Automatic Identification and Classification of Software Development Video Tutorial Fragments

Luca Ponzanelli, *Student Member, IEEE,* Gabriele Bavota, *Member, IEEE,* Andrea Mocci, *Member, IEEE,* Rocco Oliveto, *Member, IEEE,* Massimiliano Di Penta, *Member, IEEE,* Sonia Haiduc, *Member, IEEE,* Barbara Russo, Michele Lanza, *Senior Member, IEEE*

Abstract—Software development video tutorials have seen a steep increase in popularity in recent years. Their main advantage is that they thoroughly illustrate how certain technologies, programming languages, etc. are to be used. However, they come with a caveat: there is currently little support for searching and browsing their content. This makes it difficult to quickly find the useful parts in a longer video, as the only options are watching the entire video, leading to wasted time, or fast-forwarding through it, leading to missed information. We present an approach to mine video tutorials found on the web and enable developers to query their contents as opposed to just their metadata. The video tutorials are processed and split into coherent fragments, such that only relevant fragments are returned in response to a query. Moreover, fragments are automatically classified according to their purpose, such as introducing theoretical concepts, explaining code implementation steps, or dealing with errors. This allows developers to set filters in their search to target a specific type of video fragment they are interested in. In addition, the video fragments in CODETUBE are complemented with information from other sources, such as Stack Overflow discussions, giving more context and useful information for understanding the concepts.

Index Terms—Recommender Systems, Mining Unstructured Data, Video Tutorials

1 Introduction

SOFTWARE development often requires developers to face scenarios where they need to acquire additional knowledge beyond the one they already possess. Such scenarios include, for example, learning a new framework or a new programming language feature, as well as looking for information needed to fix a bug or to complete a programming task. The sources of information involved in this process are various, ranging from direct talks with other developers, to project and API documentation, either local or online. The web and online resources, such as forums, blogs, Question & Answer (Q&A) websites, slide presentations, etc. have become a precious source of information for developers [1], due to the amount and diversity of information they provide, as well as the ease of access.

When developers perform a search on the web, the returned result set is often heterogeneous, as textual artifacts can be accompanied by other non-textual sources like YouTube videos, providing tutorials on a specific topic. *Video tutorials* are a new and emerging source of information, such as step-by-step, learn-by-example introductions to how new technologies should be applied in practice.

- *L. Ponzanelli, G. Bavota, A. Mocci, and M. Lanza are with the Università della Svizzera italiana (USI), Lugano, Switzerland. E-mail: ponzanel, bavotag, moccia, lanzam@usi.ch*
- *M. Di Penta is with the University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it*
- *R. Oliveto is with the University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it*
- *S. Haiduc is with the Florida State University, USA. E-mail: shaiduc@cs.fsu.edu*
- *B. Russo is with the Free University of Bozen-Bolzano, Italy. E-mail: barbara.russo@unibz.it*

Manuscript received October 10, 2016; revised September 17, 2016.

A recent study by MacLeod *et al.* [2] investigated how and why software development video tutorials are created, and found that they share details such as software customization knowledge, personal development experiences, implementation approaches, and application of design patterns or data structures. The study also highlighted key advantages of video tutorials over other resources, such as the ability to visually follow the changes made to the source code, to see the environment where the program is executed, to view the execution results and how they relate to the source code, and to understand a development activity in depth by looking at different levels of detail. In essence, video tutorials can provide a learning experience different and complementary to that offered by traditional, text-based sources of information.

Despite these benefits, there is limited support for helping developers find relevant information within video tutorials, which are often lengthy and lack an index that allows finding specific fragments of interest. Thus, to find information about a specific concept in a video tutorial, a developer can either watch the entire video, wasting time watching irrelevant parts, or skim it, risking to miss important information. Moreover, a developer may need information from diverse sources to thoroughly understand a new concept. For example, when learning to use a new library, a developer could benefit from watching an introductory video tutorial, complemented by Q&A forum discussions about known issues or solutions to common problems of that library.

While existing approaches support developers by mining API documentation [3], [4] and Q&A websites [5], [6], or by synthesizing code examples from existing code bases [7], [8], [9], [10], there is no tool to leverage the relevant information found within video tutorial fragments and link them to other relevant sources of information available on the Web.

We propose CODETUBE, an approach to address these limitations and leverage the information found in video tutorials and other online resources to provide developers with relevant, concise, and holistic information about a topic.

CODETUBE starts from a set of videos relevant to a broad topic of interest (*e.g.,* Android development). It analyzes such videos to identify when source code is being shown (*e.g.,* through the IDE) on screen, using a series of algorithms and heuristics aimed at identifying shapes and fragments of Java code in a frame. It then identifies and isolates cohesive *video fragments*, *i.e.,* sequences of frames in which source code is being written, scrolled, or alternated with other informative material. The text contained in each video fragment is extracted and complemented with the text of the audio transcript occurring at the same time.

Machine learning techniques are then used to automatically classify the extracted video fragments into seven possible categories, namely (i) introduction to a tutorial topic, (ii) theoretical concepts, (iii) code implementation, (iv) working environment setup, (v) execution of implemented code, (vi) dealing with errors, and (vii) closing of a tutorial. These categories have been identified through an initial study (presented in Section 2), in two steps: (i) 41 computer science students/professors and professional developers manually identified and tagged a total of 784 fragments from 136 Java development video tutorials, and (ii) an open coding was performed on the obtained tags.

All the extracted information is then indexed using information retrieval techniques. Finally, CODETUBE searches and indexes Stack Overflow discussions relevant to each video fragment. A developer can then use a textual query (*e.g.,* "implementing an Android listener") to search CODETUBE through a web interface, apply filters for the "type" of video tutorial she is looking for (*e.g.,* dealing with errors), and obtain a ranked list of relevant video fragments with related Stack Overflow discussions. CODETUBE is currently available with a set of 3,526 indexed Java video tutorials retrieved from YouTube, corresponding to a total of 17,112 extracted fragments. The mean length of videos is $\sim$552s (median 437s), while the fragments have a mean length of $\sim$114s (median $\sim$78s).

We evaluated CODETUBE in three different studies:

1) We asked 41 people to manually identify and tag fragments starting from a corpus of 150 video tutorials. We used this data as ground truth to assess the accuracy of CODETUBE in identifying video fragments and in correctly classifying the type of video tutorial fragments among the considered categories.
2) We asked 34 Android developers to evaluate (i) the relevance of the video fragments retrieved by CODETUBE for a given query compared to the results returned by YouTube, (ii) the coherence and conciseness of the produced video fragments, and (iii) the relevance and complementarity of Stack Overflow discussions returned by CODETUBE for specific video fragments.
3) We performed an extrinsic evaluation of the approach by introducing CODETUBE to three leading developers involved in the development of Android apps. We then asked them questions about the usefulness of CODETUBE, focusing on the value of extracting fragments from video tutorials, and that of combining different sources of information.

The results show that:

1) The fragments automatically extracted by CODETUBE from a video tutorial are fairly similar (MoJoFM=77%) to the ones manually identified by people looking at the video (*i.e.,*CODETUBE and people identify similar cohesive fragments in a given tutorial).
2) CODETUBE is able to classify fragments along the seven identified categories with an average accuracy of 72% and an Area Under the Receiving Operating Characteristic Curve (AUROC) of 0.92.
3) Ca. 80% of the video fragments extracted by CODETUBE are considered cohesive and ca. 60% are considered self-contained by developers.
4) The Stack Overflow discussions returned by CODETUBE for a given video fragment are considered relevant to it in ca. 50% of cases, while they are almost always considered complementary in terms of the information presented with respect to the video fragment ($\sim$90%).
5) The three project managers involved in the third study see great value in CODETUBE and a great potential in this line of work.

Paper structure. In Section 2 we report the design and results of a study aimed at (i) identifying the categories of development video tutorial fragments and (ii) investigating the structure of video tutorials. This was done by asking 41 people to manually identify and categorize video fragments in video tutorials. In Section 3 we describe CODETUBE and its underlying approach for both video fragmentation and classification of tutorial fragments. In Section 4 we leverage the data collected in the study reported in Section 2 to objectively evaluate the ability of CODETUBE in automatically fragmenting and classifying video tutorials. In Section 5 we perform a second study aimed at providing a subjective assessment of the quality of the extracted video fragments as perceived by developers, as well as the relevance and complementarity of the video fragments to the linked Stack Overflow discussions. Section 6 describes a qualitative evaluation of CODETUBE with actual practitioners with the goal of evaluating its industrial applicability. Threats to validity are discussed in Section 7. After a discussion of the related literature (Section 8), Section 9 concludes the paper.

2 INVESTIGATING THE STRUCTURE OF VIDEO TUTORIALS

Previous research on development video tutorials [2] has investigated the motivation and purpose of *entire* development video tutorials, but has not analyzed their structure and content in depth. A video tutorial has an intrinsic structure embedded in the flow of actions performed by the tutor. When it comes to devising an automated approach to analyze, fragment, classify, and index video tutorials, understanding the structure of the original video is essential to provide, for example, advanced searching features.

The *goal* of this initial study was twofold: to determine how people segment video tutorials into coherent parts/sections, and what kind of parts are typically composing a software development video tutorial (*e.g.,* setting

TABLE 1
Participants' Occupation, Experience in Java, and Usage of Video Tutorials.

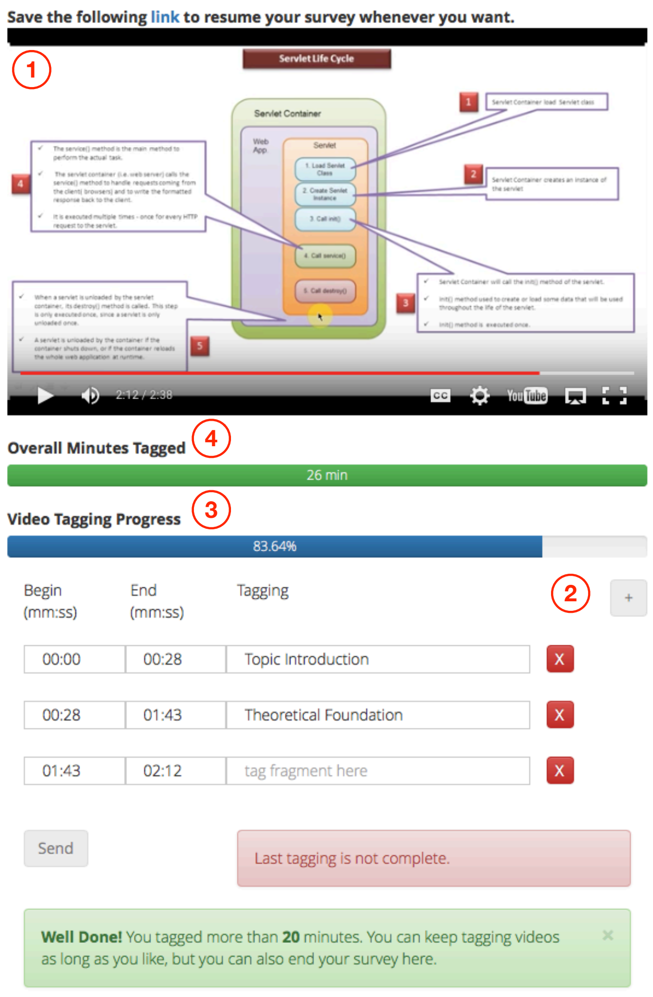
Occupation	Total	%	Experience in Java	Total	%	Usage of Video Tutorials	Total	%
Faculty	1	2%	Less than 1 year	29	71%	Daily	29	71%
PhD Student	3	7%	1-3 years	7	17%	Few times a week	7	17%
Master Student	4	10%	3-5 years	3	7%	Few times a month	3	7%
Undergraduate Student	31	76%	5-10 years	2	5%	Rarely	2	5%
Professional Software Developer	2	5%	More than 10 years	0	0%	Never	0	0%
Total	41	100%	**Total**	41	100%	**Total**	41	100%

of the IDE, code writing, *etc.*). Collecting this information is important to determine the ability of any approach - CODETUBE in this case - to automatically identify and label these fragments. The *context* consists of objects, *i.e.*, 150 tutorials collected from YouTube, and participants, *i.e.*, 41 computer science students/professors and professional developers manually identifying and tagging the different tutorial parts (*e.g.*, "from 1:00 to 3:30 the tutorial shows how to set up the IDE"). We later use the fragment beginning and end collected in this study to construct a ground truth for measuring the ability of CODETUBE to identify video tutorial fragments (Section 4.1).

2.1 Context, Data Collection & Analysis

We manually selected from YouTube the video tutorials used in the context of our tagging study. The manual collection was needed to ensure the selection of real tutorials dealing with a heterogeneous set of topics at different levels of abstraction (*e.g.*, theoretical *vs* practical tutorials). We selected (i) 50 generic Java tutorials, (ii) 50 tutorials dealing with JSPs and Servlets (*i.e.*, Java Web applications), and (iii) 50 Android-related tutorials (*i.e.*, Java mobile apps). We made sure to include both tutorials for beginners as well as for experienced developers and to select a mix of theoretical and practical tutorials. For example, the 50 Java-related tutorials included tutorials about Java basics (*e.g.*, exceptions handling), advanced topics (*e.g.*, multi-threading), and theoretical notions (*e.g.*, how the garbage collector works). The selection of such tutorials was performed by one author and double-checked by a second author. All 150 tutorials focus on the Java programming language, because (i) as it will be detailed later, our approach leverages a Java island parser [11], [12], [13] to identify code constructs shown in the video tutorial, and (ii) this eased the selection of participants for our study. Also, to limit the effort required from participants, we did not include video tutorials longer than 20 minutes.

We invited, after selecting participants through convenience sampling [14], [15], 55 computer science students and professors, as well as five industrial software developers to participate in our study. In order to limit any hypothesis guessing or bias, we did not reveal to participants the final purpose of the tagging nor our usage of the obtained data. Each participant received an email with a link to the web application where they could read instructions and then perform the fragment splitting and tagging tasks. We asked each participant to watch video tutorials and to split them into categorized fragments: they had to identify disjoint parts of the video tutorials and tag each with a category explaining its main purpose (*e.g.*, "from 1:00 to 3:30 it explains how to set the working environment, from 3:31 to 5:00 it shows how to implement a JSP"). We asked participants to extract and tag fragments for at least 20 minutes of video tutorials. Participants were free to tag more. Invitees had up to two months to perform the tasks. Data about the 41 participants is reported in Table 1.


Save the following link to resume your survey whenever you want.

Servlet Life Cycle

1 Servlet Container load Servlet class

2 Servlet Container creates an instance of the servlet

3
- ✓ Servlet Container will call the init() method of the servlet.
- ✓ init() method used to create or load some data that will be used throughout the life of the servlet.
- ✓ init() method is executed once.

4
- ✓ The service() method is the main method to perform the actual task.
- ✓ The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.
- ✓ It is executed multiple times - once for every HTTP request to the servlet.

5
- ✓ When a servlet is unloaded by the servlet container, its destroy() method is called. This step is only executed once, since a servlet is only unloaded once.
- ✓ A servlet is unloaded by the container if the container shuts down, or if the container reloads the whole web application at runtime.

Overall Minutes Tagged

26 min

Video Tagging Progress

83.64%

Begin (mm:ss)	End (mm:ss)	Tagging
00:00	00:28	Topic Introduction
00:28	01:43	Theoretical Foundation
01:43	02:12	tag fragment here

Last tagging is not complete.

Well Done! You tagged more than **20** minutes. You can keep tagging videos as long as you like, but you can also end your survey here.

Fig. 1. User Interface of the Fragment Tagging Web Application.

