Edinburgh Research Explorer

Autofolding for Source Code Summarization

Autofolding for Source Code Summarization

Jaroslav Fowkes*, Pankajan Chanthirasegaran*, Razvan Ranca†,
Miltiadis Allamanis*, Mirella Lapata* and Charles Sutton*
*School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, UK
{jaroslav.fowkes, pchanthi, m.allamanis, csutton}@ed.ac.uk; mlap@inf.ed.ac.uk
†Tractable, Oval Office, 11-12 The Oval, London, E2 9DT, UK
razvan@tractable.io

Abstract—Developers spend much of their time reading and browsing source code, raising new opportunities for summarization methods. Indeed, modern code editors provide code *folding*, which allows one to selectively hide blocks of code. However this is impractical to use as folding decisions must be made manually or based on simple rules. We introduce the *autofolding problem*, which is to automatically create a code summary by folding less informative code regions. We present a novel solution by formulating the problem as a sequence of AST folding decisions, leveraging a scoped topic model for code tokens. On an annotated set of popular open source projects, we show that our summarizer outperforms simpler baselines, yielding a 28% error reduction. Furthermore, we find through a case study that our summarizer is strongly preferred by experienced developers. More broadly, we hope this work will aid program comprehension by turning code folding into a usable and valuable tool.

I. INTRODUCTION

Engineering large software systems presents many challenges due to the inherent complexity of software. Because of this complexity, programmers tend to spend more time reading and browsing code than actually writing it [1], [2]. Despite much research [3], there is still a large need for better tools that aid program comprehension, thereby reducing the cost of software development.

A key insight is that source code is written to be understood not only by machines, but also by humans. Programmers devote significant time and attention to writing their code in an idiomatic and intuitive way that can be easily understood by others — *source code is a means of human communication.* This fact raises the intriguing possibility that technology from the natural language processing (NLP) community can be adapted to help developers make sense of large repositories of code. Often during development and maintenance, developers skim the code in order to quickly understand it [4]. A good *summary* of the source code aims to support this use case: by eliding less-important details, a summary can be easier to read quickly and help the developer to gain a high-level conceptual understanding of the code.

Source code summarization has potential for valuable applications in many software engineering tasks, such as: (a) *Understanding new code bases.* Often developers need to quickly familiarize themselves with the core parts of a large code base. This can happen when a developer is joining an existing project, or when a developer is evaluating whether to use a new software library. (b) *Code reviews.* Reviewers need to quickly understand the key changes before reviewing the details. (c) *Locating relevant code segments.* During program maintenance, developers often skim code, reading only a couple lines at a time, while searching for a code region of interest [4].

For this reason, many code editors include a feature called *code folding*, which allows developers to selectively display or hide blocks of source code. This feature is commonly supported in editors and is familiar to developers [5]–[7]. But in current Integrated Development Environments (IDEs), folding quickly becomes impractical because the folding decisions must be done manually by the programmer, or based on simple rules, such as folding code blocks based on depth [8], that some IDEs take automatically. This creates an obvious chicken-and-egg problem, because the developer must already understand the source file to decide what should be folded.

In this paper, we propose that code folding can be a valuable tool for aiding program comprehension, provided that folding decisions are made automatically based on the *code's content.* We consider the *autofolding problem*, in which the goal is to automatically create a code summary by folding non-essential code elements that are not useful on first viewing. To our knowledge, we are the first to systematically study and quantitatively compare different methods for the autofolding problem. An illustrative example is shown in Figure 1. To any Java developer the function of the `StatusLine` constructor and the `clone`, `getCode`, `getReason` and `toString` methods are obvious even without seeing their method bodies. One possible summary of this source file is shown in Figure 2.

The key problem in content-based autofolding is to determine which tokens in a file are most representative of its content. We compare two different content models for this task: a simple vector space model (VSM) and a topic model that, building on work in NLP summarization [9], endows different scopes (files, projects, and the corpus) with separate topics, allowing the model to separate out those tokens that are used most often in a particular file. We find that the summaries from the topic model are significantly better than those from the VSM.

```
1 /* Header */
2 package org.zoolu.sip.header;
3
4 /** SIP Status-line, i.e. the first
5  * line of a response message */
6 public class StatusLine {
7   protected int code;
8   protected String reason;
9
10   /** Construct StatusLine */
11   public StatusLine(int c, String r) {
12     code = c;
13     reason = r;
14   }
15
16   /** Create a new copy of the request-line */
17   public Object clone() {
18     return new StatusLine(getCode(), getReason());
19   }
20
21   /** Indicates whether some other Object
22    * is "equal to" this StatusLine */
23   public boolean equals(Object obj){
24     try {
25       StatusLine r = (StatusLine) obj;
26       if (r.getCode() == getCode()&&
27       r.getReason().equals(getReason()))
28         return true;
29       else
30         return false;
31     } catch (Exception e) {
32       return false;
33     }
34   }
35
36   public int getCode() {
37     return code;
38   }
39
40   public String getReason() {
41     return reason;
42   }
43
44   public String toString() {
45     return "SIP/2.0 " + code + " " + reason + "\r\n";
46   }
47 }
```

Figure 1: Original source code. A snippet from `bigbluebutton`'s `StatusLine.java`. We use this as a running example.

Previous work in code summarization has considered summarization using: (a) program slicing (i.e. hiding irrelevant lines of code for a chosen program path) [10], [11]; (b) natural language paraphrases [12], [13]; (c) short lists of keywords [14]–[17]; or (d) (potentially discontiguous) lines of code that match a user's query [18]. In contrast, our work is based on the idea that an effective summary can be obtained by carefully folding the original file — summarizing code *with code*. Our main contributions in this paper are:

- We introduce a novel autofolding method for source code summarization, called TASSAL[1], based on optimizing the similarity between the summary and the source file. Because of certain constraints among the folding decisions, we formulate this method as a contiguous rooted subtree problem (Section III-C). This is, to our knowledge, the first content-based autofolding method for code summarization.

[1] https://github.com/mast-group/tassal

```
1 /* Header...*/
2 package org.zoolu.sip.header;
3
4 /** SIP Status-line, i.e. the first...*/
6 public class StatusLine {
7   protected int code;
8   protected String reason;
9
10   /** Construct StatusLine...*/
11   public StatusLine(int c, String r) {...}
15
16   /** Create a new copy of the request-line ..*/
17   public Object clone() {...}
20
21   /** Indicates whether some other Object...*/
23   public boolean equals(Object obj){
24     try {
25       StatusLine r = (StatusLine) obj;
26       if (r.getCode() == (getCode()&&
27       r.getReason().equals(getReason()))
28         return true;
29       else
30         return false;
31     } catch (Exception e) {...}
34   }
35
36   public int getCode() {...}
39
40   public String getReason() {...}
43
44   public String toString() {...}
47 }
```

Figure 2: A summary of the file in Figure 1 (left) which results from folding lines 1, 4–5, 11–14, 21–22, 31–33, 36-38 and 40-42. The ellipses indicate folded segments of code.

- To determine which non-essential regions should be folded, we introduce a novel topic model for code (Section III-B), building on machine learning methods used in NLP [9], which separates tokens according whether they best characterize their file, their project, or the corpus as a while. This allows TASSAL summaries to focus on file-specific tokens.
- We perform a comprehensive evaluation of our method on a set of popular open source projects from GitHub (Section IV), and find that TASSAL performs better than simpler baselines (Section V) at matching human judgements, with a relative error reduction of 28%. Furthermore, in a user study with experienced developers, TASSAL is strongly preferred to the baselines.
- We created a live demo of TASSAL [19] to showcase how it can be used to summarize open-source Java projects on GitHub. Our demo can be found at http://groups.inf.ed.ac.uk/cup/tassal/demo.html and a video highlighting the main features of TASSAL can be found at https://youtu.be/_yu7JZgiBA4.

More broadly, we hope that this work will aid program comprehension by turning code folding, perhaps an overlooked feature, into a useful, usable and valuable tool.

II. RELATED WORK

The application of NLP methods to the analysis of source code text is only just beginning to be explored. Recent work has applied language modelling [20]–[24], natural language generation [12], [25], machine translation

[26], and topic modelling [27] to the text of source code from large software projects. A main challenge in this area is to adapt existing NLP techniques to source code text. In contrast to natural languages, programming languages are unambiguous, employ little redundancy, are meant to be interpreted literally, and consist of strictly structured text. To exploit these features of the problem, we perform the summarization at the code block level, leveraging the fact that source code is syntactically unambiguous.

There is some existing work on the use of **code folding** (also known as code elision) to aid comprehension. In particular, Cockburn et al. [28] find that illegible elision of all method bodies in a class improves programmer efficiency in editing and browsing tasks. Rugaber et al. [7] consider a conceptual model for manual folding, extending it to non-contiguous regions of code. Kullbach et al. [6] develop the `GUPRO` IDE to aid in the comprehension of C preprocessor code via rule-based folding of macro expansions and file includes. Also, Hendrix et al. develop the `GRASP` program comprehension tool, combining control structure diagramming with manual folding [5]. Bragdon et al. [8] perform code autofolding of long methods based on code block depth in their proposed Code Bubbles IDE. However, they do not evaluate the effectiveness of the autofolding method on its own, but rather as part of a larger UI. By contrast, we are the first to quantitatively study and evaluate the autofolding problem directly.

