Empirical Software Engineering (2020) 25:1536–1572
https://doi.org/10.1007/s10664-019-09759-w

Code Localization in Programming Screencasts

Mohammad Alahmadi[1] · Abdulkarim Khormi[1] · Biswas Parajuli[1] · Jonathan Hassel[1] · Sonia Haiduc[1] · Piyush Kumar[1]

Published online: 20 January 2020

Abstract
Programming screencasts are growing in popularity and are often used by developers as a learning source. The source code shown in these screencasts is often not available for download or copy-pasting. Without having the code readily available, developers have to frequently pause a video to transcribe the code. This is time-consuming and reduces the effectiveness of learning from videos. Recent approaches have applied Optical Character Recognition (OCR) techniques to automatically extract source code from programming screencasts. One of their major limitations, however, is the extraction of noise such as the text information in the menu, package hierarchy, etc. due to the imprecise approximation of the code location on the screen. This leads to incorrect, unusable code. We aim to address this limitation and propose an approach to significantly improve the accuracy of code localization in programming screencasts, leading to a more precise code extraction. Our approach uses a Convolutional Neural Network to automatically predict the exact location of code in an image. We evaluated our approach on a set of frames extracted from 450 screencasts covering Java, C#, and Python programming topics. The results show that our approach is able to detect the area containing the code with 94% accuracy and that our approach significantly outperforms previous work. We also show that applying OCR on the code area identified by our approach leads to a 97% match with the ground truth on average, compared to only 31% when OCR is applied to the entire frame.

Keywords Programming video tutorials · Software documentation · Source code · Deep learning · Video mining

1 Introduction

The World Wide Web has become one of the most used sources of information by software developers, with studies indicating that searching for and acquiring information online takes

Communicated by: Shane McIntosh, Leandro L. Minku, Ayşe Tosun, Burak Turhan

This article belongs to the Topical Collection: *Predictive Models and Data Analytics in Software Engineering (PROMISE)*

✉ Mohammad Alahmadi
alahmadi@cs.fsu.edu

Extended author information available on the last page of the article.

up to 30% of a programmer's time (Brandt et al. 2009; Grzywaczewski and Iqbal 2012). Most of this time is spent consulting informal online documentation in the form of Q&A websites, tutorials, API documentation, etc. Software programming screencasts are becoming more and more popular and have seen rapid growth in production and usage (MacLeod et al. 2015).

Screencasts are particularly useful at offering a step-by-step guide to performing certain programming tasks or learning programming concepts. However, some limitations still prevent developers from using them to their full potential. For example, one of the most important pieces of information present in programming screencasts is the source code being displayed on screen. However, this code is embedded in the video and not available for being indexed and searched, for being extracted and reused by programmers or being linked to other sources of information such as GitHub or StackOverflow discussions. Therefore, designing tools and techniques that can automatically and correctly extract code appearing in video tutorials is of immediate importance, as it would give developers access to a wealth of source code currently not leveraged, as well as open the door to numerous software engineering applications.

One solution to make the source code in screencasts available would be to apply Optical Character Recognition (OCR) on video frames. However, software programming screencasts contain much more text than just source code, such as menus, package hierarchies of a project, the program output, etc. Therefore, applying OCR to the entire screen would result in a lot of noise being extracted as well, which would be concatenated by the OCR to the source code, corrupting the latter in the process. An example of this problem is shown in Fig. 1, where OCR has been applied to the entire frame. The valid code is highlighted in yellow, but it is surrounded by noise extracted from the rest of the frame. Therefore, in order to correctly extract reusable code using OCR, the first step would be to accurately identify the code editing window and to apply OCR only on it. While a few previous works have started using OCR to extract code from video frames (Yadid and Yahav 2016; Ponzanelli et al. 2017; Khandwala and Guo 2018), their approaches for identifying the code editing window suffer from limitations that drastically limit their applicability, such as being programming language-dependent (Ponzanelli et al. 2017) and using heuristics or assumptions that are not generally applicable (Yadid and Yahav 2016; Ponzanelli et al. 2017; Khandwala and Guo 2018).

We propose the first approach to identify the code editing window in programming screencasts that is language-independent, heuristic-free, and generalizable. Our method is