We implemented a web application to support the tagging process by participants in multiple rounds (*e.g.*, tagging one tutorial today, and another one after one week). After registering, participants provided some background information which we report in Table 1. The actual tagging starts through the user interface depicted in Figure 1. The web application shows tutorials embedded in a YouTube video player (1). The tagging application was designed to balance the number of participants tagging each video tutorial, *i.e.,* in a first iteration, the web application assigned each video tutorial in our dataset to at least one participant. Then, if possible, a second participant was assigned, and so on.

At the bottom of the page there are controls to allow participants to create and tag fragments—see Figure 1 (2). Participants are allowed to freely interact with the video player as they wish, yet they are forced to follow some constraints when devising the fragments. All fragments need to be contiguous or the application does not allow the user to store the tagging session. Also, the fragments have to cover the whole video. A progress bar (3) allows the users to keep track of the amount of video covered by the fragments already devised. To avoid corrupted data, the application raises an error if the tagging progress goes beyond 100%, or if any tag is missing, and does not allow to store the session if the video coverage is not complete.

Last, each user is requested, but not forced, to tag at least 20 minutes of video tutorials. Another progress bar (4) shows the overall minutes tagged by the study participants. Once the bar gets to 20 minutes, the participant is notified with a pop-up label at the bottom of the application, but the application leaves the participant the choice to keep tagging video fragments. Each participant tagged on average 29 minutes of video tutorials (min 7, median 27, max 75).

We collected 784 tagged video fragments (1,219 minutes) from 136 video tutorials. 14 of the 150 video tutorials we selected were not analyzed by any participant, while the remaining 136 were tagged by at least one participant each. Two of the authors performed an open coding process on the 784 tagged fragments to group the tags into categories. Such a grouping was independently performed by the authors by analyzing the tags assigned by the participants as well as by personally watching each video fragment. After the first coding, the two authors disagreed on 53 fragments, for which one of them produced an *Unclassified* categorization, whereas the other produced a category. The inter-rater agreement after the first coding phase—computed in terms of Cohen's Kappa [16]—is equal to 0.86, which is considered a very strong agreement. The two authors met to discuss and refine (or merge) the identified categories, reaching an agreement when needed.

The output of this process is a set of categories that can be used to describe the different parts composing software development video tutorials. To give an example, 144 fragments were marked by participants with tags clearly referring to the *introduction of the tutorial topic* (*e.g.,* "introduction", "topic introduction", "tutorial introduction", *etc.*).

2.2 Analysis of the Results

2.2.1 Categories of video tutorial fragments

Table 2 reports the seven categories of video tutorial fragments derived from our open coding procedure.

For 36 fragments we were not able to understand the meaning of the tags assigned by the participants—see the *Unclassified* row—(*e.g.,* details, observations). We excluded these tags from our study.

Most tagged fragments (37%) refer to *code implementation* activities. Examples of tags in this category include "program writing", "JComboBox implementation", and "implementing a JSP page". *Introduction to the tutorial topic* is the second most popular category, grouping 18% of the assigned tags. It concerns parts of the tutorial where the main tutorial topic is presented from a general point of view without providing implementation details. This category is followed by *execution of the implemented code* (15%), including 124 fragments in total.

TABLE 2
Categories Resulted from the Open Coding Process.

Category	**Tags**	**%**
Code implementation (CI)	288	37%
Introduction to the Tutorial Topic (ITT)	144	18%
Execution of the Implemented Code (EIC)	124	15%
Theoretical Concepts (TC)	87	11%
Closing of the Tutorial (CT)	47	6%
Environment Setup (ES)	39	5%
Dealing with Errors (DE)	19	3%
Unclassified	36	5%
Total	784	100%

The latter category includes tags like "deployment and execution of the implemented web app" and "program execution and test logger". *Theoretical concepts* (*i.e.,* when some specific aspects of the topic are explained in detail, possibly interleaving slides/discussions with some code examples) are in 11% of the tagged fragments (*e.g.,* "explaining HTTP status codes") while the *working environment setup* category groups 39 (5%) tags (*e.g.,* "IDE settings"). Finally, 47 tags (6%) are related to the *closing of the tutorial* and 19 (3%) to explanations on how to *deal with errors* one could encounter while implementing the topics discussed in the video tutorial (*e.g.,* "what happens if we do not properly configure log4j").

These seven categories are the ones considered in CODETUBE when automatically classifying the type of the extracted video tutorial fragments.

2.2.2 Structure of Video Tutorials

The availability of tagged and classified video fragments allows us to provide—without generalizing beyond the samples on Java development—an idea of how development video tutorials are structured.

Table 3 describes the patterns—intended as sequences of fragments belonging to the seven identified categories—that occur at least twice in the analyzed dataset.

Surprisingly, the most common pattern (p1, 9 instances) is not related to explaining implementation concepts, but rather to illustrating the setup of a specific piece of technology. Other frequent patterns (p2, p3, p5, p6, p7, p8) follow typical scenarios in which there is a topic introduction, possibly followed by theoretical concepts and/or some environment setup; then, implementation details are shown in one or more fragments and, finally, the tutorial is in some cases closed by some concluding remarks.

In some cases (p4, with 4 instances, or p11, with 3 instances) the tutorial does not show the IDE with the source code at all, but just provides some theoretical elements. On the other hand, there are some kinds of video tutorials (p8, with 4 instances, p15 with 2 instances, and p20, with 2 instances), which are mainly screencasts showing implementation details in the IDE (preceded by an introduction for p15 and followed by closing remarks for p20).

TABLE 3
Video tutorial composition patterns, their occurrence, and their typical structural sequence.

Pattern	Occurrences	Sequence
p1	9	ITT, ES
p2	7	ITT, CI, EIC
p3	7	ITT, CI+
p4	6	ITT, TC+
p5	5	ITT, CI+, CT
p6	5	ITT, ES, CI+
p7	4	ITT, TC, CI
p8	4	CI+
p9	3	ITT, CI, TC+
p10	3	ITT, CI, EIC, CI, CT
p11	3	TC
p12	2	ITT, TC, CI, CT
p13	2	ITT, TC, DE, TC
p14	2	ITT, CI, DE, CT
p15	2	ITT, CI, CT
p16	2	ITT, ES, EIC+
p17	2	ITT, ES, (CI, EIC)+
p18	2	ITT, (CI, TC)+, CT
p19	2	ITT, (CI, EIC)+, CT
p20	2	CI+, CT

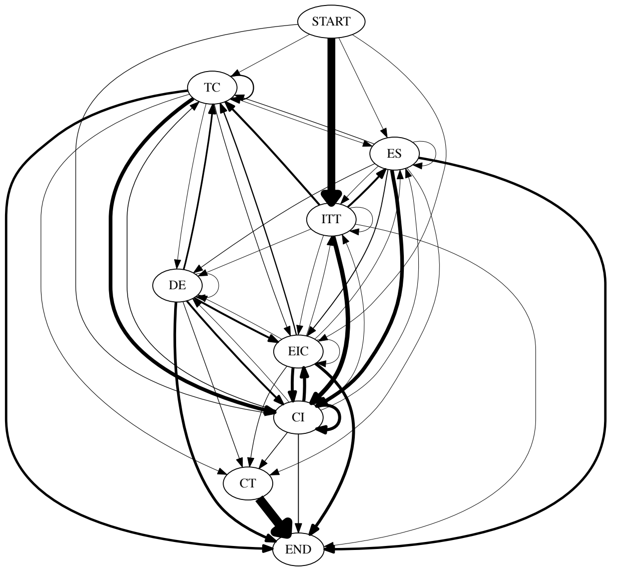


Fig. 2. Transition graph between the different parts of the video tutorials.

Some patterns (p10, p16, p17, p19) reveal a scenario in which the execution is being shown after having explained its implementation, and in some cases (p17 and p19) there are multiple iterations of implementation details along with a related execution.

In some cases (p13 and p14) the explanation of theoretical concepts or of implementation details is followed by specific fragments to explain how to deal with erroneous conditions.

Table 4 and Figure 2 report and depict the transition frequencies—estimated across all videos in our dataset—between different types of video tutorial fragments, in order to get an overview of how these transitions generally occur in the analyzed dataset.

While the table reports exact frequency values, the figure depicts the structure of video tutorials in the form of a Markov chain, where thicker transition edges indicate higher probabilities. In 85% of cases an introduction about the tutorial topics is provided. In the remaining 15% of cases, the tutorials directly start with an implementation activity (7%), the setting of the working environment (4%), an explanation of theoretical concepts (3%), or the execution of the code that will be the object of the tutorial (1%).

After the topic introduction, 49% of the tutorials deal with code implementation activities, often representing the bulk of software development tutorials. A typical transition is START→ITT→ES→CI, starting with a topic introduction (START→ITT=85%), continuing with setting up the working environment (ITT→ES=21%) and then starting an implementation activity (ES→CI=21%). Other tutorials (22%), focusing more on theoretical aspects, start explaining theoretical concepts right after the topic introduction.

In 37% of cases a code implementation fragment is followed by another one (CI→CI=37%), because the tutorial features independent implementation activities (*e.g.*, how to use method A and method B of a given API). Other frequent transitions happen from code implementation to code execution (38%), often (48%) followed by another code implementation activity (*i.e.*, a transition CI→EIC→CI).

Outgoing transitions from fragments dealing with theoretical concepts (TC node in Figure 2), are generally followed by implementation activities (TC→CI=55%) or by another theoretical fragment on a different concept (TC→TC= 17%). Instead, those outgoing from fragments dealing with common errors (DE) are almost equally distributed between: (i) theoretical concepts (DE→TC=25%), explaining why a specific error arises, (ii) the execution of the implemented code (DE→EIC=30%), showing how the error manifests at execution time, and (iii) code implementation activities (DE→CI=30%), showing how to fix the error.

The results discussed above highlight how the latent structure of a video tutorial can be quite complex. This recalls (and justifies) the need for an approach to automatically navigate among fragments to search/browse video tutorials and pinpoint the interesting parts.

3 CODETUBE OVERVIEW

CODETUBE is a multi-source documentation miner to locate useful pieces of information for a given task at hand. The results are fragments of video tutorials relevant for a given textual query, augmented with additional information mined from other "classical", text-based online resources. The analysis of video tutorials is currently limited to videos in English dealing with the Java programming language.

Figure 3 depicts the CODETUBE pipeline. It is composed of (i) an offline analysis phase aimed at collecting and indexing video tutorials and other resources, and (ii) an online service where developers can search these processed resources. In the following we detail each step.

3.1 Crawling and Analyzing Video Tutorials

The first step of the process is defining the topics of interest. The user provides (i) a set of queries Q describing the video

TABLE 4
Transition frequencies between different parts of the video tutorials.

	ITT	TC	EIC	CI	ES	DE	CT	END
START	84.62%	3.50%	0.70%	6.99%	4.20%	0.00%	0.00%	0.00%
Introduction Tutorial to Topic (ITT)	3.03%	21.97%	2.27%	48.48%	21.21%	2.27%	0.00%	0.76%
Theoretical Concepts (TC)	0.00%	12.64%	6.90%	40.23%	3.45%	4.60%	5.75%	26.44%
Execution Implemented Code (EIC)	2.70%	12.61%	5.41%	32.43%	1.80%	5.41%	7.21%	32.43%
Code Implementation (CI)	1.19%	7.94%	33.73%	33.33%	0.79%	4.37%	8.33%	10.32%
Environment Setup (ES)	2.38%	7.14%	9.52%	40.48%	2.38%	7.14%	4.76%	26.19%
Dealing with Errors (DE)	0.00%	17.86%	21.43%	21.43%	0.00%	3.57%	7.14%	28.57%
Closing of the Tutorial (CT)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%

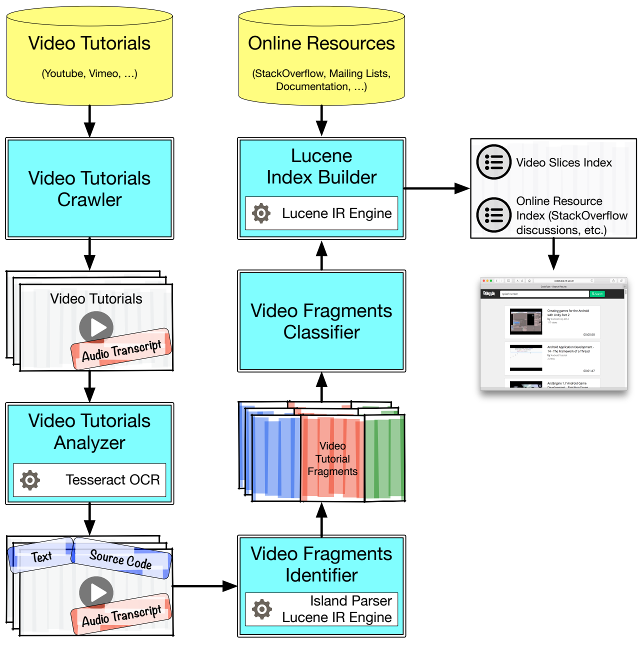


Fig. 3. CODETUBE: Analysis process.

tutorials she is interested in (*e.g.*, "Android development") and (ii) a set of related tags T to identify and index relevant Stack Overflow discussions (*e.g.*, "Android"). Each query in Q is run by the *Video Tutorials Crawler* using the YouTube Data API[1] to get the list of YouTube channels relevant to the given query $q_i \in Q$.

For each channel the *Video Tutorials Crawler* retrieves the metadata (*e.g.*, video URL, title, description) and the audio transcripts, which are either automatically generated by Google with speech recognition or written by the author. Using GOOGLE2SRT[2] we extract the contents of the transcriptions for the videos. The crawling of video meta-information is performed on YouTube, but it can be extended to any video streaming service or video collection where the same type of meta-information and transcripts are available or can be extracted, *e.g.*, using an off-the-shelf speech recognition API.

Once the videos have been crawled, their metadata is provided as input to the *Video Tutorial Analyzer*. It analyzes each video and extracts pieces of information to isolate video fragments related to a specific topic. The *Video Tutorial Analyzer* aims at characterizing each video frame with the text and the source code it contains. It uses multi-threading to concurrently analyze multiple batches of videos.

3.1.1 Frame Extraction

The analysis starts by downloading the video at the maximum available resolution. CODETUBE uses the multimedia framework FFMPEG[3] to extract one frame per second, saving each frame in a png image. Given the set of frames in the video, we compare subsequent pairs of frames (f_i, f_{i+1}) to measure their dissimilarity in terms of their pixel matrices. If they differ by less than 10% we only keep the first frame in the data analysis since the two frames show *almost* the same information. This scenario is quite common in video tutorials where the image on the screen is fixed for some seconds while the tutor speaks. To have an idea of the impact of this optimization step in the computational cost of our process, we measured the percentage of frames removed by this heuristic on a set of 100 Java video tutorials randomly selected from the YouTube videos currently indexed in CODETUBE. We found that, on average, 86% of the frames are removed (median=88%, standard deviation=9%). This means moving from an average of 463 frames per video to analyze, to only 64, thus substantially reducing the computational cost of our process. While this strong reduction in the number of frames might look surprising, it is worth noticing that a video having a fixed image on the screen for 20 seconds (*e.g.*, a slide shown while the tutor speaks) will experience a percentage reduction of the number of frames to analyze of 95% (*i.e.*, one out of the twenty frames will be kept for the analysis).

After obtaining the reduced set of frames to analyze, CODETUBE performs the following *information extraction steps*.

3.1.2 English Terms Extraction

We use the OCR (Optical Character Recognition) tool TESSERACT-OCR[4] to extract the text from the frame. OCR tools are usually designed to deal with text on white background (*i.e.*, paper documents). In order to cope with this, many OCR tools convert colored images to black and white before processing them. When using an OCR tool on video frames, the high variability of the background, and the potential low quality of a frame can result in a high amount of noise. Thus, after splitting composite words—based on camel case or other separators—we use a dictionary-based filtering, to

1. https://developers.google.com/youtube/v3/
2. http://google2srt.sourceforge.net/en/
3. http://www.ffmpeg.org/
4. https://github.com/tesseract-ocr

ignore strings that are invalid English words[5]. Note that, while this helps in filtering out possible noise generated by the OCR tool, this step can also remove some interesting technical terms (*e.g.,* `https`) not part of the standard English dictionary. The definition of a customized and more technical dictionary is part of our future research agenda.