The task of **natural language summarization** has been studied extensively [29], mostly focusing on *extractive* summarization — the problem of extracting the most relevant text segments from documents. Source code identifiers (e.g., variable names) are information-rich and have been shown to be important for tasks such as feature location [30], [31]. NLP techniques have been used on these identifiers for information retrieval tasks such as automatically selecting labels for software artifacts [32]. Extractive summarization has also been applied for the automatic summarization of bug reports [33], [34].

We are aware of only a few previous methods that consider the problem of **code summarization**. One of the first approaches is program slicing [10], [11] which hides irrelevant LOC for a chosen program path – essentially a very specific form of query-based summarization. Program slicing focuses on the display of a path for a specific statement or variable of interest, and is not obviously applicable to the first look problem that we consider. Most similar to our work are Haiduc et al. [14], [15] and the follow up work by Eddy et al. [16] and Rodeghero et al. [35], who also consider the problem of summarizing source code, particularly methods and classes, but in their work code fragments are summarized by a short list of keywords. For example, the `equals` method in Figure 1 might be summarized by the list of terms (`equals`, `code`, `reason`, `Status`). McBurney et al. [17] take this idea further and present the keywords in a navigable tree structure, with more general topics near the top of the tree. In our work, we summarize code *with* code, which we would argue has the potential to provide a much richer and more informative summary.

Also, Ying et al. [18] consider the problem of summarizing a list of code fragments, such as those returned by a code search engine. They use a supervised learning approach at the level of lines of code. Because they consider the results of code search, their classifier uses query-level features, e.g., whether a line of code uses any identifiers that were present in the query. This is a source of information that is not available in our problem setting. In contrast, we target use cases in which the developer is skimming the source code to get an overview of its operation, rather than performing a directed keyword search. Kim et al. [36] develop a system that augments API documentation with code example summaries but these are mined from the web and are therefore limited to APIs which have examples already written for them — our approach is applicable to *any* source file.

On a more technical level, our folding-based summaries are distinguished from this previous work in that our summaries are coherent with respect to the programming language's syntax. Indeed, Eddy et al. [16] observe that developers prefer summaries with a natural structure. Folding on code blocks also enables us to retain method headers in the summary — identified by Haiduc et al. [15] as highly relevant to developers and accounting for the high scores of their best performing method. Additionally, our method leverages a multiple-project corpus during the summarization process, which we exploit to identify tokens which are less characteristic of a particular file.

In addition to extractive summarization methods, **abstractive summarization** techniques have also been used in software engineering research. Work in this area includes synthesis of API usage examples [37], extraction of API usage patterns [38], [39], and generation of natural language summaries for source code [12], [13].

The use of **topic models for source code** has also been studied in depth [32], [40]–[45]. Marcus et al. [46], [47] used Latent Semantic Indexing (LSI) [48] for identifying traceability links and concept location in source code. More closely related to our work, Haiduc et al. [15] used LSI as a content model for their keyword-based source code summarizer. In their follow up paper, Eddy et al. additionally used a hierarchical pachinko allocation model (hPAM) [49], a family of generative topic models that build on Latent Dirichlet Allocation (LDA) [50] with a hierarchical topic structure. McBurney et al. [17] used the hierarchical document topic model (HDTM) [51] for their structured keyword-based summaries. Note that HDTM is not the same as the topic model we propose: our model discerns file-specific tokens, leveraging the hierarchical structure present in the code, whereas HDTM infers a tree that represents similarities between methods. Also, Movshovitz et al. [27] successfully used LDA and link-LDA [52] for predicting class Javadoc comments from source file

text.

We cast autofolding as an instance of the general problem of **selecting an optimal subtree** given a certain budget. This problem has been studied theoretically by Guha et al. [53], who propose a dynamic programming solution, but this is only pseudo-polynomial time, and so is unlikely to scale well in practice.

III. Problem Formulation

Our aim in this paper is to summarize source code so that it conveys the most important aspects of its intended function. We envisage our proposed Tree-based Autofolding Software Summarization ALgorithm (TASSAL) being embedded in a programming language IDE and providing real-time summaries to the user of selected files. The summarization could be useful at multiple levels, ranging from a single source file to an entire corpus. For the purposes of this paper, we will focus on the Java programming language as it is a popular, high-level, platform-independent language. However, since TASSAL works entirely with the source code's Abstract Syntax Tree (AST), it can be applied to any programming language for which an AST can be defined.

The target use case we envisage for TASSAL is that of a developer not familiar with a project wishing to obtain an overview of a given file. For example, a developer who is considering using a new project on GitHub might like to get an overview of the algorithms used in each file of the project. We call this the *first-look problem*. The first-look problem is in contrast to tasks such as debugging and code reviewing for which a more focused summary may be desirable; we leave these other tasks to future work.

The outline of TASSAL is as follows: TASSAL takes as input a set of source files along with a desired compression ratio (i.e. level of summarization) and outputs a summary of each file where uninformative regions of code have been folded (see Figure 2 for an example). In order to achieve this TASSAL first selects the AST locations to obtain suitable regions to fold (Section III-A). It then applies a source code language model to each foldable region. The aim of this model is to identify, for every source file, which tokens specifically characterize the file, as opposed to project-specific or Java-generic tokens that are not as informative for understanding the file. To this end, we develop a scoped topic model for source code (Section III-B), which we apply to rank how informative each code region is to its enclosing file. Using this ranking TASSAL then leverages an optimization algorithm to determine the most uninformative regions to fold while achieving the desired level of compression. This is a novel optimization procedure that takes the structure of the code into account (Section III-C).

A. Problem Definition

Most modern IDEs already have extensive support for folding *specific* code regions as well as the ability to fold
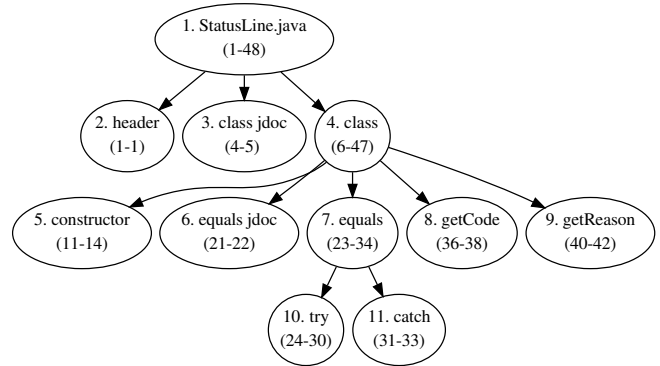

Figure 3: Partial foldable tree constructed for `StatusLine.java` (Figure 1). Numbered breadth-first with labels denoting block types and line numbers in brackets, cf. the source code snippet in Figure 1. Note that we have omitted some nodes for clarity.

regions based on user-specfied *rules*. IDEs with support for automatically folding regions based on their *location* have also been proposed [8]. But to the best of our knowledge the problem of automatically determining which regions to fold based on their *content* is novel. When we say that we *fold* a source code region we mean that the region is replaced by a one line summary and a symbol indicating that the region was folded. We define the *autofolding problem* as that of choosing a set of code regions to fold, such that the total length of the folded file as a fraction of the original is below a user-specified compression ratio, and the remaining, unfolded, text captures the most important aspects of the file in question. Autofolding can be seen as a special case of *extractive summarization*.

To encourage intuitive summaries, we let the system perform folding only on *code blocks* (regions of source code delimited by { , }), *comment blocks* (regions delimited by `/*(*)` , `*/`), and `import` statements. We call these the *foldable regions* of the code. Our reasoning for this is that it is a summary many programmers are familiar with as these are the regions that can be manually folded in the majority of modern IDEs and text editors. Moreover, code blocks are natural units for extractive summarization since they take advantage of the code structure specified by the programmer. However, since our approach works within the code's AST, it can be trivially extended to fold *any* contiguous region of interest. For example, in our implementation we have added optional features to allow autofolding of *line comments*, *fields* and *method signatures*. Also, it would be a trivial extension to allow statements, or a carefully designed subset thereof, to be folded. In keeping with the manual folding conventions in IDEs, the one line summary we display for a folded region consists of the first non-empty line of the code block, then an ellipsis, and finally the right delimiter of the region (see Figure 2).

We formalize the autofolding problem by using the AST representation of the source code [54]. Given a program's AST, we define the program's *foldable nodes* as those AST nodes which correspond to a foldable region of code. By starting at the root of an AST and sequentially extracting

all foldable nodes, we construct a directed *foldable tree*, containing just the AST nodes we are interested in. Figure 3 shows a partial foldable tree for the running example.

Constructing a foldable tree enables us to formulate the summarization problem mathematically as finding the best contiguous rooted subtree that takes up no more than a predefined number of lines of source code (LOC). That is to say, we unfold all nodes in the best subtree and fold the remaining nodes in the tree. Note that we require the tree to be rooted and contiguous as otherwise this would lead to confusing situations where we would have a deep node present in the summary with no context. We will describe the precise formulation in the next two sections.

