 25 (2013) 53–95.

[4] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, Source code retrieval for bug localization using latent dirichlet allocation., in: A. E. Hassan, A. Zaidman, M. D. Penta (Eds.), 15th Working Conference on Reverse Engineering, IEEE Computer Society, 2008, pp. 155–164.

[5] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic, An information retrieval approach to concept location in source code, in: Proceedings of the 11th Working Conference on Reverse Engineering, WCRE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 214–223.

[6] H. U. Asuncion, A. U. Asuncion, R. N. Taylor, Software traceability with topic modeling, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 95–104. URL: https://doi.org/10.1145/1806799.1806817. doi:10.1145/1806799.1806817.

[7] M. Gabel, Z. Su, A study of the uniqueness of source code, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 147–156. URL: https://doi.org/10.1145/1882291.1882315. doi:10.1145/1882291.1882315.

[8] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 837–847. URL: http://dl.acm.org/citation.cfm?id=2337223.2337322.

[9] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Synthesis Lectures on Software Engineering 1 (2012) 1–182.

[10] D. D. Ruscio], R. F. Paige, A. Pierantonio, Guest editorial to the special issue on success stories in model driven engineering, Science of Computer Programming 89 (2014) 69 – 70. Special issue on Success Stories in Model Driven Engineering.

[11] J. Krüger, T. Berger, T. Leich, Features and how to find them: A survey of manual feature location, in: Software Engineering for Variability Intensive Systems - Foundations and Applications, Taylor & Francis Group, 2019, pp. 153–172.

[12] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Transactions on Software Engineering 33 (2007) 420–432.

[13] J. Wang, X. Peng, Z. Xing, W. Zhao, An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions, in: Proceedings of the 27th Conference on Software Maintenance, IEEE, 2011, pp. 213–222. doi:10.1109/ICSM.2011.6080788.

[14] T. K. Landauer, P. W. Foltz, D. Laham, An introduction to latent semantic analysis, Discourse processes 25 (1998) 259–284.

[15] F. Pérez, R. Lapeña, J. Font, C. Cetina, Fragment retrieval on models for model maintenance: Applying a multi-objective perspective to an industrial case study, Information & Software Technology 103 (2018) 188–201.

[16] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Achieving feature location in families of models through the use of search-based software engineering, IEEE Transactions on Evolutionary Computation PP (2017) 1–1.

[17] A. Vargha, H. D. Delaney, A critique and improvement of the cl common language effect size statistics of mcgraw and wong, Journal of Educational and Behavioral Statistics 25 (2000) 101–132.

[18] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions., Psychological Bulletin 114 (1993) 494.

[19] N. Cliff, Ordinal methods for behavioral data analysis. (1996).

[20] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Softw. Test. Verif. Reliab. 24 (2014) 219–250.

[21] F. Pérez, J. Font, L. Arcega, C. Cetina, Collaborative feature location in models through automatic query expansion, Automated Software Engineering 26 (2019) 161–202.

[22] Jgibblda. a java implementation of latent dirichlet allocation (LDA) using gibbs sampling for parameter estimation and inference, http://jgibblda.sourceforge.net, 2020.

[23] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, N. A. Kraft, Configuring latent dirichlet allocation based feature location, Empirical Softw. Engg. 19 (2014) 465–500.

[24] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, A. D. Lucia, How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 522–531.

[25] A. Mahmoud, G. Bradshaw, Semantic topic models for source code analysis, Empirical Softw. Engg. 22 (2017) 1965–2000.

[26] A. Hulth, Improved automatic keyword extraction given more linguistic knowledge, in: Proceedings of the 2003 conference on Empirical methods in natural language processing, 2003, pp. 216–223.

[27] R. Lapeña, J. Font, O. Pastor, C. Cetina, Analyzing the impact of natural language processing over feature location in models, in: GPCE 2017 - 16th International Conference on Generative Programming: Concepts & Experience, 2017.

[28] B. Sisman, A. C. Kak, Assisting code search with automatic query reformulation for bug localization, in: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR, 2013, pp. 309–318. doi:10.1109/MSR.2013.6624044.

[29] E. Hill, L. Pollock, K. Vijay-Shanker, Automatically capturing source code context of nl-queries for software maintenance and reuse, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, 2009, pp. 232–242. doi:10.1109/ICSE.2009.5070524.

[30] M. Affenzeller, S. M. Winkler, S. Wagner, A. Beham, Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications, CRC Press, 2009.

[31] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Feature Location in Model-Based Software Product Lines Through a Genetic Algorithm, in: Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness, 2016.

[32] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Feature location in models through a genetic algorithm driven by information retrieval techniques, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, ACM, 2016, pp. 272–282. doi:10.1145/2976767.2976789.

[33] A. C. Marcén, J. Font, O. Pastor, C. Cetina, Towards feature location in models through a learning to rank approach, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, SPLC '17, 2017, p. 57–64. doi:10.1145/3109729.3109734.

[34] H. Störrle, On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters, 17th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2014).

[35] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining Version Histories to Guide Software Changes, in: Proceedings of the 26th International Conference on Software Engineering, 2004.

[36] M. Ballarín, A. C. Marcén, V. Pelechano, C. Cetina, Measures to report the location problem of model fragment location, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018, 2018, pp. 189–199. doi:10.1145/3239372.3239397.

[37] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Engineering 18 (2013) 594–623.

[38] G. Salton, M. J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill, Inc., New York, NY, USA, 1986.

[39] D. Falessi, G. Cantone, G. Canfora, Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques, IEEE Transactions on Software Engineering 39 (2011) 18–44.

[40] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power, Information Sciences 180 (2010) 2044–2064.

[41] W. Conover, Practical nonparametric statistics, Wiley series in probability and statistics, 3. ed ed., Wiley, New York, NY [u.a.], 1999.