3.1.3 Java Code Identification

In principle, the output of the OCR could be processed to extract the depicted Java constructs. However, such output often contains noise. Figure 4 shows three frames containing Java code. In Frame 1 the code occupies the whole screen, and there is a clear background: The noise of the OCR output is limited. The noise increases in the Frames 2 and 3, due to the buttons, menu labels, the graphics on the t-shirt, *etc.* To limit the noise produced by the OCR we identify the sub-frame containing code using two heuristics, *shape detection* and *frame segmentation*.

Shape Detection. We use BoofCV[6] to apply shape detection on a frame, identifying all quadrilaterals by using the difference in contrast in the corners. This is typically successful to detect code editors in the IDE as in Frame 2.

Frame Segmentation. The shape detection phase could fail in identifying sub-frames with code. In Frame 1 and Frame 3 of Figure 4 BoofCV fails because of missing quadrilaterals.

In this case, we apply a segmentation heuristic by sampling small sub-images having height and width equal to 20% of the original frame size and we run the OCR on each sub-image. We mark all sub-images S_m containing at least one valid English word and/or Java keyword and we identify the part of the frame containing the source code as the quadrilateral delimited by the top-left sub-image (*i.e.,* the one having the minimum x and y coordinates) and the bottom-right sub-image (*i.e.,* the one having the maximum x and y coordinates) in S_m. The heuristic might be imprecise. For example, in Frame 2, if the quadrilaterals identified by the shape detection algorithm are ignored, the heuristic identifies the whole frame as code area. Another example is Frame 3, where the opening parenthesis of the `if` statement is left outside the code area, thus breaking the integrity of the code snippet.

Identifying Java Code. After identifying a candidate sub-frame, we run the OCR to obtain the raw text that likely represents code. Then, we use an island parser [11], [12] on the extracted text to cope with the noise, the imperfections of the OCR, and the incomplete code fragments. The island parser separates invalid code or natural language (water) from matching constructs (islands), and produces a Heterogeneous Abstract Syntax Tree (H-AST) [13]. By traversing the H-AST we can exclude water nodes and keep complete constructs (*e.g.,* declarations, blocks, other statements) and incomplete fragments (*e.g.,* partial declarations, like methods without a body). If we are not able to match complete or incomplete Java constructs with any of the described heuristics, we assume that the frame does not contain source code.

5. We use the OS X English dictionary.
6. http://boofcv.org/

3.2 Identifying Video Fragments

The *Video Fragments Identifier* detects cohesive fragments in a video tutorial using the previously collected information. We refer to Figure 5 to illustrate the performed steps.

CodeTube starts by identifying video fragments characterized by the presence of a specific piece of code. The conjecture is that a frame containing a code snippet is coupled to the surrounding video frames showing (parts of) the same code.

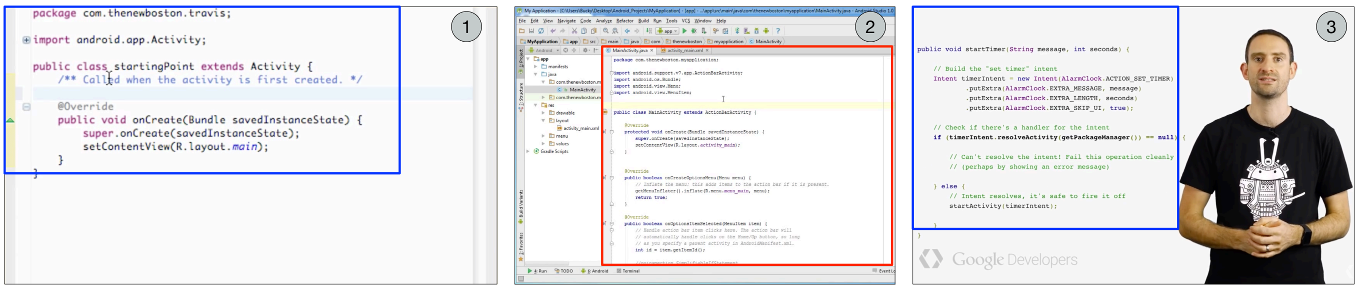
Identifying the video frames containing a specific code snippet presents non-trivial challenges: A piece of code could be written incrementally during a video tutorial: If writing a Java class lasts 3 minutes, all frames will contain snippets of code related to that class and thus should be considered as part of the same video fragment. However, such code snippets contain different programming constructs due to the incremental writing. Second, to provide a line-by-line explanation, the tutor could scroll the code snippet shown on video. This causes frames showing the same code snippet to show different portions of it. Last, the tutor could interleave two frames showing the same code snippet with slides or other applications, *e.g.,* the Android emulator.

CodeTube overcomes these challenges and identifies video fragments characterized by the presence of a specific piece of code by comparing subsequent pairs of frames containing code to verify if they refer to the same code snippet. The frames depicted in red in Figure 5 represent *code frames*, frames containing code fragments. Given two code frames CodeTube verifies if they contain at least one common complete or incomplete Java construct. If so, the two frames are marked as containing the same code component. If not, we cannot exclude that the two frames do not refer to the same code. We have to take into account (i) possible OCR imprecisions when extracting the source code from the two frames (*e.g.,* a Java construct is correctly extracted only in one frame), and (ii) scrolling from one frame to another has hidden some constructs in one of the two frames.

If the island parser fails in matching a common construct in the two frames, we compute the Longest Common Substring (LCS) between the pixel matrices representing the code frames. Specifically, we represent matrices as strings, where each pixel is converted to a 8-bit grayscale representation. If the LCS between the two frames includes more than γ of the pixels in the frames, CodeTube considers the two frames as showing the same code snippet. The process adopted to tune the threshold γ is reported in Section 3.5.

Note that the LCS is not affected by possible OCR imprecisions, and it can deal with the IDE scrolling, as shown in Figure 6 (in cyan the portion of the two frames identified as LCS).

As a drawback, LCS is sensitive to zooming. Since the alignment of the proportions between two subsequent frames changes, LCS would fail in identifying a common part. Overall, given the advantages of the LCS over the Java constructs matching between the two frames via island parser, one may think that applying the LCS for each pair of code frames is the way to go. Unfortunately, the LCS is computationally expensive due to the huge number of pixels composing a frame (a 1080p HD video has ~2M pixels per frame). For this reason, we adopt the LCS as a contingency



```
package com.thenewboston.travis;

import android.app.Activity;

public class startingPoint extends Activity {
    /** Called when the activity is first created. */

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

```
package com.thenewboston.myapplication;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;


public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
```

```
public void startTimer(String message, int seconds) {

    // Build the "set timer" intent
    Intent timerIntent = new Intent(AlarmClock.ACTION_SET_TIMER)
            .putExtra(AlarmClock.EXTRA_MESSAGE, message)
            .putExtra(AlarmClock.EXTRA_LENGTH, seconds)
            .putExtra(AlarmClock.EXTRA_SKIP_UI, true);

    // Check if there's a handler for the intent
    if (timerIntent.resolveActivity(getPackageManager()) == null) {

        // Can't resolve the intent! Fail this operation cleanly
        // (perhaps by showing an error message)

    } else {
        // Intent resolves, it's safe to fire it off
        startActivity(timerIntent);

    }
}
```

Fig. 4. Example frames from which CODETUBE is able to extract code fragments. (1) http://y2u.be/eKXnQ83RU3I, (2) http://y2u.be/NMDPxN8FgXM, (3) http://y2u.be/jQWB_-o1kz4. The red rectangles are the quadrilaterals identified by BoofCV. The blue quadrilaterals are the code areas identified by our segmentation heuristic.

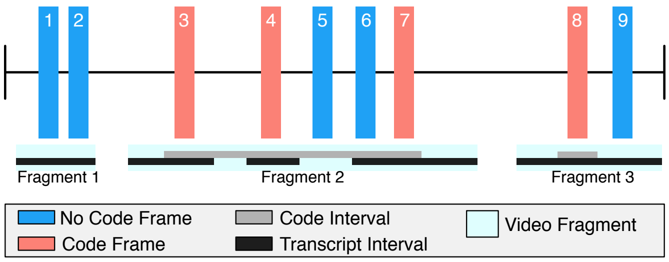
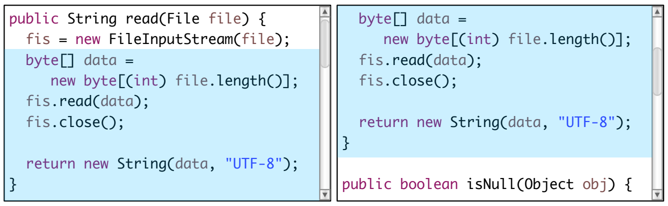


Fig. 5. Identification of video fragments.



```
public String read(File file) {
  fis = new FileInputStream(file);
  byte[] data =
     new byte[(int) file.length()];
  fis.read(data);
  fis.close();

  return new String(data, "UTF-8");
}
```

```
  byte[] data =
     new byte[(int) file.length()];
  fis.read(data);
  fis.close();

  return new String(data, "UTF-8");
}

public boolean isNull(Object obj) {
```

Fig. 6. LCS between two frames showing the same code. The right frame is scrolled down by the tutor.

strategy when the island parser is unable to identify common Java constructs in the two frames under analysis. To speed up the LCS computation we scale the frames to 25% of their size. In the example depicted in Figure 5, CODETUBE compares the code frame pairs (3,4), (4,7), and (7,8), identifying the first two pairs as containing the same code snippet. As highlighted by the grey line below the frames, it identifies the first two cohesive "code intervals", *i.e.*, the first going from Frame 3 to Frame 7 and the second containing Frame 8 only. The *non-code frames* 5 and 6 (blue in Figure 5) are included in the first code interval, since they are surrounded by two code frames (4 and 7) containing the same snippet.

To assess the reliability of the LCS in identifying related code frames we extracted all pairs of code frames for which the LCS is invoked in CODETUBE (*i.e.*, pairs of code frames for which the island parser fails to identify any code overlap) from the same set of 100 Java video tutorials previously used to test the frames reduction heuristic. This resulted in the extraction of 6,266 pairs of frames that we stored as .png files. Out of these, 182 are identified by the LCS as showing the same code snippet[7]. The first author manually analyzed these 182 pairs as well as 182 randomly selected pairs for which the LCS does not report an overlap of the pixels in the frame higher than γ (*i.e.*, these pairs are not marked as showing the same code snippet in CODETUBE). The goal of the manual validation was to verify whether the LCS is able to correctly classify the pairs of code frames as containing/not containing the same code snippet. The results of such a manual validation are reported in the confusion matrix shown in Table 5: The columns report the classification automatically made by the LCS, while the rows show the manual classification made by the first author. *True* indicates that the two code frames contain the same code snippet (based on the LCS output or on the evaluator judgment), while *false* indicates no code overlap.

The results reported in Table 5 show that when the LCS indicates the two frames as reporting the same code snippet, it rarely fails. Indeed, its precision when reporting code overlap is 95% (173 out of 182 pairs correctly classified as showing the same code snippet). Instead, when no overlap is detected by the LCS, the accuracy of such a heuristic drops to 24% (43 out of 182 correctly classified as not showing the same code snippet). The manual analysis of these cases showed that the LCS heuristic is frequently failing in two cases. First, and as expected, the LCS fails in case of zooming, totally missing the detection of code snippets shared in the two frames. Second, it happens very frequently in video tutorials that, while coding in the IDE, the tutor opens a small window covering a large part of the code shown in the IDE. In these cases, while the human evaluator can still see the few statements of code not covered by the window and map them to the previous frame, the LCS heuristic does not work. Devising more advanced strategies to successfully deal with these cases is part of our future work agenda.

Once analyzed the code in the frames, in a subsequent step CODETUBE analyzes the audio transcripts (black lines

TABLE 5
Accuracy of the LCS heuristic.

Manual \ LCS	true	false
true	**173**	139
false	9	**43**

7. We used as value for the γ threshold the one identified during the parameters' tuning process detailed in Section 3.5.

at the bottom of Figure 5) to refine the already identified code intervals (grey lines). CODETUBE identifies the audio transcripts starting and/or ending inside each code interval. The audio transcripts are provided in the SubRip[8] format when extracted from YouTube's videos. In the example reported in Figure 5, three audio transcripts are considered relevant when refining the code interval going from frame 3 to 7. CODETUBE uses the beginning of the first and the end of the last relevant audio transcript for a code interval to extend its duration and avoid that the code interval starts or ends with a broken sentence. The extended code interval represents an identified video fragment (Fragment 2—light cyan in Figure 5).

There might still be non-code frames in the video that have not been assigned to any video fragment (*e.g.,* frames 1 and 2 in Figure 5). These frames are grouped together on the basis of the audio transcript part they fall in. For example, the first two frames in Figure 5 are grouped in the same video fragment (Fragment 1), since they both fall in the same audio transcript part.

As a final step, each subsequent pair of fragments is compared to remove very short video fragments and to merge semantically related fragments. CODETUBE merges two subsequent fragments if:

1) Their textual similarity (computed using the Vector Space Model (VSM) [17]) is greater than a threshold λ. Each video fragment is represented by the text contained in its audio transcripts and in its frames (as extracted by the OCR). The text is pre-processed by removing English stop words, splitting by underscore and camel case, and stemming with the Snowball stemmer[9].
2) One fragment is shorter than μ seconds. This is done to remove short video fragments that unlikely represent a complete and meaningful fragment of a video tutorial.

3.3 Features Construction for the Classification

The *Video Fragment Classifier* is in charge of classifying them into one of the seven categories obtained as output of the study presented in Section 2 (see Table 2).

There are different aspects to consider when devising an approach to automatically classify complex objects like video fragments. The information characterizing a video fragment is heterogeneous, and it includes (i) temporal aspects, *e.g.,* the position in the video with respect to other fragments, (ii) structural features, *e.g.,* presence of shapes on the screen, (iii) semantic features, *e.g.,* textual topics, and (iv) information concerning the source code being shown on the screen. We present all features involved in the construction of the feature vector used by the machine learning algorithm to automatically classify a given video fragment.

3.3.1 Temporal Features

Some types of video fragments have a tight relationships with their temporal position in the video, and with their duration as well. Specifically, we consider the following three features:

Beginning Time, expressed as percentage of the whole video (*i.e.,* `begin_time/video_length`). Identifying the position of the fragments should help the classifier in identifying relationships between certain categories of video fragments and their temporal position within the video. For example, *introduction to the tutorial topic* and *closing of the tutorial* to the beginning and the end of the video, respectively.

Fragment Length, in seconds. Different fragment categories are likely to have different durations. For example, fragments related to the opening and closing of a tutorial are likely to be short, while fragments showing *code implementation* activities are likely to last longer, since they constitute the core of a tutorial.

Fragment Coverage, as percentage of the video tutorial covered by the fragment (*i.e.,* `fragment_length / video_length`). Although the coverage can be considered similar to the fragment length, it avoids possible issues related to video tutorials having a substantially different duration. Indeed, fragments extracted from long video tutorials are likely to be longer than those extracted from short video tutorials, despite their "type". This feature, being normalized on the video tutorial length, avoids this issue.

3.3.2 Structural Features

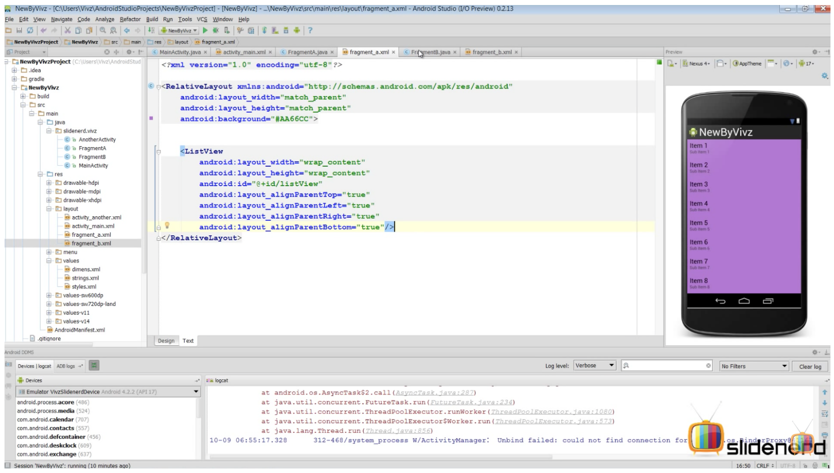
Another aspect to be considered is the structure of each frame composing a given video fragment. Specifically, different frames have different content of graphical elements, that translates into different features to be leveraged for video fragment identification. CODETUBE focuses on:

Average Pixel Overlap between all possible frame pairs in the fragment. The overlap is calculated pixel-wise. Only pixels in the same position and with the same color in two frames are considered overlapping. Fragments categorized as *theoretical concepts* are likely to have a higher percentage of overlapping pixels between frames, since the tutor may show the same slide for several seconds while discussing the concepts. Since we have discarded subsequent similar frames, we need to take this into consideration when computing such a feature. For example, assume that our video fragment V is composed of frames F_1, F_2, and F_3 and that we have discarded F_2 as being too similar to F_1. Also, suppose that the pixel overlap between F_2 and F_3 is 60%. We consider 100% of pixel overlap between F_1 and F_2, thus obtaining 80% as average pixel overlap for the fragment.

Average Number of Quadrilaterals, *i.e.,* the average number of quadrilaterals identified by shape detection analysis in the fragment's frames. Figure 7 shows an example of frame taken from a fragment tagged as *code implementation*.

There are several quadrilaterals corresponding to code editor, console output, package explorer, and the UI designer. The number of quadrilaterals is likely to discriminate fragments in which the IDE is shown (*e.g.,* a *code implementation* fragment) from the others. Also, the number of quadrilaterals on the screen also helps in discriminating *execution of the implemented code* from *code implementation*. Indeed, we expect the execution of the implemented code to open new windows (*e.g.,* the Android emulator) on the screen. In this case we adjust our computation to take into account the removal of (quasi-)identical frames. In this case, if our video fragment is composed by the frames F_1 (4 quadrilaterals), F_2 (4 quadrilaterals), and F_3 (1 quadrilateral) and F_2 has been

8. https://en.wikipedia.org/wiki/SubRip
9. http://snowball.tartarus.org