B. Content Model

In order to determine which nodes of the foldable tree should be unfolded, we require a content selection method for choosing the best nodes to retain in the summary. Intuitively, one would like to retain the most informative nodes and a natural approach, as in text summarization, is to tokenize the node text and select the nodes with the most representative tokens. For this reason we make use of a *topic model*, an extension of the TopicSum model [9]. The high-level idea is to extend a standard topic model, latent Dirichlet allocation, with topics that are specific to particular projects, files and methods, thus allowing the model to identify which tokens are predominantly used in a specific file or project. Other types of probabilistic models that have been used for code are not well suited for this task. For example, *n-gram language models* can learn only local patterns of nearby tokens, whereas we require a global model of the distribution of code across files. An n-gram model has no way to identify whether the usages of a token are concentrated within a single file.

Tokenization A first idea would be to have one token in the topic model for each Java token in the code. However, some tokens (e.g. operators and delimiters) are not informative about the program content and identifier names have substructure that we wish to exploit. For example, in Figure 1, the `getCode` method name is closely related to the `code` member variable, which becomes apparent to the topic model only if it is split into two tokens `get` and `code`. For these reasons, we preprocess the Java tokens before incorporating them into the topic model. Given a code block, we first tokenize it into a set of Java tokens using standard tools for the Java programming language. Then we remove all tokens except for identifiers, i.e., programmer assigned names of variables, methods, classes, etc. Finally, we convert each of the identifiers into a new set of tokens by splitting on camel case and underscores, and converting to lowercase (e.g., `FooBarBaz` becomes three tokens `foo`, `bar`, and `baz`, as would `foo_bar_baz`). Additionally, we include the text of all comment blocks in the topic model, splitting the comment text based on words, again applying the identifier splitting procedure on any comment tokens that contain camel case or underscores. Let the

vocabulary $\mathcal{V} = \{t_1, \ldots t_T\}$ be the set of all unique tokens that occur in the corpus. We use the term *lexical item* $t \in \mathcal{V}$ to refer to elements in the vocabulary (which can occur multiple times in the corpus as different tokens).

We do not use a stoplist because we expect that the set of appropriate stop words for program text would be different to those for natural language text. Instead, our topic model identifies background words automatically.

Vector Space Model (VSM) The VSM is a standard method in information retrieval [55], in which documents are represented by continuous-valued vectors, and similarity is measured by metrics such as the cosine similarity between such vectors. To apply this idea to summarization, we compare a vector representing a source file with a vector representing a summary, and find the closest match. More specifically, for a file f, let $\mathbf{v}_f$ be the log term frequency (tf) vector; this is vector containing the log frequency of each token in the file. Similarly, for a potential summary $\mathbf{u}$ we can define a log-tf vector $\mathbf{v_u}$. Then if we have a set of potential summaries of the file f, we can choose the summary that maximizes

$$\sigma(\mathbf{u}) = \text{csim}(\mathbf{v}_f, \mathbf{v_u}), \tag{1}$$

where csim denotes the cosine similarity. One potential disadvantage of this method is that $\mathbf{v}_f$ includes many generic tokens that are used throughout the project or multiple projects. Therefore we next turn to methods that specifically identify such generic tokens.

Topic Model Now we describe an approach to identifying file specific tokens based on a topic model. Because topic models are less familiar to a software engineering audience, we will explain the model in some detail; although the mathematics of this section may seem complex to some readers, in fact, the model that we employ is a straightforward extension to the basic latent Dirichlet allocation (LDA) model [50], [56]. A topic model [57] is a type of statistical model over documents, that represents each document as a combination of *topics*, which are groups of words that tend to occur together. Formally, each topic is modelled as a probability distribution ϕ_k over lexical items and can be viewed as a vector of length T where each entry ϕ_{kt} is a probability. A document is modelled as a probability distribution θ_d over K topics; this can again be represented as a vector where each element θ_{dk} is a probability that represents how important topic ϕ_k is to document d. Given a document d with tokens $\mathbf{w}_d = (w_1^{(d)} \ldots w_{N_d}^{(d)})$, we seek to infer topic assignments $z_i^{(d)}$ for each token $w_i^{(d)}$. Each topic assignment $z_i^{(d)} \in \{1 \ldots K\}$ is an indicator variable that specifies which topic was responsible for generating the token $w_i^{(d)}$. We use $\mathbf{z}_d = (z_1^{(d)} \ldots z_{N_d}^{(d)})$ to denote a vector of topic assignments to all the tokens in document d.

Topic modelling typically follows the paradigm of Bayesian statistics. One first defines a probability distribution $P(\phi_1 \ldots \phi_K, \theta_1 \ldots \theta_D, \mathbf{z}_1 \ldots \mathbf{z}_D, \mathbf{w}_1 \ldots \mathbf{w}_D)$ that de-

scribes how the topics and documents would be distributed if all modelling assumptions were correct. Often the easiest way to describe such a model is to consider an algorithm that samples from it. In the case of a topic model, this will be an algorithm that samples documents. Then, when we receive a corpus of documents $\mathbf{w}_1 \ldots \mathbf{w}_D$, we infer topics for that corpus by computing the posterior distribution $P(\phi_1 \ldots \phi_K, \theta_1 \ldots \theta_D, \mathbf{z}_1 \ldots \mathbf{z}_D | \mathbf{w}_1 \ldots \mathbf{w}_D)$. The posterior distribution is the conditional distribution over the quantities we don't know, given the ones that we do; it is uniquely determined by the joint probabilistic model and the laws of probability. Given samples from the posterior distribution, we estimate each topic ϕ_k by averaging over the samples.

TopicSum [9] is a *scoped* topic model that extends LDA to handle topics at multiple levels. In TopicSum, each topic can be one of three kinds: a) a probability distribution over words that is local to a single document, b) a distribution over words that is local to a subcollection of related documents (such as all articles from the New York Times), c) a distribution over background words that is available to all documents in the corpus. The intention is that the background topic models stop words, the document subcollection topic represents significant content and the document topic very specific document words.

We adapt TopicSum to source code by defining a set of scopes that are appropriate for program text. Specifically, in our model we consider five different kinds of topic: one topic ϕ_f for each file f, one topic ϕ_p for each software project p and three background topics shared across projects. Although the model isn't aware of this, in practice we find that the three background topics correspond to common Java tokens (ϕ_{B_J}), common Javadoc comment tokens (ϕ_{B_D}) and common header comment tokens (ϕ_{B_H}). This model fits well into our summarization procedure because it separates out which tokens are characteristic of general Java code, of a specific project and of a specific file, so that when we generate the summary we can focus on preserving the file-specific tokens rather than the generic Java tokens.

We describe the model by providing a procedure to sample from it. In our model, a file f in project p is generated as follows: each token $w_i^{(nfp)}$ in every foldable node n of the file is chosen from a specific topic $\phi_{z_i^{(nfp)}}$, where the topic assignment $z_i^{(nfp)}$ is selected according to the distribution over topics θ_n in the node n. That is, we have the following generative procedure:

1) Choose token distributions $\phi_k \sim \text{Dirichlet}(T, \beta_k)$ for topics $k \in \{B_J, B_D, B_H, p, f\}$.
2) Choose topic distribution $\theta_n \sim \text{Dirichlet}(K, \alpha\mathbf{m})$ for node n.
3) For each token $w_i^{(nfp)}$ in node n:
   a) Choose a topic $z_i^{(nfp)} \sim \text{Categorical}(K, \theta_n)$.
   b) Choose a token $w_i^{(nfp)} \sim \text{Categorical}(T, \phi_{z_i^{(nfp)}})$.

Here $\alpha\mathbf{m}$ and β_k denote hyperparameters for the prior distributions of topics and tokens, i.e. the initial topic and
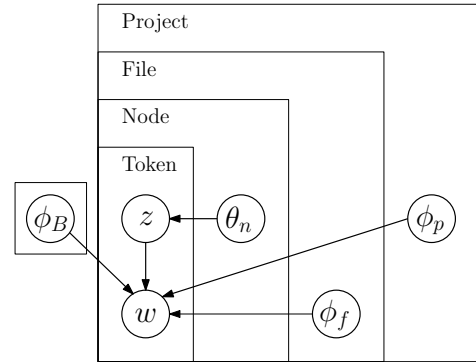

Figure 4: Graphical model depiction of the TASSAL content model using plate notation (we refer readers unfamiliar with such notation to the tutorial [58]). The plates denote repeated groups of variables.

token assignments. Readers familiar with topic models will recognize this as a simple scoped extension of LDA [56]. Figure 4 is a graphical illustration of our model using plate notation, we refer readers unfamiliar with such notation to the tutorial [58].

To estimate the topics ϕ_k, we need to compute the posterior distribution over the topic assignments z. Unfortunately, as in most topic models, this distribution is intractable, so we use a popular approximation called *collapsed Gibbs sampling*. Collapsed Gibbs sampling is a stochastic iterative procedure, which does not make explicit reference to the parameters θ_d and ϕ_k, which at each iteration returns samples of topic assignments $z_i^{(nfp)}$ for each token $w_i^{(nfp)}$ in the corpus. It can be shown that such successive samples approximate the posterior *marginal* distribution over topic assignments [56], marginalizing out θ_d and ϕ_k. We omit the details of the algorithm for space, but it is a simple extension of the one used for LDA [56]. For the purposes of this paper, the sampler can be thought of simply as a black box that outputs a topic assignment $z_i^{(nfp)}$ for each token in the corpus.