[42] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys, in: annual meeting of the Florida Association of Institutional Research, 2006, pp. 1–33.

[43] A. S. Sayyad, J. Ingram, T. Menzies, H. Ammar, Scalable product line configuration: A straw to break the camel's back, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 2013, pp. 465–474. doi:10.1109/ASE.2013.6693104.

[44] C. Carpineto, G. Romano, A survey of automatic query expansion in information retrieval, ACM Comput. Surv. 44 (2012) 1:1–1:50.

[45] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, 2nd ed., Addison-Wesley Professional, 2009.

[46] Apache opennlp: Toolkit for the processing of natural language text, https://opennlp.apache.org/, 2020.

[47] English (porter2) stemming algorithm, http://snowball.tartarus.org/algorithms/english/stemmer.html, 2020.

[48] Efficient java matrix library, http://ejml.org/, 2020.

[49] D. Dyer, The watchmaker framework for evolutionary computation (evolutionary/genetic algorithms for java), http://watchmaker.uncommons.org/, 2016.

[50] A. Abadi, M. Nisenson, Y. Simionovici, A traceability technique for specifications, in: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, ICPC '08, IEEE Computer Society, USA, 2008, p. 103–112. URL: https://doi.org/10.1109/ICPC.2008.30. doi:10.1109/ICPC.2008.30.

[51] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, S. Panichella, Using IR methods for labeling source code artifacts: Is it worthwhile?, in: D. Beyer, A. van Deursen, M. W. Godfrey (Eds.), IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012, IEEE Computer Society, 2012, pp. 193–202. URL: https://doi.org/10.1109/ICPC.2012.6240488. doi:10.1109/ICPC.2012.6240488.

[52] G. Maskeri, S. Sarkar, K. Heafield, Mining business topics in source code using latent dirichlet allocation, in: Proceedings of the 1st India Software Engineering Conference, ISEC '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 113–120. URL: https://doi.org/10.1145/1342211.1342234. doi:10.1145/1342211.1342234.

[53] A. Mahmoud, N. Niu, On the role of semantics in automated requirements tracing, Requir. Eng. (2015) 281–300.

[54] A. Kuhn, S. Ducasse, T. Gírba, Inf. Softw. Technol. 49 (2007) 230–243.

[55] T. Hofmann, Probabilistic latent semantic indexing, in: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '99, Association for Computing Machinery, New York, NY, USA, 1999, p. 50–57. URL: https://doi.org/10.1145/312624.312649. doi:10.1145/312624.312649.

[56] D. Andrzejewski, A. Mulhern, B. Liblit, X. Zhu, Statistical debugging using latent topic models, in: J. N. Kok, J. Koronacki, R. L. d. Mantaras, S. Matwin, D. Mladenič, A. Skowron (Eds.), Machine Learning: ECML 2007, Springer Berlin Heidelberg, 2007, pp. 6–17.

[57] S. Grant, J. R. Cordy, Estimating the optimal number of latent concepts in source code analysis, in: Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10, IEEE Computer Society, USA, 2010, p. 65–74. URL: https://doi.org/10.1109/SCAM.2010.22. doi:10.1109/SCAM.2010.22.

[58] S. Lohar, S. Amornborvornwong, A. Zisman, J. Cleland-Huang, Improving trace accuracy through data-driven configuration and composition of tracing features, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, Association for Computing Machinery, New York, NY,

USA, 2013, p. 378–388. URL: https://doi.org/10.1145/2491411.2491432. doi:10.1145/2491411.2491432.

[59] A. Agrawal, W. Fu, T. Menzies, What is wrong with topic modeling? and how to fix it using search-based software engineering, Information and Software Technology 98 (2018) 74 – 88.

[60] H. M. Wallach, D. M. Mimno, A. McCallum, Rethinking LDA: Why priors matter., in: Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, A. Culotta (Eds.), Advances in Neural Information Processing Systems 22 (NIPS 2009), Curran Associates, Inc., 2009, pp. 1973–1981.

[61] M. de Oliveira Barros, A. C. D. Neto, Threats to Validity in Search-based Software Engineering Empirical Studies, Technical Report 0006/2011, 2011.

[62] W. K. Assunção, S. R. Vergilio, R. E. Lopez-Herrejon, Automatic extraction of product line architecture and feature models from uml class diagram variants, Information and Software Technology 117 (2020) 106198.

[63] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Automating the extraction of model-based software product lines from model variants (t), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, 2015, pp. 396–406.

[64] L. Arcega, J. Font, Ø. Haugen, C. Cetina, Leveraging models at run-time to retrieve information for feature location, in: Proceedings of the 10th International Workshop on Models@run.time co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 29, 2015, 2015, pp. 51–60.

[65] C. Cetina, J. Font, L. Arcega, F. Pérez, Improving feature location in long-living model-based product families designed with sustainability goals, Journal of Systems and Software 134 (2017) 261–278.

[66] J. C. Campbell, A. Hindle, E. Stroulia, Chapter 6 - latent dirichlet allocation: Extracting topics from software engineering data, in: C. Bird, T. Menzies, T. Zimmermann (Eds.), The Art and Science of Analyzing Software Data, Morgan Kaufmann, Boston, 2015, pp. 139 – 159. URL: http://www.sciencedirect.com/science/article/pii/B9780124115194000069. doi:https://doi.org/10.1016/B978-0-12-411519-4.00006-9.

[67] M. Gethers, D. Poshyvanyk, Using relational topic models to capture coupling among classes in object-oriented software systems, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10, IEEE Computer Society, USA, 2010, p. 1–10. URL: https://doi.org/10.1109/ICSM.2010.5609687. doi:10.1109/ICSM.2010.5609687.