```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#AA66CC">


    <ListView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/listView"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

```
        at android.os.AsyncTask$2.call(AsyncTask.java:287)
        at java.util.concurrent.FutureTask.run(FutureTask.java:234)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1080)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:573)
        at java.lang.Thread.run(Thread.java:856)
10-09 06:55:17.328      312-468/system_process W/ActivityManager: Unbind failed: could not find connection for
```

Fig. 7. A frame taken from a *code implementation* fragment.

removed as too similar to F_1, we consider F_1 twice in the computation of the average: (4+4+1)/3=3. This approach is exploited in all features requiring the computation of the average between properties of the fragment's frames.

Average Largest Quadrilateral shown in the fragment's frames, expressed as a percentage of the total screen area. The goal is to understand if the IDE is the foreground of the frame. In Figure 7 the code editor is the largest quadrilateral and occupies about 32% of the frame space.

Having a high average coverage of the largest quadrilateral could imply having an IDE on the foreground and therefore helping the classifier in discriminating the categories involving development, *e.g.*, *code implementation*, *execution of the implemented code*, and *dealing with errors*. As for the previous structural features, we considered frames that were removed, and we replicate the value of their predecessor in the calculation of the average.

3.3.3 Code Features

Using the information extracted with the island parser, we compute the following features:

Average Constructs *i.e.*, the average number of code constructs found in the fragment's frames. The classifier could use this feature to discriminate between fragment categories likely to show a lot of code (*e.g.*, code implementation), and categories with less or no code (*e.g.*, topic introduction, working environment setup). We considered frames that were removed, by replicating the value of their predecessor.

Average Specific Node Types *i.e.*, the average number of occurrences of some specific AST nodes in the fragment's frames. In particular, we count *identifiers*, *imports*, *class declarations*, *method declarations*, *blocks*, *statements*, *stack traces*, *XML tags*, and *JSON constructs*. The idea is to differentiate the type of constructs for the different categories. For example, categories like *theoretical concepts*, *topic introduction*, or *closing* are unlikely to have complex constructs like declarations and statements. Fragments *dealing with common errors* are likely to contain more constructs related to stack traces. Instead, *JSON constructs* or *XML tags* could be shown in a frame when detailing some specific portions of the implementation (*e.g.*, JSON for illustrating access to remote services, or XML for Android app permissions or activity design). Also, *XML constructs* can be shown in the context of the *environment setup*. As for the previous structural features, we considered frames that were removed, and we replicate the value of their predecessor in the calculation of the average.

3.3.4 Semantic Features

The last set of features concerns the semantics of a fragment as captured by its textual content. Textual content relates to (i) the audio transcript, when available, and specifically to the transcript subset related to the segment time interval, and (ii) the text contained in the frames.

Considering the occurrences of each term as a feature for the machine learning algorithm would inevitably hinder the performance of any classifier, *e.g.*, by introducing problems related to synonymy or polysemy. A viable solution is to reduce the dimensionality of the semantic features by substituting terms with a set of topics extracted from the fragments. In our approach we use *Latent Dirichlet Allocation (LDA)* [18], an unsupervised topic modeling technique that, as suggested by Blei *et al.* [18], can be used as feature reduction approach for terms. We use the Stanford Topic Modeling Toolbox[10] configured to identify seven topics from the fragments corpus. The number of topics has been selected according to the number of labels resulted from the open coding session and reported in Table 2. Although a near-optimal configuration of LDA could require a proper setting—*e.g.*, through search-based optimization techniques [19]—in this work we have set the number of topics equal to the number of expected categories, an approach already followed when LDA has been used to categorize text [20]. Also, we adopted the default parameters for α=0.01, β=0.01, and n =1,000 (where n is the number of Gibbs iterations).

3.4 Classifying Video Fragments

The starting point for building a classifier able to discriminate between the different types of video fragments is a *training set* built from a collection of video fragments, their temporal, structural, code, and semantic features, and their respective category (*e.g.*, *code implementation*). While the computation of the video fragments' features is fully automated, the training

10. http://nlp.stanford.edu/software/tmt/

set labeling with categories is manual (*e.g.,* by relying on a process similar to the one presented in Section 2).

Once a training set is available, a supervised learning algorithm is run on it. Our approach uses the *Weka* [21] implementation of the Random Forest machine learning algorithm [22], which builds a collection of decision trees with the aim of solving classification-type problems, where the goal is to predict values of a categorical dependent variable from one or more continuous and/or categorical predictor variables.

The categorical dependent variable is represented by the video tutorial fragment category as identified in the open coding process (see Table 2), *e.g., code implementation, execution of the implemented code, etc.,* and we use the features described in Section 3.3 as predictor variables.

After experimenting with different machine learning algorithms, *e.g.,* J48, Bayesian Network, Logistic Regression, and Bagging classifiers (details of the comparison are in the replication package [23]), we selected the Random Forest algorithm. The built model can then be used to classify new video fragments. To do so, we extract the same set of features considered in the training set, and based on their values, the Random Forest is used to automatically determine the video fragment category.

Since some of the features we considered might correlate, we perform an *information gain* feature selection process [24] aimed at removing all features do not contributing to the information available for the prediction of the video fragment category. Also, when training the model we check the distribution of training set samples across the seven fragment categories. In case of imbalanced dataset, we apply a re-balancing technique, as it will be detailed in Section 4.1.

3.5 Tuning of CodeTube Parameters

The performance of CODETUBE depends on three parameters that need to be properly tuned:

- γ: the minimum percentage of LCS overlap between two frames to consider them as containing the same code fragment;
- λ: the minimum textual similarity between two fragments to merge them in a single fragment;
- μ: the minimum video fragment length.

To identify the most suitable configuration, one of the authors—who did not participate in the approach definition—built a "video fragment oracle" by manually partitioning a set of 10 video tutorials into cohesive video fragments[11]. Then, we looked for the CODETUBE parameters configuration best approximating the manually defined oracle. A challenge in this context is how to define the "closeness" of the automatically- and manually-generated video fragments.

Estimating Video Fragments Similarity. A video can be seen as a set of partitions (video fragments) of frames, where each frame belongs to only one partition, *i.e.,* the generated video fragments are clusters of frames. To compare the closeness of the video fragments generated by CODETUBE and those manually defined in the oracle, we used the MoJo effectiveness Measure (MoJoFM) [25], a normalized variant of the MoJo distance, computed as:

$$MoJoFM(A,B) = 100 - \left(\frac{mno(A,B)}{max(mno(\forall E_A, B))} \times 100\right)$$

where $mno(A,B)$ is the minimum number of *Move* or *Join* operations needed to transform a partition A into a partition B, and $max(mno(\forall\ E_A, B))$ is the maximum possible distance of any partition A from the partition B. Thus, $MoJoFM$ returns 0 if A is the farthest partition away from B, and returns 100 if A is exactly equal to B.

While MoJoFM is suitable to compare different partitions (video fragments) of the same elements (frames), we must take into account that video fragments are characterized by a constraint of sequentiality (*i.e.,* they can only contain subsequent frames). This could lead the MoJoFM to return high values (similarity) even when applied to two totally different video partitions. For example, consider the video frames $F = \{1,2,3,4,5,6\}$ and two sets of video fragments where the first set, $A = \{1,2,3,4,5,6\}$, contains a unique partition (video fragment) with all the elements (frames), and the second set, $B = \{\{1,2,3\},\{4,5,6\}\}$, contains 2 partitions of size 3. Since the MoJoFM is not a symmetric function, it would return $MoJoFM\,(A,B) = 25.0$ and $MoJoFM\,(B,A) = 80.0$, *i.e.,* two different values, despite the fact that the two partitions are the same.

Keeping a one-way comparison between the oracle and the obtained video fragments undermines the tuning phase. To avoid this, yet keeping a margin of approximation, both sides of the MoJoFM should be taken into account. Two sets of fragments will tend to have the same value if they are close in their partitioning. For this reason, we calculate the similarity in both directions and compute their mean value:

$$closeness(A,B) = \frac{MoJoFM(A,B) + MoJoFM(B,A)}{2}$$

In doing so, spikes of high values for the *MoJoFM* between two sets of video fragments for one direction are lowered or preserved depending on the opposite.

Estimating the Most Suitable Parameter Configuration. For each parameter, we identified a set of possible values. Table 6 shows the intervals we adopted, and the step (Δ) used whenever a new combination is generated.

TABLE 6
Parameter tuning intervals.

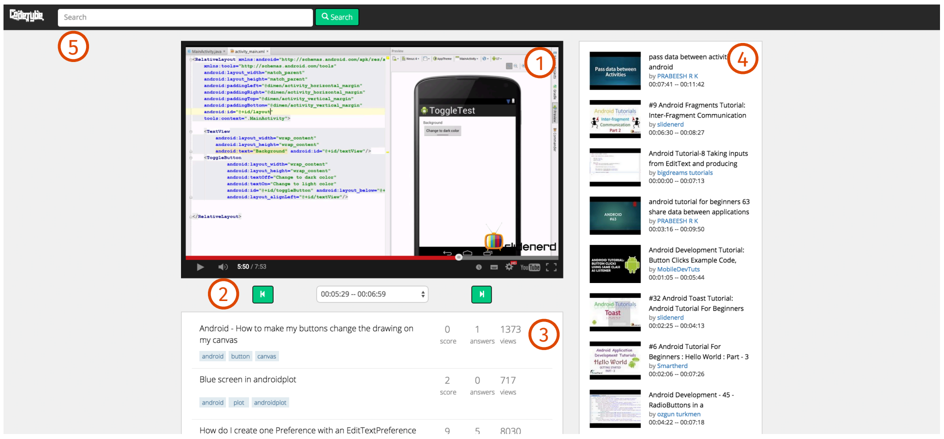
Parameter	Min	Max	Δ
γ	5%	50%	5%
λ	10%	80%	5%
μ	1,000s	120,000s	10,000s

In total, we experimented 1,800 different parameter combinations, adopting the one with the top ranked MoJoFM (γ =5%, λ =15%, μ =50s) for the full-fledged analysis phase.

3.6 Integrating Other Sources of Information

CODETUBE can be enriched by mining other online resources, as our goal [26] is to offer a *holistic* point of view, also because we argue that no single type of resource can offer

11. These 10 videos are not part of the 136 considered in the study presented in Section 2.



00:05:29 -- 00:06:59

Android - How to make my buttons change the drawing on my canvas — 0 score, 1 answers, 1373 views — android, button, canvas

Blue screen in androidplot — 2 score, 0 answers, 717 views — android, plot, androidplot

How do I create one Preference with an EditTextPreference — 9, 5, 8030

pass data between activit android — by PRABEESH R K — 00:07:41 -- 00:11:42

#9 Android Fragments Tutorial: Inter-Fragment Communication — by slidenerd — 00:06:30 -- 00:08:27

Android Tutorial-8 Taking inputs from EditText and producing — by bigdreams tutorials — 00:00:00 -- 00:07:13

android tutorial for beginners 63 share data between applications — by PRABEESH R K — 00:03:16 -- 00:09:50

Android Development Tutorial: Button Clicks Example Code, — by MobileDevTuts — 00:01:05 -- 00:05:44

#32 Android Toast Tutorial: Android Tutorial For Beginners — by slidenerd — 00:02:25 -- 00:04:13

#6 Android Tutorial For Beginners : Hello World : Part - 3 — by Smartherd — 00:02:06 -- 00:07:26

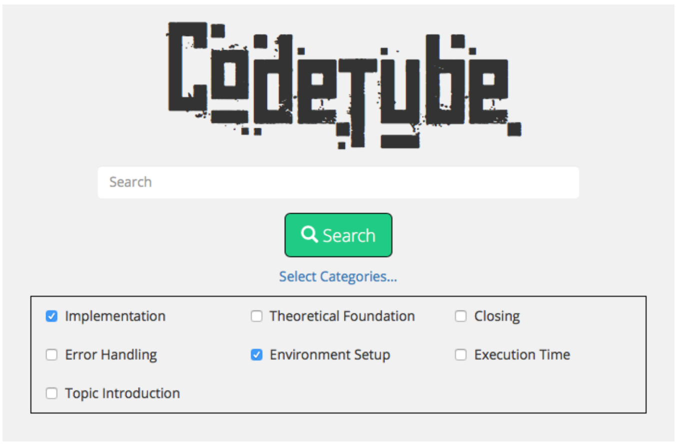
Android Development - 45 - RadioButtons in a — by ozgun turkmen — 00:04:22 -- 00:07:18

Fig. 8. The CODETUBE user interface: (1) video fragment, (2) fragment controls, (3) related Stack Overflow discussions, (4) related video fragments, (5) search box.

exhaustive assistance. As an initial contribution, we added as an additional online information source the Stack Overflow data dump. We mined and extracted discussions related to the topics of the extracted video tutorials, pre-processed them to reduce the noise, and made them available to CODETUBE.

The last step in the data pre-processing of CODETUBE consists in indexing both the extracted video fragments and the Stack Overflow discussions.