Once we have samples of $z_i^{(nfp)}$, it is easy to compute an estimate of each topic distribution ϕ_k, which is what we will need for summarization. These can be estimated using the maximum a posteriori (MAP) estimate [56]

$$\hat{\phi}_{kt} = \frac{N_{t|k} + \beta_k}{\sum_{t=1}^{T} \left(N_{t|k} + \beta_k\right)}, \qquad (2)$$

where $N_{t|k}$ denotes the number of times the lexical item t has been assigned to topic k in the entire set of samples $\{z_i^{(nfp)} | \forall i, n, f, p\}$. In other words, we simply compute the proportion of times that topic k is used to generate each lexical item $t \in \mathcal{V}$.

Additionally, unlike the original TopicSum model which used fixed hyperparameters $\alpha\mathbf{m}, \boldsymbol{\beta}$, we incorporate efficient hyperparameter optimization, namely MacKay and Peto's method [59] with histogram based computation of the necessary counts (see [60] for details).

C. Optimization Method

Given a content model, we now need an algorithm that extracts the most relevant summary by selecting suitable AST nodes to unfold in the foldable tree. We propose an iterative greedy optimization algorithm to extract the most relevant rooted contiguous subtree from the foldable tree given constraints on the subtree size. At each iteration of the greedy procedure, we have a candidate summary, i.e., a subtree we have gathered so far and decided to include in the summary. We then consider each additional node of the foldable tree to decide whether it is relevant and should also be included.

We need to compare a candidate summary to the original file f. For a VSM, cosine similarity is a standard measure. For topic models, a suitable information-theoretic measure for this similarity is the Kullback-Leibler (KL) divergence, which measures the difference between two probability distributions. To convert the candidate summary into a probability distribution, we divide the number of times each token w occurs in the summary over the total number of tokens in the summary, i.e. by using the *empirical unigram distribution*. Using our topic model, we can estimate the file token distribution $\hat{\phi}_f$ via (2). We can therefore assign the node under consideration a score based on the KL divergence between the corresponding file token distribution $\hat{\phi}_f$ and the empirical unigram distribution of the candidate summary. Intuitively, we want our summary to contain tokens that characterize the file rather than tokens that are common elsewhere in the corpus.

Formally, for each node i let $u_i \in \{0, 1\}$ indicate whether it is unfolded, with 1 corresponding to true (i.e. unfolded) and let $\mathbf{u}$ be the vector containing all the u_i (so that $\mathbf{u}$ contains u_i at position i). We define a score $\sigma(\mathbf{u})$ for a candidate summary $\mathbf{u}$. For the VSM, we use $\sigma(\mathbf{u})$ as (1) whereas for the topic model we use the score

$$\sigma(\mathbf{u}) = -KL(\hat{\phi}_f | P_{\mathbf{u}}) \qquad (3)$$

where $\hat{\phi}_f$ denotes the file token distribution, $P_{\mathbf{u}}$ the empirical unigram distribution over tokens in the candidate summary and $KL(\cdot|\cdot)$ the KL divergence. Foldable regions with no tokens are assigned a score of $-\infty$ for consistency, which means that they are never unfolded.

We can now formulate the autofolding summarization problem as finding the optimal rooted contiguous subtree. Requiring the subtree to be rooted and contiguous ensures that all nodes in the summary are presented within their syntactic context. Suppose we wish to summarize a file using a line-level compression ratio of $p\%$, i.e., we would like to compress the file to $p\%$ of its original size in LOC. We can then define L_{max}, the maximum number of lines of code that are allowed in the summary, as $L_{max} := (p/100)L_0$ where L_0 denotes the number of LOC in the original file. Note that since we are folding on a block level, a line-level compression ratio of $p\%$ does not mean that $p\%$ of the blocks are compressed, in fact for small files all the blocks are often folded at 50% compression.

Moreover, folding on a block level also means that in practice the specified line-level compression ratio will never be achieved exactly, but instead the returned summary will always be slightly shorter. In certain situations, it may be desirable to allow the specified compression ratio to be slightly exceeded, such as when a slightly larger summary would have many more relevant terms. In our approach, we do not handle this tradeoff in the optimization method and the target compression ratio is taken as a hard maximum. It would be interesting to modify the user interface so as to indicate when a slight increase in compression ratio would lead to large increase in the estimated quality of the summary, for example, by visualizing the score $\sigma(\mathbf{u})$ alongside a slider that controls the compression ratio, but we leave this to future work.

Let $G = (V, E)$ be the foldable tree, that is, a directed tree with a set of AST nodes $V = \{1, \dots, N\}$ consisting of the aforementioned foldable regions and E a set of *directed* edges between the nodes, where it is understood that $(i, j) \in E$ means that i is the parent node of j (cf. Figure 3). Furthermore, for a node $i \in V$ let L_i denote the number of LOC underneath node i. We then define C_i, the cost of unfolding node i as the number of LOC *unique* to node i, i.e., underneath node i but not any of its children. Formally,

$$C_i := (L_i - 1) - \sum_{j:(i,j)\in E} (L_j - 1). \qquad (4)$$

The first line of a node is never folded (cf. Figure 2), hence the minus one. Let $\sigma(\mathbf{z})$ denote the score obtained from the summary nodes $u_1, \dots, u_N$ as defined in (1) or (3). The optimal rooted contiguous subtree problem is

$$\max_{\mathbf{u}} \sigma(\mathbf{u}) \qquad (5a)$$

$$\text{s.t. } \sum_{i=1}^{N} C_i u_i \leq L_{max} \qquad (5b)$$

$$u_i \geq u_j \qquad \text{if } (i, j) \in E \qquad (5c)$$

$$u_i \in \{0, 1\} \qquad \forall\, i \in V. \qquad (5d)$$

That is, we unfold the nodes which maximise the total score (5a) subject to staying below the maximum allowed LOC (5b) and retaining a rooted contiguous subtree (5c).

As the score (3) is nonlinear we approximate this problem using a greedy approximation algorithm: that is we unfold the next available node that will give the maximum score (3) per cost (4) increase, honoring the cost constraint (5b) and unfolding any folded parent nodes (5c). That is, starting from all the nodes being folded (i.e. $\mathbf{u} = \mathbf{0}$), iteratively choose the node that maximizes

$$\sigma(\mathbf{u} + \mathbf{e_i})/C_i$$

while $\sum_{i=1}^{N} C_i u_i \leq L_{max}$. Here $\mathbf{e_i}$ denotes the i-th unit vector, which is 1 at position i and 0 everywhere else.

Project	Description	LOC	Methods	Classes	Forks	Watchers
storm	Distributed Computation System	59,827	5,740	761	1,416	7,471
elasticsearch	REST search engine	518,905	32,077	4,990	1,283	5,246
spring-framework	Application Framework	798,249	47,214	8,395	1,774	2,568
libgdx	Game Development Framework	334,706	33,821	2,651	1,844	2,243
bigbluebutton	Web Conferencing System	105,315	6,364	852	1,602	969
netty	Network Application Framework	160,579	10,324	1,267	927	2,304

Table I: The top Java projects on GitHub, used in the current work. Ordered by popularity.

IV. Experimental Setup

In this section, we describe how we obtain a gold standard summary on real code. The gold standard allows automatic evaluation, which is the de facto standard in NLP summarization [61]. Automatic evaluation is a standard technique in modern artificial intelligence research. It is vital because it leads to a rapid development cycle in which we can compare many more differing algorithms than would be possible with user studies alone. In particular, it enables us to perform a comprehensive evaluation of TASSAL in Section V.

Data To evaluate the performance of TASSAL we obtained the source code for the top six Java projects on the popular GitHub service for hosting open source code repositories. The top six projects were determined by a popularity score, which is the sum of the number of forks and the number of watchers, where each is separately normalized to have zero mean and unit variance. We selected the projects with the highest score that were greater than 100,000 KB in size as of December 1st 2013. These are given in Table I along with a brief description of their domains.

For each of these six projects, we divided the project files into quartiles by file size, and sampled four files from each quartile, to obtain a total of 96 files (12, 347 LOC), each one of which was annotated by two human annotators. Only `.java` files were considered (excluding the `package-info.java` files, which only contain package-level documentation). Figure 5 shows the distribution of file sizes across the top projects.

Annotation Procedure Human annotators were given guidelines prior to performing the annotation. Annotators were asked, for each source code file, to manually fold the file in Eclipse until they reached a compression ratio of 50% and save their work. The annotators were asked to fold regions that would be least useful to a developer who was reading the file for the first time in order to understand its overall purpose, reflecting our interest in the first-look problem. A compression ratio of 50% was chosen as, on average, it gave the best balance between providing a reasonable summary whilst not eliding all the interesting details of the underlying source code. Although 50% may not seem like a dramatic compression, in fact many of the remaining lines are block headers or blank (cf. the running example in Figure 2). We found that on

average across all 96 files in the annotated dataset, half of the LOC remaining unfolded in the file were blank or block headers. Moreover, when we ran TASSAL on the dataset, we found that 22% of files had their top-level class entirely folded due to the fact that unfolding *any* nodes in the top level class would have resulted in exceeding the 50% compression ratio.

Annotators were allowed to browse the full source code of the project while annotating each file. We used two experienced Java developers as annotators, who each independently annotated the entire data set. We performed our annotation prior to the development of our summarization system so it was impossible for the annotators to unconsciously favour the system's output in their judgements.

Annotators were asked to fold regions until they thought the file conveyed the most important aspects of its function (or equivalently provided a good overview of its purpose to a programmer unfamiliar with it) with the following constraints: Annotators were asked to always fold import statements and header comments (such as copyright notices). Empty and one-line blocks were also folded by default. Setters and getters along with other I/O methods were asked to be folded, unless they contained core logic of the code. Similarly, annotators were asked to fold commonly overridden and overloaded methods (e.g. `toString()` methods) unless they provide sufficient new information about the functionality of the code. Finally, Javadoc and block comments were left unfolded if they were informative and succinctly explained the function of the associated class or method. Comments whose text spanned only one line were folded by default.

As for potential threats to validity, the first concern that could be raised is whether it is indeed possible to expect different annotators who are generating a reference summary to have similar ideas about what constitutes a good summary. To address this concern, we report in the next section the inter-annotator agreement at the line level, which reflects substantial agreement among the annotators. Second, one could raise the concern that different types of developers, or developers who are considering different tasks, would require different types of summaries. We would agree with this concern: it is certainly true that developers who are familiar with the projects could favor a different type of summary than the annotators. However, recall that in this study, our target use case focuses on the first-look problem, i.e. developers who are new to the
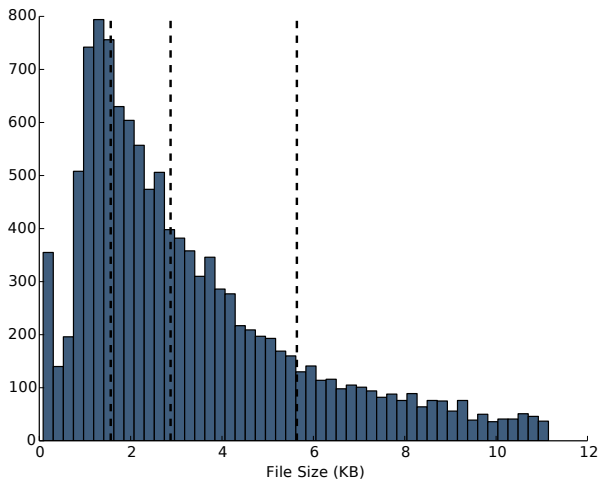
Figure 5: Histogram of file sizes across the top projects (dashed vertical lines denote quartiles).

Figure 6: Line-level inter-annotator agreement for the 96 source files at a compression ratio of 50%.

projects, for which we would argue our annotators are good representatives. Finally, note that annotators were asked to fold regions whereas TASSAL unfolds regions, however this is merely for mathematical convenience and the two formulations are entirely equivalent.

Annotation Statistics As autofolding is a new task, we need to verify that the task is well-defined. We therefore calculated the line-level agreement between annotators and found that it was substantial, with a Fleiss' Kappa value of 0.71 at 50% compression, averaged across all files. Figure 6 shows the line-level inter-annotator agreement for each of the 96 files at a compression ratio of 50% and one can see that for the majority of files we obtain substantial agreement between the two annotators. We also found that the line-level annotation agreement for non-trivial nodes (i.e. nodes that are not *block comments* or *import statements*) was moderate with a Fleiss' Kappa of 0.60.

V. Results

In this section, we evaluate the performance of TAS-SAL against our annotated test set from Section IV. We begin by training and assessing the quality of TASSAL's underlying topic model before moving on to a comprehensive evaluation of TASSAL itself. As our method is fully unsupervised, we use all 96 annotated files from the six projects in our corpus for evaluation.

Topic Model We train TASSAL's topic model on all 12,093 `.java` files from the six projects in our corpus. This enables the topic model to automatically discern common coding patterns, such as common libraries, and thus allows TASSAL to recognize unimportant code. We run the topic sampling algorithm for 5,000 iterations, performing hyperparameter optimization every 10 iterations, to infer our trained model.

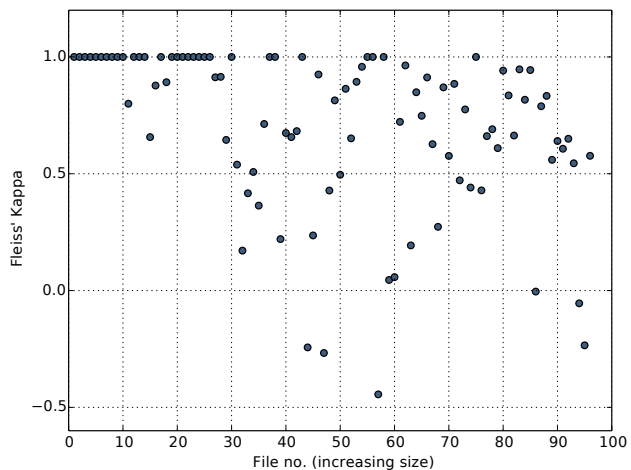To demonstrate that the topic model learns to distinguish between project-specific and file-specific tokens, we show the top ten tokens in each of the five topic types (Java background, Javadoc background comment, header background comment, project and file) for two projects and two files in Table II. Note that background tokens which are very common in a file/project can also appear among the top ten file/project tokens. One can see that these are representative of their respective topics: the Java background topic contains many commonly used terms in Java such as `get`, `set`, `map` and `object`. The background comment topics on the other hand contain common English stop words such as `the`, `a`, `or` and one can clearly see the distinction between the topics, i.e., words that commonly appear in headers (such as `license`, `distributed`) are found in the header background comment topic whereas those that commonly appear in Javadoc comments (`link`, `code`) are found in the Javadoc background comment topic. Looking at the topics for the `spring-framework` project one can see that it contains the fully resolved project name `org.springframework` as well as common project-specific tokens such as `bean`, `context` and `factory`. Similarly, the `bigbluebutton` project contains the specific tokens `sip`, `event`, `message` and `gnu`. Looking at selected files from both projects, the `DataSourceUtils` file topic contains tokens specific to the function of that file, e.g. `connection`, `data`, `synchronization` and `isolation`. Finally, the `QuaLsp` file topic, a codec implementation, contains the very function-specific tokens `lsp` and `ld8`.

This illustrates the quality of our proposed topic model and suggests that it can distinguish file- and project-specific tokens from those that are common across all Java projects. It also raises the exciting possibility that our topic model is robust enough for wider applications and not merely restricted to summarization.

Baselines To provide a comprehensive performance comparison of TASSAL, we also evaluate three alternative baseline systems that represent more naïve approaches for summarizing source code. All baselines start from a fully folded tree and gradually unfold nodes, making local

Background			Project		File	
Java	header comments	Javadoc comments	spring-framework	bigbluebutton	DataSourceUtils	QuaLsp
get	the	the	bean	sip	connection	lsp
string	license	a	org	org	con	j
value	or	to	test	log	holder	constants
name	under	of	context	it	source	k
type	you	link	springframework	event	data	ld8
object	distributed	for	exception	gnu	synchronization	mode
i	of	is	request	listener	isolation	tmp
set	0	this	factory	message	order	wegt
exception	is	and	get	public	level	index
map	2	code	class	general	close	m

Table II: The top ten tokens in each topic-type as found by our topic model.

decisions, until they reach the required compression ratio. If a node is to be unfolded, all of its parent nodes are also unfolded as well. The baselines are:

Shallowest unfold the shallowest available node first, choosing randomly if there is more than one. This would unfold node 1 in Figure 3 first, followed by either node 2, 3 or 4, etc.

Largest unfold the largest available node first, as measured by the number of tokens, breaking ties randomly. In Figure 3, this would unfold the class block (node 4) first, followed by the class Javadoc (node 3), etc.

Javadoc first unfold all Javadoc comments (in random order) and then fallback at random to an available node, unfolding methods last. This would unfold the Javadoc nodes 3, 6 in Figure 3 first and the method nodes 5, 8, 9 last (7 would already be unfolded as it is the parent of 10 and 11).

Each of the baselines represents a possible assumption that we can make about summarizing source code. The Largest baseline assumes that the largest nodes are more valuable in a summary, the Shallowest baseline is representative of the folding approach used in the Code Bubbles IDE [8] and the Javadoc baseline is representative of the current defaults of IDEs such as Eclipse. While we are aware that these baselines are rather simple methods, this reflects the fact that the autofolding problem has not received much attention in the research literature — to our knowledge, there simply do not exist more advanced methods in the literature for us to compare against.

To further verify our annotation procedure, we implemented the basic guidelines followed by the annotators as an additional system:

Guidelines unfold getters/setters, I/O methods (read|write|load|save.*), block comments, import statements and empty blocks last in that order, unfolding other node types first in random order. In Figure 3, this would unfold nodes 1, 3–7, 10, 11 first in random order followed by the getter nodes 8, 9 and the header node 2. This allows us to get some good insight into how much additional content-sensitive folding was done by the annotators after these basic annotation guidelines were followed: we found that only 27% of all nodes were forced

to be folded by the guidelines, meaning that 73% of all nodes that were folded in the gold standard require human judgement.