For this step we use LUCENE[12]. Each video fragment is considered as a document, while for each Stack Overflow discussion we separately index each question and answer. The text pre-processing phase is identical to the one explained in Section 3.2. The text indexed for a video fragment is represented by the terms contained in its frames and audio transcripts. The text indexed for the Stack Overflow post is represented by the terms they contain.



[✓ Implementation] Theoretical Foundation Closing
Error Handling [✓ Environment Setup] Execution Time
Topic Introduction

Fig. 9. CODETUBE: Search page with filtering.

3.7 The CodeTube User Interface

The main user interface of CODETUBE is depicted in Figure 8. CODETUBE provides a service that allows users to search, watch, and navigate the different fragments of a video tutorial. The user can input a textual query and select via checkboxes the type of fragment categories she is interested in (see Figure 9).

CODETUBE will provide a list of relevant video tutorial fragments (search results) from which the user can select the one she is interested in watching. When a video fragment is selected for watching from the search results, the GUI depicted in Figure 8 is shown.

CODETUBE uses the YouTube player (1) provided by the YouTube API[13]. The video starts at the time devised by the selected fragment. CODETUBE provides an additional controller (2) to visualize the timestamps of the fragments identified by our approach, select a specific fragment, or move to the next/previous fragment. During the playback, the selector underneath the video player keeps the pace of the video timing and shows the current fragment. When a new fragment is reached, or the user jumps to it, CODETUBE automatically extracts a query from the text contained in the fragment (*i.e.*, transcripts and OCR output of the frames it contains), queries both the index of Stack Overflow and of the video fragments, and updates the related discussions (3) and the suggested YouTube video fragments (4). A search bar (5) is always available to the user to run new queries.

12. https://lucene.apache.org/
13. https://developers.google.com/youtube/js_api_reference

4 Study I: Identification and Classification of Video Tutorial Fragments

The *goal* of this study is to evaluate CODETUBE with the *purpose* of determining its ability to (i) extract meaningful video tutorial fragments, and (ii) correctly classify them, using the categories identified in Section 2.

We cast such an evaluation into two research questions:

RQ$_1$: *To what extent do automatically and manually extracted video tutorial fragments overlap?*

RQ$_2$: *How accurate is* CODETUBE *in classifying video tutorial fragments in the considered categories?*

The first research question assesses the overlap between the video tutorial fragments automatically extracted by

CODETUBE and those manually identified by people watching the same tutorial. A high overlap indicates the ability of CODETUBE to split video tutorials as humans would do.

The second question (i) assesses the accuracy of our technique in classifying video tutorial fragments in the seven categories listed in Table 2, and (ii) investigates the relevance of the different categories of features (*i.e.,* temporal, structural, code, and semantic) described in Section 3.

4.1 Study design and procedure

We use the dataset of video tutorial fragments described in Section 2, consisting of 748 manually tagged fragments taken from 136 video tutorials related to Java programming, JSPs/Servlets, and Android development, distributed across the seven categories of video fragments. We run CODETUBE on this dataset to identify and categorize video fragments.

To answer **RQ**$_1$, we computed the MoJoFM [25] between the fragments extracted by CODETUBE and the ones manually identified by the study participants, by essentially applying the same procedure adopted for the parameters' tuning (see Section 3.5). We show boxplots of the MoJoFM achieved by CODETUBE over the 136 subject video tutorials.

To answer **RQ**$_2$, we performed a 10-fold cross validation, computing the overall average accuracy of the model when (i) only relying on temporal, structural, code, and semantic features in isolation, (ii) combining the four categories of features in pairs (six possible pairs) and in groups of three (four possible groups), and (iv) considering all of them.

We performed a feature selection process using the *information gain* feature selection technique [24] before running the 10-fold cross-validation. Table 7 shows selected and discarded features (and their rank as provided by the *information gain*) when considering the complete dataset.

TABLE 7
Feature selection results.

Feature Type	Name	Selected	
Temporal	Beginning Time	✓	1
	Fragment Length	✓	2
	Fragment Coverage	✓	4
Structural	Average Pixel Overlap	✓	3
	Average Number of Rectangles	✓	8
	Average Largest Rectangle	✓	7
Code	Average Constructs	✓	9
	#identifiers	✓	15
	#class declarations	✓	17
	#method declarations	✓	19
	#blocks	✓	13
	#statements	✓	10
	#imports	✓	20
	#stack traces	✗	–
	#JSON constructs	✗	–
	#XML tags	✗	–
Semantic	topic 1	✓	16
	topic 2	✓	5
	topic 3	✓	14
	topic 4	✓	18
	topic 5	✓	12
	topic 6	✓	11
	topic 7	✓	6

The temporal feature is top ranked as some fragments occur in specific time frames of the video. Some features that seem to be related (*e.g.,* fragment length and coverage) are both taken into account and ranked in the top position, hence indicating that they bring complementary information. Instead, data-specific features (*e.g.,* JSON constructs and XML tags) do not bring information for discriminating the considered categories. The same happens for stack traces, that could have been potentially useful for discerning error scenarios. One possible interpretation is that the OCR failed to successfully capture stack traces, *e.g.,* because the console output is not fully visible in the IDE.

Since our dataset is strongly unbalanced (see Table 2), we balanced the training set at each iteration (*i.e.,* for each of the ten folds) by exploiting the Synthetic Minority Oversampling TEchnique (SMOTE) [27]. SMOTE re-balances the training set by creating artificial instances obtained by joining nearest neighbors of the minority class instances. While we balanced the training set to build the classifier, the test set was never modified to avoid any bias.

We assess the overall performances of the model with its average accuracy. Also, we dig into the results by presenting (i) the obtained confusion matrix, (ii) the model accuracy for each of the seven considered fragment categories, and (iii) the Area Under the ROC curve (AUROC) [28] obtained for each category as well as for the overall model. An AUROC of 0.5 indicates a model having the same prediction accuracy in identifying true positives as a random classifier. A perfect model (*i.e.,* zero false positives and zero false negatives) has instead AUROC=1.0. Thus, the closer the AUROC to 1.0, the higher the model performances.

4.2 Study results

4.2.1 RQ$_1$: To what extent do automatically and manually extracted video tutorial fragments overlap?

Figure 10 shows (i) the boxplots of the MoJoFM obtained when comparing the video fragments automatically extracted by CODETUBE with those manually defined by watching the 136 video tutorials subject of this study, and (ii) the scatterplot of the MoJoFM and video length, useful to investigate possible relationships between the accuracy of CODETUBE in identifying video fragments and the video length.

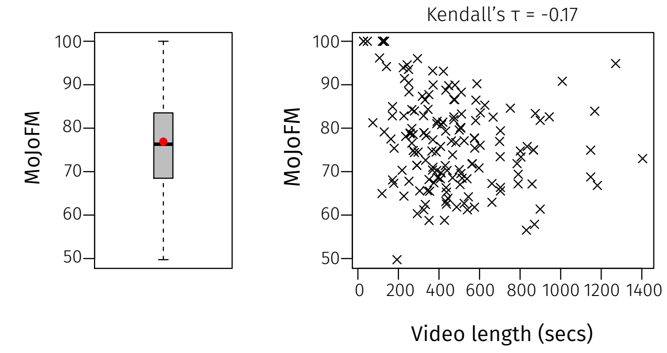


Fig. 10. RQ$_1$: MoJoFM achieved on the 136 video tutorials and scatterplot between MoJoFM and video length

On average, CODETUBE achieves 77% of MoJoFM (median=76%), suggesting a high similarity between manually- and automatically-identified video fragments. In seven cases, the video partition proposed by CODETUBE is exactly the same manually defined by participants and, for twenty videos, the MoJoFM is higher than 90%. Clearly, there are cases in which CODETUBE fails in identifying meaningful fragments, as it happens for the five video tutorials in which it achieves a MoJoFM lower than 60%. We looked into these cases to understand the reasons behind such a low performance of CODETUBE. We identified three main situations in which CODETUBE clearly exhibits limitations in identifying meaningful video fragments:

1) *Very low video quality*. Low quality videos[14] make it difficult to extract meaningful text with the OCR, substantially limiting the information available to the CODETUBE*Video Fragments Analyzer*.
2) *Zooming on the screen*. In some videos the tutor zooms in and out the screencast[15] , thus making it challenging to identify video fragments. Indeed, zooming at different levels in different frames limits the effectiveness of some of the heuristics adopted by the *Video Fragments Analyzer* (*e.g.,* the LCS).
3) *Continuous shifting of the portion of the screen captured in the screencast.* In some tutorials the screencast does not capture the whole screen, but just the portion of the screen surrounding the mouse pointer. This results in a continuous shifting of the part of the screen shown in the tutorial[16].

The scatterplot in the right part of Figure 10 does not show any strong relationship between the MoJoFM and the length (in seconds) of the video tutorials. Also the Kendall's τ coefficient (-0.17) confirms that while CODETUBE is slightly more precise in the identification of video fragments on shorter videos, there is a very weak negative correlation between MoJoFM and video length.

4.2.2 RQ$_2$: *How accurate is* CODETUBE *in classifying video tutorial fragments in the considered categories?*

Table 8 reports the accuracy (*i.e.,* percentage of correctly classified instances) and AUROC obtained by our approach when relying on different sets of features.

TABLE 8
RQ$_2$: Performances when using different combinations of features.

Considered set of features	Accuracy	AUROC
Temporal	56%	0.82
Structural	30%	0.65
Code	41%	0.68
Semantic	40%	0.70
Temporal+Structural	61%	0.85
Temporal+Code	67%	0.88
Temporal+Semantic	66%	0.84
Structural+Code	46%	0.74
Structural+Semantic	48%	0.75
Code+Semantic	45%	0.75
Temporal+Structural+Code	68%	0.88
Temporal+Structural+Semantic	68%	0.88
Temporal+Semantic+Code	70%	0.90
Structural+Semantic+Code	52%	0.79
Temporal+Structural+Semantic+Code	**72%**	**0.92**

When exploiting the temporal, structural, code, and semantic features in isolation, the best performances are provided by the temporal features, with 56% of accuracy and AUROC=0.82. While this result might look surprising, temporal features can be very effective in identifying at least two of the categories considered in our study: the *introduction of the tutorial topic* and the *closing of the tutorial*.

Indeed, 85% of the tutorials start with an introduction to the tutorial topic and 35% end with a closing part generally featuring a summary of the tutorial and/or information about future tutorials that will be published. Thus, it is quite simple for the model to learn how to spot out these types of fragments by exploiting temporal features—*e.g.,*`if` *beginning time*[17] < 0.1 `then` fragment type `is` *introduction of the tutorial topic*. Temporal features also help in identifying fragments dealing with *code implementation* activities. For this fragment type, our approach learns, by exploiting temporal features, that if a fragment starts during the first half of the tutorial (*i.e., beginning time* < 0.58) and it lasts for over 40% of the overall tutorial length (*i.e., fragment coverage* > 0.42) it likely represents a *code implementation* fragment. This makes sense since code implementation often represents the bulk of software development video tutorials.

The other sets of features (*i.e.,* structural, code, and semantic) obtain substantially lower performances than temporal features when used in isolation. However, for some specific categories of fragments, they perform as well as or even better than the temporal features.

Semantic features help in characterizing fragments describing how to *deal with common errors*. This is possible thanks to a specific LDA topic (*topic* 2) described by key words such as *exception*, *try*, and *throw*. This topic is exploited by our approach in the identification of video fragments explaining how to deal with common errors. However, this is not enough for our technique to provide a high accuracy in the identification of this type of fragments. This is mainly due to the fact that *topic 2* plays a major role in tutorial fragments dealing with implementation activities, thus leading to a high number of mis-classifications.

Code features are the best ones in identifying *code implementation* fragments: our approach learns that a high number of code snippets shown on the screen for the whole fragment duration likely indicates its focus on implementation tasks. Finally, structural features provide the lowest accuracy and, when used in isolation, exhibit very low accuracy in the identification of all categories of video fragments.

When combining the four sets of features in pairs, performances are boosted up to 67% of accuracy and 0.88 of AUROC (obtained when combining temporal and code features), indicating quite good performances of the built model. Again, it is clear the major role played by the temporal features. Indeed, the three models exploiting them have AUROC$\geq$0.84 as compared to the 0.74 and 0.75 obtained in the two models do not exploiting temporal features. The accuracy further increases up to 70% (AUROC=0.90) when using three groups of features at a time (temporal, semantic, and code features), reaching its maximum value (72%) when all four sets of features are exploited. In this case, the built model exhibits a quite high AUROC of 0.92.

14. For example, https://www.youtube.com/watch?v=-VRUX-iSPWc
15. For example, https://www.youtube.com/watch?v=xuX96Lik3Co
16. For example https://www.youtube.com/watch?v=-L8FAKadrhg
17. Beginning time expresses the relative beginning of the video fragment as percentage of the video tutorial length.

TABLE 9
RQ$_2$: Confusion Matrix and AUROC per each Category when Using all Features.

	Predicted Category							
Reference Category	CT	DE	ES	EIC	CI	TC	ITT	AUROC
Closing of the Tutorial (CT)	**37**	0	0	5	3	0	0	0.98
Dealing with Errors (DE)	0	**7**	0	0	8	3	1	0.88
Environment Setup (ES)	1	0	**19**	0	7	4	7	0.88
Execution Implemented Code (EIC)	11	2	3	**83**	17	6	1	0.91
Code Implementation (CI)	7	8	18	26	**200**	20	6	0.89
Theoretical Concepts (TC)	1	4	0	6	14	**55**	7	0.90
Introduction tutorial topic (ITT)	0	1	3	0	2	6	**131**	0.98

Table 9 reports the confusion matrix obtained by this comprehensive model, showing as well the AUROC for each of the seven categories of fragments.

Our approach is very effective in identifying fragments related to the *introduction to the tutorial topic* (accuracy=82%, AUROC=0.98) and to the *closing of the tutorial* (accuracy=92%, AUROC=0.98). This very high AUROC is possible because the temporal features are effective to discriminate video fragments of these two categories. Classification performances are also very good for fragments dealing with the *execution of the implemented code* (accuracy=67%, AUROC=0.91), the explanation of *theoretical concepts* (accuracy=63%, AUROC=0.90), and *code implementation* activities (accuracy=70%, AUROC=0.89). Concerning the former (*e.g.,* the execution of an implemented app in the Android emulator), we expected structural features to be highly discriminating, because the execution of the code would have opened a new window, resulting in more quadrilaterals shown on the screen.

However, these features resulted to be useless for the identification of these fragments. This is due to the fact that often the implemented code is executed directly inside the IDE's console (always shown on the screen) without opening a new window. This makes it difficult to discern this situation from a code implementation with no execution activity. Currently, our approach identifies these fragments as "short implementation activities" (*i.e.,* they are characterized exactly as code implementation fragments, but they are much shorter). Probably, other features should be thought to increase the classification accuracy for this category of fragments.

An effective identification of *code implementation* fragments is possible with a combination of temporal and code features. For example, one of the rules used in a decision tree generated by our approach identifies implementation fragments as those lasting at least 42% of the overall video length, starting during the first half of the tutorial, and containing at least one code statement and one block statement.

Finally, while still being acceptable, performances decrease when categorizing fragments related to the *environment setup* (accuracy=50%, AUROC=0.88) and to *common errors* one could encounter during implementation activities (accuracy=37%, AUROC=0.88). In these cases, we expected semantic features to help in the classification of these fragments. As previously mentioned, semantic features only partially help in the identification of fragments related to implementation errors. A deeper investigation revealed that the limited contribution of the semantic features is due to the high imprecision of the OCR in extracting terms from the video frames. As previously explained, OCR tools are usually designed to deal with text on white background (*i.e.,* paper documents). When using an OCR tool on video frames, the high variability of the background can result in a high amount of noise. Very likely, the accuracy of our approach could strongly benefit from the implementation of more robust OCR tools designed to deal with such a noise.

5 Study II: Intrinsic evaluation with users

In our previous study we assessed the accuracy of CODETUBE in (i) identifying meaningful video fragments, and (ii) correctly classifying video fragments in the seven considered categories. In this study we dig deeper in the quality of the extracted video fragments, looking at their cohesiveness, self-containment, and relevance to a query as perceived by developers. Also, we assess the relevance and complementarity of the video fragments to the Stack Overflow discussions recommended by CODETUBE.

The three research questions of this study are:

RQ$_3$: *To what extent are the extracted video tutorial fragments cohesive and self-contained?*

RQ$_4$: *To what extent are the Stack Overflow discussions identified by CodeTube relevant and complementary to the linked video fragments?*

RQ$_5$: *To what extent is CodeTube able to return results relevant to a textual query?*

RQ$_3$ aims at assessing the capability of CODETUBE to extract fragments that are on the one hand cohesive—*i.e.,* related to a very specific (sub)topic of the tutorial—and on the other hand self-contained, *i.e.,* they can be understood without watching the rest of the video.

The purpose of RQ$_4$ is to assess the capability of CODETUBE to link video tutorial fragments to relevant Stack Overflow discussions. While specific approaches to recommend Stack Overflow discussions exist [6], our aim is to determine whether the textual content of the video tutorial fragment can be used to retrieve such discussions. Also, to determine the usefulness of a multi-source recommender like CODETUBE, we are interested to understand whether the Stack Overflow discussions provide complementary information with respect to the video tutorial.

RQ$_5$ assesses CODETUBE's retrieval capabilities over the indexed video fragments, to determine whether the indexed textual corpus allows to find relevant video fragments, and whether such fragments are at least as relevant as those returned by YouTube with the same query.

The *context* of the study consists of *participants* and *objects*. The *participants* have been identified using convenience sampling among personal contacts of the authors, and by sending invitations over mailing lists for open-source developers. In total, 40 participants completed the survey. The *objects* of the study are a set of 4,747[18] video tutorials about Android development indexed in CODETUBE. From these video tutorials, CODETUBE extracted a total of 38,783 fragments. Note that in this study we chose to focus on video tutorials dealing with a specific technology (*i.e.,* Android) and we only involved participants having experience with such a technology. This was a constraint to ensure a good assessment of the video tutorial fragments and Stack Overflow discussions identified by CODETUBE.

5.1 Study design and procedure

The study has been conducted using an online survey questionnaire, through which we asked questions to the potential respondents to assess the results of CODETUBE. The questionnaire is composed of two sections, preceded by preliminary assessment of the primary activity (industrial/open source developer, student, academic), programming experience, and specific experience about Android development of respondents. This preliminary section also included exploratory questions aimed at understanding (i) how often and in which circumstances respondents use video tutorials and Q&A Websites, (ii) whether they found useful information there, and (iii) how they react to video tutorials being too long (*e.g.,* scroll it, watch it anyway, or give up). We also asked participants what the main aspects of strength and weakness of video tutorials are, compared to standard documentation and Q&A Websites.

The first section shows to respondents a video tutorial from YouTube together with its video fragments extracted by CODETUBE. Each video shown to respondents is randomly selected from the set of videos composed by exactly three fragments in the CODETUBE dataset. For each video fragment we ask (**RQ**$_3$) whether the fragment is cohesive and self-contained. We also show the top-three relevant Stack Overflow posts, and ask (**RQ**$_4$) to what extent they are relevant and complementary to the video tutorial fragments. For each respondent, this section is repeated for two video tutorials randomly chosen from a sample of 20 video tutorials randomly selected from the 4,747.

TABLE 10
Queries formulated by graduate students.

1	How to send logs to server android
2	How to initiate an activity as a background service in Android
3	How to display multiple items in listview android
4	Material ui examples android
5	How to animate a transition between fragments android
6	How to access accelerometer data on Android device
7	How to access external file system android
8	Stop background services android
9	How to modify the layout of the UI in android
10	Android GUI thread

The second section aims to assess the relevance of the top-three returned video fragments to a given query (**RQ**$_5$). As a baseline for comparison, we evaluate the relevance of the top-three videos returned by YouTube using the same query. The query shown to each respondent is sampled from a set of 10 queries (see Table 10) formulated by graduate students at Florida State University, having a long experience in Android development. The queries involve typical Android tasks, *e.g.,* sending logs to servers, initiate activities in background, animate transitions, access accelerometer data, stopping background services, or modifying the UI layout.

The queries are generic, and YouTube is likely able to return as relevant results as CODETUBE. Only specific queries, referring to code elements—not contained in YouTube metadata—would show the advanced of the indexing capabilities of CODETUBE. Instead, we are interested in showing that, for the typical queries a developer formulates, CODETUBE returns at least as relevant as YouTube, but consisting in shorter, cohesive and self-contained fragments.

Finally, after the second section, we asked the respondents to evaluate, through an open comment, the main points of strength and weakness of CODETUBE.

All the assessment-related questions follow a three-level Likert scale [30], *e.g.,* "very cohesive", "somewhat cohesive", and "not cohesive". We limit the number of video fragments, Q&A discussions and queries for each respondent to avoid the questionnaire being too long. Before sending the questionnaire to perspective respondents, we ran a pilot study to assess its estimated duration, which resulted to be between 25 and 40 minutes.

The questionnaire was then uploaded on the QUALTRICS[19] online survey platform, and a link to the questionnaire was sent via email to the invitees. We made it clear that anonymity of participants was protected and data were only published in aggregate form. The *Qualtrics* survey platform allowed us to achieve randomization and balancing, by automatically selecting video tutorials (with related Stack Overflow discussion) and queries to be evaluated by each respondent. After sending out the invitation, invitees had two weeks to respond.

5.2 Study results

Out of the 40 study participants, 6 declared to have no experience in Android development. Since the video tutorials considered in the study were not introductory but related to specific Android topics, we excluded their answers. We collected a total of 180 video tutorial fragment evaluations (with respect to their cohesiveness and self-containment), 540 Stack Overflow discussion evaluations, and 90 video tutorial fragment evaluations with respect to a query. Each fragment and SO discussion received 3-5 evaluations, except for 3 videos and 2 queries, that, due to the exclusion of some participants motivated above, received less than 3 evaluations. These videos and queries were excluded from the analysis.

Table 11 shows the demographics of the participants involved in the survey. The population who completed our survey is composed of 70.6% of professional and open source developers, 17.6% of master students, and 11.8% of PhD students. 32.3% of the population has more than 10 years experience, 17.5% has between 5 and 10 years, 38.3% between

18. These numbers refer to a previous publication [29]. The current numbers are reported in Section 1.

19. https://az1.qualtrics.com

TABLE 11
Participants' Occupation, Experience in Java, and Usage of Video Tutorials for Study II.

Occupation	Total	%	Experience in Java	Total	%	Usage of Video Tutorials	Total	%
Faculty	0	0%	Less than 1 year	0	0%	Daily	1	3%
PhD Student	4	11.8%	1-3 years	4	11.8%	Few times a week	13	38.2%
Master Student	6	17.6%	3-5 years	13	38.3%	Few times a month	12	35.3%
Undergraduate Student	0	0%	5-10 years	6	17.5%	Rarely	8	23.5%
Professional Software Developer	24	70.6%	More than 10 years	11	32.3%	Never	0	0%
Total	34	100%	**Total**	34	100%	**Total**	34	100%

3 and 5 years, 11.8% between 1 and 3 years. When asked about Android programming experience, the majority (38.3%) declared less than 1 year of experience, followed up by 23.5% of respondents with more than 3 years experience, 20.5% between 2 and 3 years, and 17.6% between 1 and 2 years of experience.

The participants use video tutorials either on a weekly (38.2%) or monthly (35.3%) basis. 3% declared to use video tutorials on a daily basis; nobody declared to never use them. Video tutorials are unlikely to help bug/error fixing (5%), but are the primary means to learn new concepts (43%).

When asked to provide open comments on the weaknesses and strengths of video tutorials, respondents pointed out different aspects. The primary point of strength is the step-by-step nature of a video. One respondent wrote *"As opposed to Q&A Websites, video tutorials describe a complete process step-by-step. The visualized flow of actions is particularly useful in setting up working environments"*, another emphasized the *"possibility to see the complete interaction of the developer with the IDE"* and *"how a specific library is imported before it is used in the code. This does not hold when you simply copy and paste code from Websites"*. Another positive point concerns the guidance given by a tutor. One respondent reported that *"there is a 'real' person talking with you, so it is easy to learn new concepts"*, while another respondent emphasized the fact that *"you can see what the tutor does"*. The primary weakness identified by respondents concerns time. When a video tutorial is too long, respondents said they would either try to scroll it to seek the relevant information (47%), or give up to find alternative sources (53%). Nobody opted for the third option, *i.e.,* watching the whole video anyway. Respondents generally consider videos too long and slow and not suited when *"you need to quickly solve a problem"*, or *"just a small piece of information"*. One respondent reported how *"due to time constraints I cannot always watch the entire tutorial"*. The lack of searching and indexing functionalities of the contents of a video is also considered a weakness. One of the respondents claimed that *"browsing is not easy, unless the video has an index to navigate through the concepts/sections in the video"*, while another highlighted how *"searching for a particular piece of information in the whole video is much harder than doing the same in a text document"*.

5.2.1 RQ$_3$: To what extent are the extracted video tutorial fragments cohesive and self-contained?

Figure 11 synthesizes the results of our study for RQ$_3$.

More specifically, the top of Figure 11 shows a diverging stacked bar chart reporting the distribution of responses concerning the perceived fragment cohesiveness score. 129 responses (64%) indicated fragments as *very cohesive*, 56 (28%) as *somewhat cohesive*, and 16 (8%) as not cohesive.

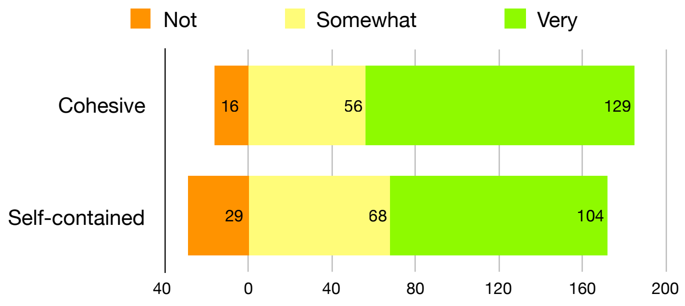


Fig. 11. RQ$_3$: Distribution of cohesiveness and self-containment.

The bottom of Figure 11 shows the distribution of the perceived self-containment score. In this case, 104 (52%) fragments were evaluated as *very self-contained*, 68 (34%) as *somewhat self-contained* and 29 (14%) as *not self-contained*. The lower scores achieved for self-contained are not surprising, because obtaining self-contained fragments—and hence understandable without watching the rest of the video—is more challenging than achieving a high cohesiveness.

5.2.2 RQ$_4$: To what extent are the Stack Overflow discussions identified by CodeTube relevant and complementary to the linked video fragments?

Figure 12 synthesizes the results of our study for RQ$_4$.

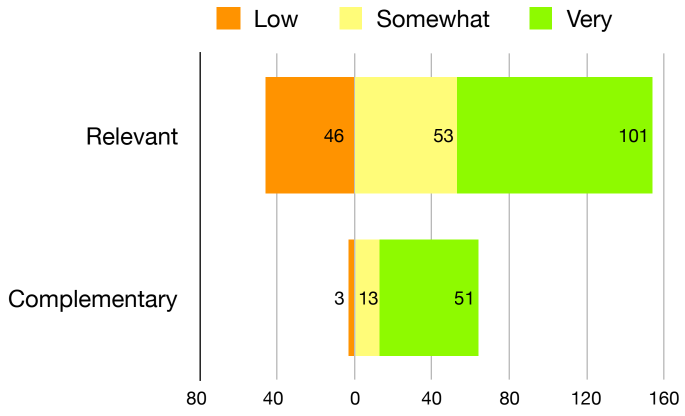


Fig. 12. RQ$_4$: Relevance of Stack Overflow discussions to video fragments, and complementariness to videos.

The top of Figure 12 shows the distribution of the perceived relevance of the Stack Overflow discussions associated to the video fragments of each video tutorial considered in the study. 101 (50.5%) of the provided response indicated a *very high relevance*, 53 (26.5%) a *somewhat high relevance*, and 46 (23%) no relevance.

The bottom of Figure 12 shows results related to the complementariness. In 51 cases (76%) the Stack Overflow discussions were evaluated as *very complementary*, in 13 cases (19%) as *somewhat complementary*, and in 3 (4%) case as having a *low complementariness*. In summary, results indicate that, while respondents only considered the retrieved discussions fairly relevant to the fragments from where the queries were generated, they almost totally agreed about the complementarity of the provided information. We believe that video tutorials have a different purpose than Stack Overflow discussions. The former have an introductory, step-by-step guide to a given problem, the latter discuss a specific problem/answering a specific questions.

5.2.3 RQ$_5$: *To what extent is CodeTube able to return results relevant to a textual query?*

We asked participants to evaluate the top-three results that CODETUBE and YouTube retrieved for a set of 10 queries (see Table 10). Each participant evaluated the relevance of a result with respect to the query by following a three-level Likert scale [30], *i.e.,* "very related", "somewhat related", and "not related". We use the Normalized Discounted Cumulative Gain (NDCG) [31] to aggregate the results.

Similarly to what done for the other research questions, queries with less than 3 replies are ignored. The NDCG is thus calculated on a set of 8 queries out of the initial 10. We obtained $NDCG_{CT}(Q,3) = 0.67$ and $NDCG_{YT}(Q,3) = 0.63$ for CODETUBE and YouTube, respectively.

Even if CODETUBE seems to perform slightly better than YouTube, a statistical analysis of the $NDCG_{YT}$ and $NDCG_{CT}$ distributions, performed using the Wilcoxon signed-rank (paired) test, did not show the presence of a statistically significant difference (p-value=0.49). Even though the data collected is not enough to draft any statistically significant conclusion, there are some considerations to make. First, when extracting the top-three results from YouTube we removed all the retrieved videos that are not included in the CODETUBE dataset. This makes the comparison unfair for our approach. Second, YouTube recommends entire videos, while CODETUBE recommends specific fragments. Thus, our approach is potentially more focused even if both the fragment and the whole video recommended by YouTube are equally relevant.

5.2.4 *Strengths and Suggestions for Improvement*

We also asked participants to freely comment about CODETUBE. They had a positive impression. One respondent reported *"CodeTube looks very useful, added to the bookmarks!"*, while another wrote *"excellent work, [..] the idea behind CodeTube is brilliant"*. The extraction of fragments has been appreciated and considered *"very useful for developers who are already knowledgeable about the topic, they can save a lot of time"*. The possibility of having complementary sources of information, *e.g.,* Stack Overflow, was appreciated. One respondent reported *"the concept is amazing, and has a lot of possibility of improvement, given the huge amount of different sources of data available"*, while other participants asked for additional features. One participant asked for *"the possibility to search for SO discussions directly below the video"*, while another wondered that *"it would be nice if the tool can provide a summary/description that describes the context"*.

6 STUDY III: EXTRINSIC EVALUATION

A successful technological transfer is the main target objective for each prototype tool. Thus, the *goal* of this second study is to extrinsically investigate CODETUBE's industrial applicability. Specifically, the research question we aim at answering with this evaluation is:

RQ$_6$: *Would* CODETUBE *be useful for practitioners?*

The *context* of the study is represented by three leading developers—all with more than five years of experience in app development—of three Italian software companies, namely Next, IdeaSoftware, and Genialapps. Although this is a fairly limited context, the main purpose of this study is to collect feedback about the possibility of adopting CODETUBE in realistic development scenarios, as well as suggestions for its improvement.