Performance To assess the performance of TASSAL and the baselines, we used the annotated test set provided by each of the two annotators as our gold standard in turn, averaging the results across both annotators. In this way we can measure whether the output of the summarizer matches human judgements.

For each file in the test set, we treated the folding problem as a binary classification, classifying each line of code in a foldable region as either unfolded (positive) or folded (negative), excluding one-line foldable regions (which are indistinguishable since all foldable regions are folded to one line). This enabled us to calculate the average accuracy, precision and recall of our summarizer across all test files and also F_1 as the harmonic mean of the average precision and recall.

We compared both TASSAL with the topic model and TASSAL with VSM (which we denote as TASSAL VSM). The resulting average performance metrics at a compression ratio of 50% are given in Table III. As one can clearly see from the results, TASSAL (with topic model) outperformed all the systems, by a margin of about 10% when compared against the best performing baseline (Javadoc). TASSAL VSM on the other hand performed poorly, losing to the Javadoc baseline. We also performed the same analysis on each foldable region of code, i.e. classifying each foldable region as either unfolded or folded. The resulting average performance metrics are given in Table IV and one can see that TASSAL is once again the best performing system.

We are also interested in the performance of TASSAL on *non-trivial* foldable regions, i.e. foldable regions that are not Java *block comments* or *import statements*. Note that *Javadoc comments* are considered non-trivial as these often contain usage and implementation details. We therefore performed the same analysis as above on non-trivial regions at both a line- and node-level. The results are given in Tables V,VI and one can see that TASSAL remains the best performing system. One can clearly see

	Accuracy	F1	Precision	Recall
TASSAL	**0.77**	**0.75**	**0.74**	**0.76**
TASSAL VSM*	0.65	0.61	0.61	0.60
Javadocs*	0.68	0.65	0.64	0.66
Shallowest*	0.65	0.62	0.60	0.63
Largest*	0.60	0.56	0.56	0.57
Guidelines	0.74	0.71	0.71	0.72

Table III: *Per-line* evaluation statistics for the summarizers evaluated on *all nodes* at a compression ratio of 50%. Averaged across annotators and all test files, ordered best first. *significantly different from TASSAL ($p < 0.05$).

	Accuracy	F1	Precision	Recall
TASSAL	**0.62**	**0.66**	**0.57**	**0.77**
TASSAL VSM*	0.54	0.55	0.50	0.60
Javadocs*	0.54	0.57	0.50	0.66
Shallowest*	0.56	0.57	0.52	0.63
Largest*	0.54	0.53	0.50	0.57
Guidelines	0.59	0.62	0.55	0.72

Table V: *Per-line* evaluation statistics for the summarizers evaluated on *nontrivial nodes* at a compression ratio of 50%. Averaged across annotators and all test files, ordered best first. *significantly different from TASSAL ($p < 0.05$).

	Accuracy	F1	Precision	Recall
TASSAL	**0.69**	**0.55**	**0.53**	**0.56**
TASSAL VSM*	0.57	0.43	0.39	0.50
Javadocs	0.60	0.46	0.42	0.52
Shallowest	0.61	0.46	0.43	0.50
Largest	0.65	0.47	0.47	0.46
Guidelines	0.62	0.50	0.47	0.54

Table IV: *Per-node* evaluation statistics for the summarizers evaluated on *all nodes* at a compression ratio of 50%. Averaged across annotators and all test files, ordered best first. *significantly different from TASSAL ($p < 0.05$).

	Accuracy	F1	Precision	Recall
TASSAL	**0.71**	**0.59**	0.57	**0.61**
TASSAL VSM*	0.61	0.51	0.49	0.53
Javadocs	0.63	0.52	0.49	0.56
Shallowest	0.65	0.51	0.51	0.51
Largest	0.70	0.52	**0.60**	0.46
Guidelines	0.65	0.56	0.54	0.59

Table VI: *Per-node* evaluation statistics for the summarizers evaluated on *nontrivial nodes* at a compression ratio of 50%. Averaged across annotators and all test files, ordered best first. *significantly different from TASSAL ($p < 0.05$).

the advantage of using a topic model here, as TASSAL consistently outperforms TASSAL VSM which exhibits similar performance to the baselines.

Turning our attention to the Guidelines system, we find that it performs very well as expected (we are after all following and evaluating on the annotation guidelines), outperforming all the baselines and TASSAL VSM. This not only further validates our annotated dataset (demonstrating that our annotators did indeed follow the basic guidelines given to them), but also demonstrates that TASSAL is indeed able to match human judgements: 28% of the nodes folded by the Guidelines system at 50% compression are required to be folded by the annotation guidelines, meaning that 22% of the folded nodes require human judgement. As TASSAL clearly outperforms the Guidelines system in Tables III–VI, we can conclude that, despite being an unsupervised system, TASSAL is able, to some degree, to match the intuitive human judgements of the annotators. Conversely, this also shows that TASSAL VSM is either unable to match the basic guidelines or the human judgements or both, clearly demonstrating the need for a more sophisticated content model.

As for statistical significance, we calculated two-tailed p-values using `sigf` [62] for the F_1 scores on both gold standards. The difference between TASSAL and the baselines is significant at the line-level for all nodes and non-trivial nodes ($p < 0.05$). At the node-level, the difference between TASSAL and the baselines is not significant.

As a further test of whether the proposed summaries are plausible, we consider several classes of methods that are likely to be uninteresting, and verify that TASSAL usually folds them. We show the percentage of times header comments, imports, constructors, getters, setters

and other generally uninteresting pattern-based method types were folded at 50% compression in Table VII. TASSAL folds these methods in most cases. Furthermore, when such methods are included in the summary, these exceptional methods turn out to be qualitatively more interesting, as we verify by manual examination. We show example snippets in Figures 7–9. As one can see, the unfolded methods tend to exhibit unusual or non-standard behaviour in the method body, so much so that they cannot be easily summarized by their signature alone. This lends further evidence to the credibility and usefulness of our summarization approach.

It is also evident from the results that the Shallowest and Largest baselines perform poorly and it is not difficult to see why. The shallowest nodes tend to be top-level code blocks defining methods and classes, which rarely contain core logic directly. Rather, the core logic tends to be nested in children of top-level blocks, such as `if-else` statements, `for`/`while` loops and `try-catch` blocks. The Largest nodes may contain a substantial amount of code but rarely the core logic as the number of tokens is a bad indicator of code importance. To see this, consider header comments which contain many tokens (words) often stating the code copyright, or common class methods such as the equals method in Figure 1 which perform routine functions yet contain blocks with many identifiers. The Javadoc baseline, on the other hand, represents the de facto summarization method currently used in IDEs and therefore performs much better as one would expect.

We compared the performance of the summarization systems at a range of compression ratios (effectively treating it as a threshold) against the gold standard at the fixed compression ratio of 50%. The standard approach

header	import	construct	get	set	put	is	has	read	write	add	remove	contains	clear	reset
80%	70%	83%	83%	88%	100%	91%	100%	75%	80%	100%	80%	100%	100%	83%

Table VII: The percentage of times specific types of node in the test set (header comments, import statements, constructors and method names starting with the listed keywords) were *folded* at 50% compression by TASSAL.

	All Nodes		Nontriv. Nodes	
	Per-line	Per-node	Per-line	Per-node
TASSAL	**0.86**	**0.79**	**0.57**	**0.79**
TASSAL VSM	0.72	0.56	0.51	0.63
Javadocs	0.81	0.71	0.53	0.71
Shallowest	0.77	0.69	0.54	0.71
Largest	0.73	0.74	0.55	0.79
Guidelines	0.82	0.66	0.54	0.69

Table VIII: Area under the curve (AUC) for the receiver operating characteristic (ROC) curves of the summarization systems as the compression ratio is varied.

	Conciseness		Usefulness	
Summary	Mean	St. dev.	Mean	St. dev.
Gold	3.34	1.03	3.33	1.04
TASSAL	**3.27**	**1.01**	**3.18**	**0.97**
Javadocs*	3.07	1.03	2.69	1.09
Shallowest*	2.97	1.05	2.50	1.15
Largest*	3.08	1.07	2.67	1.06

Table IX: Mean and standard deviation averaged across developer ratings for summaries produced by the four summarization systems and a randomly chosen gold standard at a compression ratio of 50%. *significantly different from Gold and TASSAL ($p < 0.05$).

to evaluating a binary classifier with a threshold is via the area under the curve (AUC) values for its receiver operating characteristic (ROC) curve. An ROC curve plots the recall (equiv. true positive rate) against the fraction of false positives out of the negatives (false positive rate) at various thresholds (compression ratios). Essentially the higher above the diagonal a binary classifier is, the better it performs and a good measure for this is the AUC. Intuitively, the AUC measures the quality of the ranked list produced by the system, i.e., where all the lines/nodes are ranked by how likely the system is to fold them, whereas the F scores we used earlier measure only the performance at a single compression ratio. The AUC for each of the summarizers is given in Table VIII. Once again, we can see that TASSAL outperforms all the baselines with an AUC of 0.86 when evaluated per-line on all nodes in the gold standard.

Finally, it should be noted that TASSAL has a very fast runtime, needing less than five seconds of CPU time to summarize an average file in the dataset on a 2.66GHz Core 2 Quad machine (note that this excludes training the topic model which, in an implemented system, we assume is handled in an offline preprocessing step).

Developer Study We conducted a developer study to test whether content-based autofolding methods produces better summaries than non-content based methods. We asked developers to rate the summaries produced by our best content-based method, TASSAL using the topic model, and the summaries the three non-content baselines and one randomly chosen annotator at a compression ratio of 50%. We recruited six experienced developers for our study, separate from the annotators who created the gold standard. All were recently graduated computer science masters students with an average 5.3 years Java programming experience and 4 years industry programming experience.

To this end, we randomly selected four of the six projects and five of the annotated files from each project for the study, resulting in 20 files in total. For every file, developers were presented with each of the five possible summaries in random order and asked to rate the conciseness and usefulness of each summary on a five-point Likert scale (higher is better). Developers were allowed to browse the full source code of each project during the study.

We show the average ratings across all six developers in Table IX along with the average standard deviations. One can see that summaries produced by TASSAL score around 0.2 points higher on conciseness and 0.5 points higher on usefulness than the three baselines. Moreover, TASSAL is only 0.07 points lower on conciseness and 0.15 points lower on usefulness than the gold standard summary. We performed ANOVA on the developer conciseness and usefulness ratings for the different summaries and found that the difference between TASSAL and the baselines was significant ($p < 0.05$) as denoted in Table IX whereas the difference between TASSAL and the gold standard was indeterminate.

In a follow up questionnaire we asked the developers to summarize the measures they used to rate each summary. The developers generally favoured the same criteria that we identified when creating the annotation guidelines for the gold standard in Section IV. That is to say, the developers preferred not to see accessor and housekeeping methods but felt it was important to show method/class Javadoc comments (as reported in previous code summarization studies [15], [16]).

In short, the study results clearly show that our summarization system is not only preferred by developers over the baselines, it is almost as good as the summary produced by an expert annotator.

Some of the developers also offered interesting qualitative feedback about their experience with TASSAL, even though we did not explicitly ask them for this information. Developer 2 felt that in order to understand a project better, he would like to see "some kind of graph of class relationships" from which he could determine "which

```java
public SocksInitResponse(SocksAuthScheme authScheme) {
    super(SocksResponseType.INIT);
    if (authScheme == null)
        throw new NullPointerException("authScheme");
    this.authScheme = authScheme;
}

public ChannelGroupException(
    Collection<Map.Entry<Channel, Throwable>> causes) {
  if (causes == null)
    throw new NullPointerException("causes");
  if (causes.isEmpty())
    throw new IllegalArgumentException("causes must be
            non empty");
  failed = Collections.unmodifiableCollection(causes);
}
```

Figure 7: Snippets of constructors that were *unfolded* by TASSAL.

```java
public <T> T getOption(ChannelOption<T> option) {
  if (option == SO_TIMEOUT)
    return (T) Integer.valueOf(getSoTimeout());
  return super.getOption(option);
}

public void set(final ContactID c) {
  indexA = c.indexA;
  indexB = c.indexA;
  typeA = c.typeA;
  typeB = c.typeB;
}
```

Figure 8: Snippets of getters/setters that were *unfolded* by TASSAL.

```java
public void readFrom(StreamInput in) throws IOException {
  super.readFrom(in);
  if (in.getVersion().onOrBefore(Version.V_0_90_3))
    in.readBoolean(); // refresh flag
  full = in.readBoolean();
  force = in.readBoolean();
}

public static void writeVLong(DataOutput stream, long i)
  throws IOException {
    if (i >= -112 && i <= 127) {
        stream.writeByte((byte) i);
        return;
    }
    int len = -112;
    if (i < 0) {
        i ^= -1L; // take one's complement'
        len = -120;
    }
    long tmp = i;
    while (tmp != 0) {
        tmp = tmp >> 8;
        len--;
    }
    stream.writeByte((byte) len);
    len = (len < -120) ? -(len + 120) : -(len + 112);
    for (int idx = len; idx != 0; idx--) {
        int shiftbits = (idx - 1) * 8;
        long mask = 0xFFL << shiftbits;
        stream.writeByte((byte) ((i & mask) >> shiftbits));
    }
}
```

Figure 9: Snippets of readers/writers that were *unfolded* by TASSAL.

class is the most important, which package is the most important, how each of the classes fits in holistically and how often some methods or classes are called or created", in order to see the big picture before diving deeper into individual classes. Developer 5 thought that it would be helpful to include the ability to unfold method comments and bodies "so the person browsing [the summary] could dig in deeper in a way that suited them". This ability for the user to selectively unfold folded regions of code was subsequently implemented in our live demo of TASSAL (see Section VI). Developer 5 also preferred to understand the intention of a piece of code first and if they "had some doubts" then take a closer look, lending further support for the need for a tool such as TASSAL.

VI. Web Demo

We created a live demo of TASSAL [19] using the Play Framework (https://www.playframework.com) to showcase how it can be used to summarize open-source Java projects on GitHub (however note that TASSAL can summarize the source code of any Java project). Our demo of TASSAL can be found at http://groups.inf.ed.ac.uk/cup/tassal/demo.html and a video highlighting the main features of TASSAL can be found at https://youtu.be/_yu7JZgiBA4.

A screenshot of the demo is shown in Figure 10 and as one can see from the figure, the basic layout of the demo is very simple. On the left hand side is a tree view showing all the Java source files for a user-selected project on GitHub. Upon clicking on a source file, the remainder of the screen uses the Javascript-based ACE code editor (https://ace.c9.io) to show a summary of the file where less informative code regions have been folded. The user can adjust the conciseness of the summary using the compression ratio slider at top-left, ranging from viewing the complete file (0%) to folding all the foldable regions (100%). As a sanity check, the fold icons (⊞) to the left of the line numbers denote the code regions that were marked as folded by TASSAL. Note that while TASSAL is able to fold fields, we did not find a satisfactory way to implement this in ACE and therefore omitted it from the demo (however the fold icons for fields are still displayed).

If the user wishes to unfold a folded region, they can do so by clicking on the symbol denoting the fold ({⊟}) and conversely, they can fold any foldable region by clicking on the down arrow (▼) to the right of the line numbers as is standard in modern editors. One can see from the example (StatusLine.java from the bigbluebutton project displayed at 50% compression) that the header has been folded, as have the toString, getCode, getReason, clone and equals methods, i.e. Java boilerplate code — precisely the less salient code regions. Note also, how by *folding* the less informative regions the code *remains readable and navigable* and *no information is lost*. This is not true of other summarization approaches to source code [10], [11], [14]–[17], [35].

As for the choice of language model, using TASSAL with the topic model will in general produce better summaries (Section V), however training a topic model is too expen-
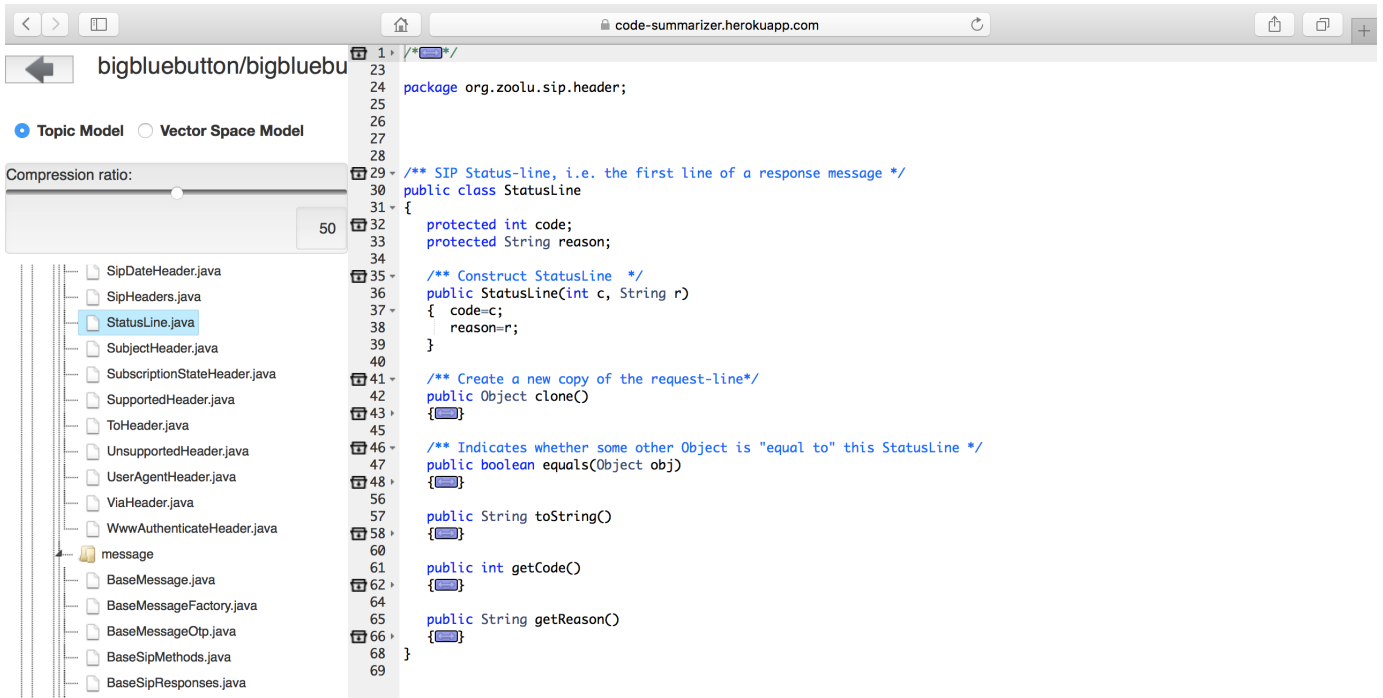
13

Figure 10: A screenshot of our source code autofolding tool being used to summarize `StatusLine.java` from `bigbluebutton`.

sive for an interactive system. Therefore, we train the topic model in advance on a small set of projects and cache the topic assignments. If the user requests summaries of GitHub projects for which we have not run the topic model, we fall back to the the VSM model. When both language models are available, the user can toggle between them by means of a radio button in the top-left corner.