6.1 Study design and procedure

We conducted semi-structured interviews to get quantitative and qualitative feedback on CODETUBE. Each interview lasted two hours. During the interview we let developers explore CODETUBE for about 90 minutes, searching for video tutorials on specific technology or to fix problems.

Each interview was based on the think-aloud strategy. We also explicitly asked the following questions: (1) Do you use video tutorials during development tasks? (2) Would the extraction of shorter fragments make you more productive? (3) Is the multi-source nature of CODETUBE useful? (4) Are you willing to use CODETUBE in your company?

Participants answered each question using a 4-point scale: absolutely no, no, yes, and absolutely yes. We avoided the intermediate, neutral level, as we wanted participants to take a clear position. The interviews were conducted by one of the authors, who annotated the answers as well as additional insights about the strengths and weaknesses of CODETUBE that emerged during the interviews.

6.2 Study results

Nicola Noviello, Project Manager @ Next. Nicola positively answered to our first three questions (*i.e.,* "absolutely yes"). Nicola declared to use video tutorials daily; *"they are particularly useful for senior and junior developers for both learning a new technology or finding the solution to a given problem. I see very often my developers on specialized YouTube channels searching for and watching video tutorials."* Nicola also appreciated the multi-source nature of CODETUBE; *"the video tutorial provides the general idea on the technology, while Stack Overflow discussions are particularly useful to manage alternative usage scenarios and specific issues."*. Regarding the extraction of fragments, Nicola commented that *"I usually discard video tutorials that are too long, because when I try to scroll/fast forward it to manually locate segments of interest, I am generally not able to find what I need. I strongly believe that the relevant segment is there but randomly scrolling a video tutorial is not worthwhile! I prefer to look for more focused video tutorials. Also, the possibility to filter video fragments on the basis of their category is a fantastic feature!"*. Nicola then confirmed that the availability of shorter fragments would make him much more productive.

Nicola answered "yes" to the question related to the usefulness of CODETUBE; *"I did not answer absolutely yes* *because of the limited number of indexed tutorials. However, I strongly believe that the tool has an enormous potential.”*. Nicola declared that he will present the tool to a newcomer trainee to quantify to what extent the tool is useful for developers that have a little knowledge on the Android world; *“I usually suggest to trainees to look for and watch video tutorials but very often they are not able to find the right information. I would like to see whether* CODETUBE *is able to mitigate such a problem.”*.

Luciano Cutone, Project Manager @ IdeaSoftware. Luciano positively answered to our first three questions; *“I love video tutorials but several times they are too long and I do not have enough time to watch whole videos. Thus, I have to scroll the video hoping to identify relevant segments. This takes time and makes video tutorials less effective. With* CODETUBE *life will be easier!”* Luciano particularly appreciated also the possibility to filter video fragments on the basis of their category. He also suggested an interesting new features:*It could be nice to give the possibility to the user to change the classification of the video tutorials. In this way it is possible to correct possibly misclassification and improve the classification accuracy of the tool. Of course, a moderator is required to accept the proposed change.”* When exploiting different sources of information, Luciano works differently from Nicola; *“I like the idea of having video tutorials together with Stack Overflow discussions. However, the main source of information for me is Stack Overflow, while video tutorials should be used to fix problems; if I need to apply a new technology, I would like to start from Stack Overflow since there I can find snippets of code that I can copy and paste into my application. Then, if something goes wrong, I try to find a video tutorial to fix the problem.”*. Luciano also suggested a nice improvement; *“Besides the integration of video tutorials with discussions on forums, I suggest to add another source of information, namely sample projects. Specifically, on GitHub there are several sample projects that explain how to apply specific technologies. Having them together with video tutorials and Stack Overflow discussions would be fantastic.”* Another suggestion was the addition of a voting mechanism to provide information on the usefulness and the effectiveness of a specific (fragment of a) video tutorial. Luciano answered “absolutely yes” to our last question (*i.e.*, the one related to the usefulness of CODETUBE); *“I just added* CODETUBE *to my bookmarks. This is the tool I wanted. I spent several hours of the day and of the night on YouTube and Stack Overflow to fix problems or learn new things. This is part of my job, unfortunately. With* CODETUBE *I am sure that I will find relevant information quickly. I can finally go back to sleep during the night!”*. The day after the interview, we got a text message from Luciano: *“I have just used* CODETUBE *this morning. I was looking for something related to Android WebSocket. I found all I needed. Awesome!”*.

Giuseppe Socci, Project Manager @ Genialapps. Giuseppe answered “absolutely yes” to our first question, stating that in his opinion *“Video tutorials are a crucial source of information for learning a new technology”*. Instead, he answered “no” to our second research question related to the extraction of fragments; *“I am not 100% sure that extracting shorter fragments makes you more productive. It depends on the scenario where the video tutorial is used. To me, video tutorials should be used to learn a new technology. In this case I should watch the whole video. However, there could be cases where you just need to fix a problem or have some clarifications on a specific part of the technology. In this case watching fragments instead of whole videos could be worthwhile”*. In this particular scenario, Giuseppe found the possibility to filter video fragments on the basis of their category a great feature: *“the filtering based on the category of the video fragments can really help in improving productivity.”*

Giuseppe also suggested a way to make the tool more usable based on his way of interpreting video tutorials; *“the search of a video tutorial should be scenario-sensitive. Before searching, the user should specify why she is searching for a video tutorial. The first option could be 'I have a problem'. In this case, the search is based on fragments. The second option could be 'I want to learn'. Here, whole videos should be retrieved”*. As well as the other two developers, Giuseppe liked the integration of video tutorials with forum discussion (he answered “absolutely yes” to our third research question). Consistently with findings of Study I (Section 5.2.4), he highlighted the need for manually refining queries when retrieving Stack Overflow discussions: *“all the visualized Stack Overflow discussions are related to a specific video tutorial. However, Stack Overflow discussions should be useful to resolve a problem I encountered when applying the technology explained in the video tutorial. Thus, it might be useful to filter the retrieved discussion by a specific query (e.g., the type of error I got).”*. Finally, Giuseppe answered “yes” to our final question; *“I think that the tool is nice. You are trying to solve an important and challenging problem, that is merging accurately different sources of information in order to make them more productive.”*. Giuseppe also gave a suggestion on how to improve the visualization of the relevant fragments; *“After submitting a query,* CODETUBE *provides the list of relevant video fragments. However, it is quite difficult from the title of the video and the cover image to identify the most relevant one. I strongly suggest to show for each video the relevant textual part of the video content, similar to the part of the text in a Web page content visualized by Web search engines. The same approach could be used also to make the navigation of the fragments of a specific video easier.”*.

6.3 Reflection: Approach vs. Tool

The reception of CODETUBE was positive. All leading developers saw great value and even greater potential in this line of work. Several improvement suggestions, obtained also in Study I, regard the tool that embodies our approach, which we are currently considering. Clearly, tools can always be improved, given sufficient time and human resources. However, we would like to emphasize that, stepping beyond mere implementation and UI concerns, the main contribution of the paper lies in the underlying approach.

7 THREATS TO VALIDITY

Threats to *construct validity* are mainly related to the measurements performed in our studies, and in Study I and II in particular. In Study I this is mainly due to subjectiveness in the construction of the labeled fragment dataset used in our study, since each video has been fragmented and tagged by one expert only. However, during the second phase (open coding) two authors, before starting the open coding activity, performed a sanity check of the obtained fragments and tags. Finally, subjectiveness in the open coding was mitigated by employing a multiple-coder strategy, for which there has also been a very strong inter-rater agreement even since the first coding phase.

In Study II, instead of using proxy measures, we preferred to let developers evaluate video fragments and their related Stack Overflow discussions. Subjectiveness of such an evaluation was mitigated by involving multiple evaluators for each video, although as explained in Section 5.1 we favored the number of videos over the number of responses per fragment. Also, although a four or five-level Likert scale [30] could have provided a more accurate evaluation, we preferred a simpler three-level scale for the sake of facilitating the task to the respondents, which was already long, due to the need for watching the videos before answering the questions.

Threats to *internal validity* concern factors internal to our studies that could have influenced our results. In Section 3.5 we have shown how the CODETUBE parameters γ, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$ have been tuned. Instead, we did not tune the similarity threshold used to detect similar subsequent frames (10%). This is because such a threshold is mostly an optimization parameter that, based on tests we performed while defining our approach, does not really affect the way the video fragments are defined but only reduces the required computational time. In Section 3.1 we have reported results of a study aimed at assessing the impact of such a threshold on the approach's cost. Last, but not least, in Section 3.2 we have reported a study aimed at assessing the impact of the LCS heuristic in identifying related frames.

Another possible problem is that the evaluation in Study II could have been influenced by the knowledge of respondents about the topic. We mitigated this threat by discarding responses of participants not having any knowledge about Android. In addition, the evaluation is mainly related to cohesiveness, self-containment and relevance of video fragments, and relevance and complementariness of Stack Overflow discussions, rather than to how they would be helpful for the respondents. Also, a possible bias could have been introduced by the choice of the videos used in the survey. Such videos have been randomly sampled by considering 7 minutes as maximum video duration, and three as maximum number of fragments for each video and query results. These limitations have been introduced to restrict the survey duration to a reasonable time.

Concerning the machine learning classifier, for the preprocessing phase, we have mitigated possible multicollinearity problems by using a feature selection approach. Also, we have used SMOTE to deal with unbalanced data. Finally, while we have tried different machine learning techniques and chosen the one (Random Forest) producing the best results, it is possible that we did not consider techniques (or parameter settings for a technique) producing even better results than what we achieved. As explained in Section 3 we adopted a simple and possibly sub-optimal LDA calibration when extracting semantic features. This is in line with work using LDA in classification approaches [20].

Threats to *external validity* concern the generalizability of our findings. Our studies are limited to Java video tutorials only. We do not expect large differences in the structure of a video tutorial for a different programming language. Also, it is possible that results might not generalize. Also, although our approach captures a wide range of information characterizing video tutorials from different perspectives, it is possible that additional information might be required when dealing with tutorials about pieces of technology not considered in our dataset. Finally, the validity of the third study is limited to the three very specific mobile app development contexts we considered. However, the main aim of the third study was not to achieve generalizability, nor conclusions on statistical basis. Instead, as explained in Section 6, the main aim was to collect feedback about CODETUBE's applicability and suggestions for its improvement.

8 RELATED WORK

To the best of our knowledge, there is limited work about studying and analyzing video tutorials in the software engineering context. MacLeod *et al.* [2] have conducted an empirical study investigating how and why developers produce video tutorials. This study focused on screencasts, and its goals were to understand how and why developers produce and share such documentation. The findings revealed that videos are a useful medium for communicating program knowledge between developers, and that they present key advantages compared to written documentation. MacLeod *et al.*'s study therefore brings into attention the need to leverage such information and motivates our work. CODETUBE is the first approach to actually process and recommend information extracted from video documentation for software development.

A very recent work by Yadid and Yahav [32] proposes an approach to extract source code from video tutorials. Their approach combines OCR with statistical language models and ensures a recall between 66% and 81%, and a precision between 80% and 82%. The approach adopted by CODETUBE to identify the source code to be indexed is different (as explained in Section 3) as it is based on identifying the IDE source code frame and in filtering source code text by combining the use of programming language dictionaries and island parsing. In future work we contemplate the possibility of comparing or combining the two source code extraction approaches. CODETUBE, however, goes beyond just source code extraction, as it (i) identifies, classifies, and indexes cohesive video tutorial fragments, (ii) links such fragments to relevant Stack Overflow discussions, and (iii) provides a mechanisms for querying the indexed fragments.

In the following subsections we discuss related work about (i) recommender systems for software documentation and code examples, (ii) automatic categorization of software engineering artifacts, (iii) multimedia retrieval and processing, (iv) analysis of multimedia tutorials and lectures, and (v) use of multimedia in learning.

8.1 Recommender systems for software documentation and code examples

Numerous approaches have been proposed to provide developers with official or informal documentation for their task at hand, as well as code samples they can reuse. Among the various informal documentation sources, Stack Overflow has been used by many recommender systems [3], [6], [33], [34],

[35], [36]. Other recommenders have focused on recovering links between code and documentation [37], with some focusing on recommending or enhancing API documentation [4], [38]. Among these, the work of Petrosyan *et al.* [39] is the most related, as it analyzed tutorials to extract fragments explaining API types. With respect to such approaches, CODETUBE is specific for analyzing video tutorials and recommending their cohesive and self-contained fragments.

Other approaches focused on the in-project knowledge [40], [41], [42], [43] to retrieve relevant artifacts for developers. A prominent example is Hipikat [40], a tool that builds a project memory from artifacts created during a software development project (*e.g.,* source code, documentation, emails), and recommends such artifacts if they result relevant to the task being performed. Several approaches focused on mining API usage to synthesize code samples [7], [8], [9], [10], [44], [45], [46], [47]. An example is proposed Prospector [44], a tool that synthesize jungloids using both API method signatures and jungloids mined from sample client programs. Prospector integrates with the Eclipse IDE code assistance feature, and infers queries without programmer intervention.

Other work suggested relevant web discussions to help solve coding problems [6], [35], [48], or from online resources in general. An example is Dora [48], a tool integrated into Visual Studio that automatically query and analyze online discussions (*e.g.,* Stack Overflow) to locate relevant solutions to programming problems.

The infrastructure of CODETUBE is designed such that any source of documentation can potentially be used to complement the information extracted from video tutorials. While the current CODETUBE instantiation leverages Stack Overflow discussions, future work will investigate integrating additional information sources.

8.2 Automatic Categorization of Software Engineering Artifacts

Researchers have proposed many approaches that categorize various kinds of software engineering artifacts, like bug reports, change sets or API documentation. For example, Antoniol *et al.* used text mining techniques to separate bug reports from enhancements when issue trackers fail to do so [49]. Hindle *et al.* built a classifier to categorize a change request into corrective/adaptive/perfective maintenance, feature addition, or non-functional improvement [50]. Instead, Kim *et al.* applied machine learning to detect and distinguish change sets that induce (or not) a bug fix [51], and Thung *et al.* classified the type of defects by using features collected from both bug tracking systems and code repositories [52]. Concerning other type of artifacts, Bacchelli *et al.* proposed a technique to classify the content of development emails at the line level, detecting for example code, patches, or stack traces [53]. McMillan *et al.* presented an approach to classify software applications by using API calls from third-party libraries [54]. Last, but not least, the approach developed by Chatri and Robillard [4] aimed at distinguishing "reference" (*i.e.,* indispensable, or at least valuable) parts of API documentation, from less valuable details. In our previous work we focused on Stack Overflow discussions, and in particular to the classification and categorization of quality using readability and structural metrics [55], [56].

8.3 Multimedia Retrieval

Multimedia information retrieval focuses on extracting and retrieving relevant information from multimedia resources (*e.g.,* images, audio, or video). One problem in the field is splitting a video into semantically coherent fragments. Existing approaches usually employ supervised machine learning techniques applied to various textual, acoustic, and visual features [57] to resolve such an issue. Galuščáková and Pecina [58] explored the use of Passage Retrieval segmentation techniques to retrieve relevant segments by a textual query in a set of audio-visual recordings. Mettes *et al.* [59] proposed an approach using hierarchical clustering and syntactic and semantic similarity metrics to identify the segments.

While CODETUBE also identifies fragments within a video, it is significantly different than those proposed in multimedia retrieval: it is not supervised and bases its segmentation algorithm on information specific to the software development domain, *i.e.,* the occurrence of code in the tutorials, something that has not been done in the field of multimedia information retrieval [60].

8.4 Analysis of Multimedia Tutorials and Lectures

Recent work in related fields, such as human-computer interaction has also recognized the general limitations of current multimedia platforms and has set out to address some of these limitations. Banovic *et al.* [61] proposed an approach able to analyze end-user software video tutorials and extract the GUI elements appearing on the screen (*e.g.,* menus, icons, etc.), as well as the interaction with those elements present in the video (*e.g.,* user clicks), without a-priori knowledge of the application. Other works have also extracted GUI elements from end-user software training materials, but using templates to identify them [62], [63], [64]. Moslehi *et al.* [65] proposed an approach to mine the speech in YouTube videos, extract use case scenarios, and show how their content can supplement existing documentation.

Other related works have focused on segmenting videos based on various criteria. Pongnumkul *et al.* [64] introduced Pause-and-Play, a system that detects important events in video tutorials, segments videos based on these events, and links them to events in the target application as the end user is replicating what they see on the screen. DemoCut [66], a video editing system introduced by Chi *et al.* also segments video tutorials based on important events, but it requires the users to mark these key moments within videos showing physical demonstrations. Shin *et al.* [67] introduced "visual transcripts" for blackboard-style video lectures, obtained by segmenting a video based on visual entities such as equations or figures, linking parts of the available audio transcripts to these entities, and then producing a transcript that intertwines them.

In addition, some research investigated ways to improve categorization, summarization, and navigation of video lectures and tutorials. For example, the technique by Chatbri *et al.* [68] generates and analyzes the audio transcripts of science and math videos in order to categorize them based on their discipline. The video digests introduced by Pavel *et al.* [69] allow users to browse and skim long talks by segmenting and organizing the videos into chapters, and then providing short summaries for each section. Kim *et al.* [70] also aim to improve the navigation and summarization of educational videos, and propose various techniques to address this, such as a video timeline, transcript search, keyword summaries, relevant frame extraction, and a visual summary.

While similar to our work in analyzing part of the information captured in video tutorials, CODETUBE differs from all this previous work as it focuses specifically on software development tutorials, and includes tailored analysis for this purpose: identifying and extracting source code found within the video, segmenting videos based on the presence of code, classifying video fragments into categories found specifically in software development video tutorials, and linking video fragments to other, complementary sources of developer documentation such as Stack Overflow.

An extensive treatment of approaches for video analysis and repackaging for the purpose of distance education is provided in a book by Ram and Chaudhuri [71]. In particular, they introduce the concept of media repackaging on client's side, to provide remote students with customize content. The general idea is to provide videos eliminating redundant frames, *e.g.,* by detecting slide transitions in presentations, or by distinguishing frames providing content by those not providing content (*e.g.,* where an instructor just talks).

While we share with Ram and Chaudhuri the goal of producing short video fragments, CODETUBE is very specific to software development video tutorial. This is because (i) it employs specific heuristics for identifying IDEs and their content; (ii) it relies on source-code related text-mining approaches [72], [73], [74] to analyze the source code shown in video frames and to identify fragments relevant to a query.

8.5 Use of Multimedia in Learning

Multimedia resources, especially those using videos, have been shown to be a very effective medium for learning, which is also often preferred by students over written text. Mayer [75] established twelve principles, based on numerous studies, that define the use and efficiency of multimedia in learning environments. Some of these principles clearly motivate CODETUBE: (i) the *multimedia principle* states that people learn better from words and graphics than from words alone; (ii) the *temporal contiguity principle* indicates that people learn better when corresponding words and pictures are presented simultaneously rather than successively. These two principles stand at the core of CODETUBE and suggest that video tutorials could be a better learning medium for developers than traditional software documentation, where text is often the only information source, or, at best, it is shown in a sequence with graphics. Future work will investigate this direction through user studies. Previous work has also shown that YouTube can be an efficient way to teach new concepts. Duffy [76] has shown that students like to use YouTube, as it provides a user-guided experience. For Mullamphy *et al.* [77] videos allow students to learn at their own pace. These observations were also confirmed within our first study.

9 CONCLUSION

Software development video tutorials are a modern medium to disseminate in-depth technical knowledge, and if we were to make an informed guess, they represent the next frontier in software documentation. However, the intrinsic nature of audio-visual content poses a number of challenges. First, it is difficult, if not impossible, to search videos based on their contents. This is a prime requisite to make the information contained in the video tutorials more accessible. Second, it is non-trivial to understand whether a video contains the information one is looking for, short of watching the whole video.

We presented CODETUBE, a novel approach to extract and classify relevant fragments from software development video tutorials. CODETUBE mixes several existing approaches and technologies like Optical Character Recognition (OCR) and island parsing to analyze the complex unstructured contents of the video tutorials. Our approach extracts video fragments by merging the code information located and extracted within video frames, together with the speech information provided by audio transcripts. Also, it automatically classify the "type" of video fragment (*e.g.,* theoretical, implementation) and complements the video fragments with relevant Stack Overflow discussions. We conducted three studies to evaluate CODETUBE, showing its ability to identify and correctly classify meaningful code fragments. Also, we investigated the perception of our approach in industry environments by interviewing three leading developers, receiving useful insights on the strengths and potential extensions of our current work. To our knowledge, CODETUBE is the first, and freely available[20] tool to perform video fragment analysis for software development.

The approach presented in this paper is, to the best of our knowledge, the first classification and fragmentation approach for video tutorial fragments used in software engineering. The work on CODETUBE will continue by experimenting additional strategies to successfully deal with sources of noise (*e.g.,* scrolling, zooming) present in video tutorials as well as for improving the linking between each video fragment with the related Stack Overflow discussions. Finally, we plan to revise the CODETUBE user interface to take inspiration from recent advances in the field of Human-Computer interaction (see *e.g.,* [70]). Our long-term vision is to seamlessly integrate video tutorials with the development environment, and to move towards the concept of a *holistic recommender*, where we aim at providing additional sources of information other than Stack Overflow altogether.

ACKNOWLEDGMENTS

Ponzanelli and Lanza thank the Swiss National Science foundation for the financial support through SNF Project ESSENTIALS, No. 153129. Bavota thanks the Swiss National Science foundation for the financial support through SNF Project JITRA, No. 172479.

REFERENCES

[1] M. Umarji, S. Sim, and C. Lopes, "Archetypal Internet-Scale source code searching," in *Open Source Development, Communities and Quality*, ser. IFIP The International Federation for Information Processing, B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, Eds., vol. 275. Springer US, 2008, pp. 257–263.

20. http://codetube.inf.usi.ch

[2] L. MacLeod, M.-A. Storey, and A. Bergen, "Code, camera, action: How software developers document and share program knowledge using YouTube," in *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*, 2015, pp. 104–114.

[3] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*. IEEE Press, 2013, pp. 832–841.

[4] M. P. Robillard and Y. B. Chhetri, "Recommending reference API documentation," *Empirical Software Engineering*, pp. 1–29, 2014.

[5] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*. ACM, 2005, pp. 117–125.

[6] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the IDE into a self-confident programming prompter," in *Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*. ACM, 2014, pp. 102–111.

[7] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*. IEEE, 2012, pp. 782–792.

[8] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of ESEC/FSE 2007 (6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering)*. ACM, 2007, pp. 25–34.

[9] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*. ACM, 2014, pp. 664–675.

[10] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can I use this method?" in *Proceedings of ICSE 2015 (37th IEEE/ACM International Conference on Software Engineering)*, 2015, pp. 880–890.

[11] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*. IEEE CS, 2001, pp. 13–22.

[12] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *In Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, 2011, pp. 476–479.

[13] L. Ponzanelli, A. Mocci, and M. Lanza, "Stormed: Stack overflow ready made data," in *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*. ACM Press, 2015, pp. 474–477.

[14] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[15] R. M. Groves, *Survey Methodology, 2nd edition*. Wiley, 2009.

[16] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychosocial Measurement*, vol. 20, pp. 37–46, 1960.

[17] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, Mar. 2003.

[19] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, 2013, pp. 522–531.

[20] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of ICSE 2014 (36th ACM/IEEE International Conference on Software Engineering)*, 2014, pp. 1025–1035.

[21] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*. Morgan Kaufmann Publishers Inc., 2011.

[22] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[23] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, and M. Lanza, "Replication package." http://reveal.inf.usi.ch/reports/codetube-tse/.

[24] T. M. Mitchell, *Machine Learning*, 1st ed. McGraw-Hill, Inc., 1997.

[25] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of ICPC 2004 (12th IEEE International Workshop on Program Comprehension)*. IEEE, 2004, pp. 194–203.

[26] L. Ponzanelli, "Holistic recommender systems for software engineering," in *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*. ACM, 2014, pp. 686–689.

[27] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, Jun. 2002.

[28] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[29] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza, "Too long; didn't watch! extracting relevant fragments from software development video tutorials," in *Proceedings of ICSE 2016 (38th International Conference on Software Engineering)*. ACM Press, 2016, pp. 261–272.

[30] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.

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

[32] S. Yadid and E. Yahav, "Extracting code from programming tutorial videos," in *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, 2016, pp. 98–111.

[33] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*. IEEE Press, 2012, pp. 85–89.

[34] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of SUITE 2011 (3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation)*. ACM, 2011, pp. 17–20.

[35] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *Proceedings of CSMR 2013 (17th European Conference on Software Maintenance and Reengineering)*, 2013, pp. 59–66.

[36] ——, "Seahawk: Stack overflow in the ide," in *Proceedings of ICSE 2013 (35th International Conference on Software Engineering), Tool Demo Track*. IEEE, 2013, pp. 1295–1298.

[37] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia, "Information retrieval models for recovering traceability links between code and documentation," in *Proceedings of ICSM (16th IEEE International Conference on Software Maintenance)*. IEEE CS Press, 2000, pp. 40–51.

[38] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*. ACM, 2014, pp. 643–652.

[39] G. Petrosyan, M. P. Robillard, and R. D. Mori, "Discovering information explaining api types using text classification," in *Proceedings of ICSE 2015 (37th ACM/IEEE International Conference on Software Engineering)*, 2015, pp. 869–879.

[40] D. Cubranic, G. Murphy, J. Singer, and K. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[41] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*. ACM, 2006, pp. 1–11.

[42] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, Dec. 2006.

[43] R. Holmes and A. Begel, "Deep intellisense: A tool for rehydrating evaporated information," in *Proceedings of MSR 2008 (5th IEEE International Working Conference on Mining Software Repositories)*. New York, NY, USA: ACM, 2008, pp. 23–26.

[44] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," in *Proceedings of PLDI 2005 (16th ACM SIGPLAN Conference on Programming Language Design and Implementation)*. ACM, 2005, pp. 48–61.

[45] N. Sawadsky and G. C. Murphy, "Fishtail: From task context to source code examples," in *Proceedings of TOPI 2011 (1st Workshop on Developing Tools As Plug-ins)*. ACM, 2011, pp. 48–51.

[46] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the*

ASE (22nd IEEE/ACM International Conference on Automated Software Engineering). New York, NY, USA: ACM, 2007, pp. 204–213.

[47] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *Proceedings of the Visual Languages and Human-Centric Computing*, ser. VLHCC '06. IEEE Computer Society, 2006, pp. 195–202.

[48] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes, "Automatically locating relevant programming help online," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 127–134.

[49] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of CASCON 2008*. ACM, 2008, pp. 304–318.

[50] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classification of large changes into maintenance categories," in *In Proceedings of ICPC 2009 (17th IEEE International Conference on Program Comprehension)*. IEEE Press, 2009, pp. 30–39.

[51] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[52] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*. IEEE Press, 2015, pp. 60–70.

[53] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza, "Content Classification of Development Emails," in *Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, 2012.

[54] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Proceedings of ICSM 2011 (27th IEEE International Conference on Software Maintenance)*. IEEE Computer Society, 2011, pp. 343–352.

[55] L. Ponzanelli, A. Mocci, A. Bacchelli, and M. Lanza, "Understanding and Classifying the Quality of Technical Forum Questions," in *Proceedings of QSIC 2014 (14th International Conference on Quality Software)*. IEEE CS Press, 2014, pp. 343–352.

[56] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton, "Improving Low Quality Stack Overflow Post Detection," in *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution)*. IEEE, 2014, pp. 541–544.

[57] T. Du, Y. Junsong, and D. Forsyth, "Video event detection: From subvolume localization to spatiotemporal path search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 2, pp. 404–416, Feb. 2014.

[58] P. Galuščáková and P. Pecina, "Experiments with segmentation strategies for passage retrieval in audio-visual documents," in *Proceedings of ICMR 2014 (4th International Conference on Multimedia Retrieval)*. ACM, 2014, pp. 217:217–217:224.

[59] P. Mettes, J. C. van Gemert, S. Cappallo, T. Mensink, and C. G. M. Snoek, "Bag-of-fragments: Selecting and encoding video fragments for event detection and recounting," in *Proceedings of ICMR 2015 (5th ACM International Conference on Multimedia Retrieval)*. ACM, 2015, pp. 427–434.

[60] A. R. Ram and S. Chaudhuri, *Video Analysis and Repackaging for Distance Education*, 1st ed. Springer-Verlag New York, 2012.

[61] N. Banovic, T. Grossman, J. Matejka, and G. Fitzmaurice, "Waken: reverse engineering usage information and interface structure from software videos," in *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12)*. ACM, 2012, pp. 83–92.

[62] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using gui screenshots for search and automation," in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '09. ACM, 2009, pp. 183–192.

[63] L. Bergman, V. Castelli, T. Lau, and D. Oblinger, "Docwizards: a system for authoring follow-me documentation wizards," in *Proceedings of the 18th annual ACM symposium on User interface software and technology (UIST '05)*, ser. UIST '09. ACM, 2005, pp. 191–200.

[64] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen, "Pause-and-play: automatically linking screencast video tutorials with applications," in *Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11)*. ACM, 2011, pp. 135–144.

[65] P. Moslehi, B. Adams, and J. Rilling, "On mining crowd-based speech documentation," in *Proceedings of MSR 2016 (13th International Conference on Mining Software Repositories)*. ACM, 2016, pp. 259–268.

[66] P.-Y. Chi, J. Liu, J. Linder, M. Dontcheva, W. Li, and B. Hartmann, "Democut: generating concise instructional videos for physical demonstrations," in *Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST '13)*. ACM, 2013, pp. 141–150.

[67] H. V. Shin, F. Berthouzoz, W. Li, and F. Durand, "Visual transcripts: lecture notes from blackboard-style lecture videos," *ACM Transactions on Graphics*, vol. 34, no. 6:240, Nov. 2015.

[68] H. Chatbri, K. McGuinness, S. Little, J. Zhou, K. Kameyama, P. Kwan, and N. E. O'Connor, "Automatic mooc video classification using transcript features and convolutional neural networks," in *Proceedings of the 2017 ACM Workshop on Multimedia-based Educational and Knowledge Technologies for Personalized and Social Online Training (MultiEdTech '17)*. ACM, 2017, pp. 21–26.

[69] A. Pavel, C. Reed, B. Hartmann, and M. Agrawala, "Video digests: a browsable, skimmable format for informational lecture videos," in *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14)*. ACM, 2014, pp. 573–582.

[70] J. Kim, P. J. Guo, C. J. Cai, S.-W. D. Li, K. Z. Gajos, and R. C. Miller, "Data-driven interaction techniques for improving navigation of educational videos," in *Proceedings of UIST 2014 (27th Annual ACM Symposium on User Interface Software and Technology)*. ACM, 2014, pp. 563–572.

[71] A. R. Ram and S. Chaudhuri, *Video Analysis and Repackaging for Distance Education*. Springer Publishing Company, Incorporated, 2014.

[72] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[73] 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.

[74] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 420–432, 2007.

[75] R. E. Mayer, *Multimedia Learning*, 2nd ed. New York, NY, USA: Cambridge University Press, 2009.

[76] P. Duffy, "Engaging the youtube google-eyed generation: Strategies for using web 2.0 in teaching and learning," in *European Conference on ELearning, ECEL*, 2007, pp. 173–182.

[77] D. Mullamphy, P. Higgins, S. Belward, and L. Ward, "To screencast or not to screencast," *Anziam Journal*, vol. 51, pp. C446–C460, 2010.