VII. Discussion and Conclusions

We presented a novel fully unsupervised approach for extractive source code summarization, proposing that the folding procedure common to IDEs can serve as the basis of an automatic summary. We formulated this autofolding problem as an optimal subtree problem on the source code's AST. Our method incorporates a novel topic model for source code that identifies which tokens are most relevant to their enclosing files and projects. Our evaluation demonstrates that our summarizer outperforms several baselines, achieving an error reduction of 28%, is favoured by experienced developers and even outperforms methods used as standard in modern IDEs. Furthermore, our live demo showcases how our summarizer can be used to summarize open-source Java projects on GitHub.

As with all automatic summarization systems, TASSAL is not perfect and indeed any algorithm that performs summarization based solely on file topics cannot be, since language, be it natural or programming, has a rich semantic structure. However, by looking in detail at the summaries produced, we can gain an insight into situations where TASSAL fails to perform optimally and what improvements, if any, we can make. A detailed examination of the file summaries used in the developer study brought to light the following cases: (a) *Unfolding obvious comments.* Once a Javadoc comment containing very standard content (parameter and return types) and little additional information (stating that the method is a constructor) was unfolded. It is very difficult for for an automatic system to determine what constitutes an obvious comment as this requires deeper semantic information. (b) *Unfolding getters/setters with obvious names.* Once a getter with an obvious name was unfolded because it had an interesting method body. Again, it is difficult for for an automatic system to determine what constitutes an obvious name as this also requires more semantic information. (c) *Deciding whether a Javadoc or associated class/method body provides a better summary.* Once a class Javadoc provided a better summary of the class than the body of the class, which was itself interesting. Once again, an automatic system would require substantial semantic information to correctly decide this.

In future, we would like to extend our approach to generate targeted summaries for specific software engineering tasks such as bug localization or code review as well as investigating the possibility of folding at a statement (rather than block) level while maintaining a coherent summary. We would also like to explore the idea, suggested by one of the study participants, of creating a higher-level overview of class relationships that would allow a developer to see the bigger picture before delving into summaries of specific classes. Finally, in response to the shortcomings in TASSAL's current content model, we would like to see whether a more sophisticated content

model based on deep learning would be better able to capture the semantic information in the source code.

More broadly, NLP techniques for source code are only just beginning to be explored and have the potential for a much wider range of exciting applications from learning coding conventions to programming with natural language.

References

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 492–501.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[3] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *International Workshop on Program Comprehension (IWPC)*. IEEE, 2005, pp. 181–191.

[4] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 157–166.

[5] T. D. Hendrix, J. H. Cross II, L. A. Barowski, and K. S. Mathias, "Visual support for incremental abstraction and refinement in Ada 95," in *ACM SIGAda Ada Letters*, vol. 18, no. 6. ACM, 1998, pp. 142–147.

[6] B. Kullbach and V. Riediger, "Folding: An approach to enable program understanding of preprocessed languages," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2001, pp. 3–12.

[7] S. Rugaber, N. Chainani, O. Nnadi, and K. Stirewalt, "A conceptual model for folding," Georgia Institute of Technology, Tech. Rep. GT-CS-08-09, 2008.

[8] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 455–464.

[9] A. Haghighi and L. Vanderwende, "Exploring content models for multi-document summarization," in *Association for Computational Linguistics (ACL)*. ACL, 2009, pp. 362–370.

[10] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 12, 2012.

[11] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *Program analysis for software tools and engineering (PASTE)*. ACM, 2008, pp. 57–63.

[12] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 43–52.

[13] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.

[14] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2010, pp. 223–226.

[15] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Working Conference on Reverse Engineering (WCRE)*, 2010, pp. 35–44.

[16] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 13–22.

[17] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *International Conference on Program Comprehension (ICPC)*. ACM, 2014, pp. 291–294.

[18] A. T. T. Ying and M. P. Robillard, "Code fragment summarization," in *Foundations of Software Engineering (FSE)*. ACM, 2013, pp. 655–658.

[19] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "TASSAL: Autofolding for source code summarization," in *International Conference on Software Engineering (ICSE) Companion*. ACM, 2016, pp. 649–652.

[20] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Working Conference on on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.

[21] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[22] F. Jacob and R. Tairas, "Code template inference using language models," in *ACM Southeast Regional Conference (ACM SE)*. ACM, 2010, p. 104.

[23] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2006, pp. 3–12.

[24] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2013, pp. 532–542.

[25] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *International Conference on Software Engineering (ICSE)*, 2011, pp. 101–110.

[26] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2013.

[27] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *Association for Computational Linguistics (ACL)*, 2013.

[28] A. Cockburn and M. Smith, "Hidden messages: evaluating the efficiency of code elision in program navigation," *Interacting with Computers*, vol. 15, no. 3, pp. 387–407, 2003.

[29] K. Spärck Jones, "Automatic summarising: The state of the art," *Information Processing & Management*, vol. 43, no. 6, pp. 1449–1481, 2007.

[30] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011, pp. 125–134.

[31] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[32] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *Empirical Software Engineering*, pp. 1–38, 2013.

[33] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: approach for unsupervised bug report summarization," in *Foundations of Software Engineering (FSE)*. ACM, 2012, p. 11.

[34] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *International*

15

Conference on Software Engineering (ICSE). ACM, 2010, pp. 505–514.

[35] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 390–401.

[36] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching documents with examples: A corpus mining approach," *Transactions on Information Systems (TOIS)*, vol. 31, no. 1, p. 1, 2013.

[37] R. P. Buse and W. Weimer, "Synthesizing API usage examples," in *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 782–792.

[38] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *International Workshop on Mining Software Repositories (MSR)*. IEEE Press, 2013, pp. 319–328.

[39] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *International Workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 54–57.

[40] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "CodeTopics: which topic am I coding now?" in *International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 1034–1036.

[41] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "Topic XP: Exploring topics in source code using latent Dirichlet allocation," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–6.

[42] S. W. Thomas, "Mining software repositories using topic models," in *International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 1138–1139.

[43] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 543–562.

[44] M. Allamanis and C. Sutton, "Why, when, and what: analyzing stack overflow questions by topic, type, and code," in *Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 53–56.

[45] 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 *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 522–531.

[46] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *International Conference on Software Engineering (ICSE)*. Washington, DC, USA: IEEE, 2003, pp. 125–135.

[47] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2004, pp. 214–223.

[48] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIS*, vol. 41, no. 6, pp. 391–407, 1990.

[49] D. Mimno, W. Li, and A. McCallum, "Mixtures of hierarchical topics with pachinko allocation," in *International Conference on Machine Learning (ICML)*. ACM, 2007, pp. 633–640.

[50] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[51] T. Weninger, Y. Bisk, and J. Han, "Document-topic hierarchies from document graphs," in *International Conference on Information and Knowledge Management (CIKM)*. ACM, 2012, pp. 635–644.

[52] E. Erosheva, S. Fienberg, and J. Lafferty, "Mixed-membership models of scientific publications," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. Suppl 1, pp. 5220–5227, 2004.

[53] S. Guha, A. Moss, J. S. Naor, and B. Schieber, "Efficient recovery from power outage," in *Symposium on Theory of Computing (STOC)*. ACM, 1999, pp. 574–582.

[54] Eclipse-Contributors, "Eclipse JDT," http://www.eclipse.org/jdt/, 2013, visited December 21, 2013.

[55] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[56] M. Steyvers and T. Griffiths, "Probabilistic topic models," *Handbook of latent semantic analysis*, vol. 427, no. 7, pp. 424–440, 2007.

[57] D. Blei, "Probabilistic topic models," *Communications of the ACM*, vol. 55, no. 4, pp. 77–84, 2012.

[58] W. L. Buntine, "Operations for learning with graphical models," *arXiv preprint cs/9412102*, 1994.

[59] D. J. MacKay and L. C. B. Peto, "A hierarchical dirichlet language model," *Natural language engineering*, vol. 1, no. 03, pp. 289–308, 1995.

[60] H. M. Wallach, "Structured topic models for language," Ph.D. dissertation, 2008.

[61] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, 2004, pp. 74–81.

[62] S. Padó, *User's guide to* `sigf`*: Significance testing by approximate randomisation*, 2006.