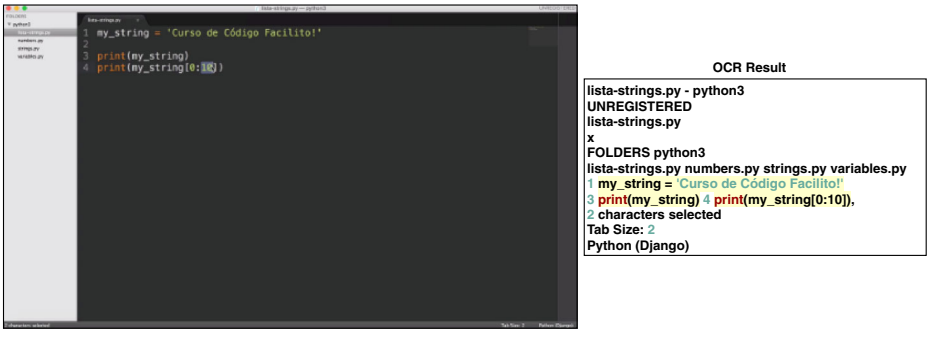


Fig. 1 The result of applying OCR on the entire frame, showing a significant amount of noise being extracted. The actual code is highlighted in yellow

based on extracting the spatial features of video frames using a deep Convolutional Neural Network (CNN). Then, we detect the main code editing window using object detection approaches based on a CNN. Our approach is able to determine the code editing window with an accuracy of 94% on average.

This work builds upon our previously published paper for detecting the code editing window (Alahmadi et al. 2018) and extends it in several ways:

- Our previous work was evaluated on a dataset of video frames extracted from only 150 Java video programming tutorials. In this paper, we extended our dataset to include frames from a total of 450 Java, C#, and Python video tutorials (150 videos for each language). Thus, we also show that our approach is generalizable to screencasts covering different programming languages.
- In our previous work, we only used a region-free object detection architecture (i.e., YOLO) to determine the code editing window. In this paper we experiment with two region-free, as well as three region-proposal approaches and determine that one of the region-proposal object detectors, Faster R-CNN, performs the best, outperforming YOLO and reaching an accuracy of 94% on average.
- We replicated the approach proposed by Ponzanelli et al. in CodeTube (Ponzanelli et al. 2017) for finding the code editing window in Java screencasts and compared our approach with it in a new research question in our evaluation.
- We also evaluated the impact of our approach in removing noise when applying OCR on screencast frames for extracting source code and compared it with CodeTube (Ponzanelli et al. 2017). We show that applying OCR on the code region identified by our approach leads to a 97% match with the ground truth source code on average, compared to 62% when applying the approach in CodeTube and only 31% when OCR is applied to the entire frame. Therefore, our approach leads to a noise reduction of 35% in the extracted source code when compared with CodeTube and 66% compared to OCR applied to the entire frame.
- Last but not least, to further reduce the noise and speed up the process of code extraction from screencasts, we evaluated the ability of our approach to detect frames that contain only partially visible code (due to the presence of an obstructing window) and frames containing no code. These frames could then be discarded to make the code extraction process even faster. We compare our approach against the classification approach proposed by Ott et al. (2018a) for the same purpose and find that our technique outperforms it.

In this paper, we focus on answering the following research questions:

RQ1: **Which deep architecture model performs the best at localizing the code editing window in video programming tutorials?** To answer this question, we first identified the video frames that contain code that is fully visible (i.e., code is present in the frame and not obstructed by any other window). We then train five different deep neural networks to detect the code editing window in these frames (i.e., the area of the frame where the code is located). We found that the best architecture, called Faster R-CNN, can detect the code editing window with an overall accuracy of 94%. Our approach for the next research questions is therefore based on this best-performing deep architecture.

RQ2: **How does our approach compare to previous work that identified the code editing window in video frames?** To further evaluate our approach, we compared it with a previous approach for determining the code editing window proposed

by Ponzanelli et al. (2016a, b, 2017) in their tool called CodeTube. We replicated the proposed approach of CodeTube in finding the code editing window and compared it to our approach. We found that our approach significantly outperformed CodeTube in identifying the code section.

RQ3: **Does our approach help OCR techniques to better extract code from video programming tutorials?** We compared the OCR-ed text from the code editing window predicted by our approach to that extracted from the code editing window predicted by CodeTube and also the text extracted from the entire frame. We found that our approach leads to a much more accurate code extraction compared to the other approaches.

RQ4: **Can our approach be used for discriminating between frames containing fully-visible code, partially-visible code and frames containing no code?** In RQ1, we showed that our approach can achieve a high accuracy in identifying the code editing window in frames containing fully visible code. However, not all frames fall under this category. Therefore, in this question we aim to determine if we can use our approach to also predict if the frames contain fully visible code in the first place. In particular, we aim to use our approach for classifying frames into three categories: containing fully visible code, partially visible code or no code at all. For RQ1, this classification was done manually, to identify only the fully visible code frames on which our approach was applied. However, if we prove that our approach is successful in this classification task, this step could be done automatically in the future, to filter out the unneeded frames before localizing the code editing window. We compared the performance of our approach in addressing this problem with a previous approach proposed by Ott et al. (2018a) for the same purpose. We found that our technique improves the results compared to previous work, achieving an overall average accuracy of 96%.

The rest of the paper is organized as follows: Section 2 introduces the main architectures and approaches for image analysis used in our work, Section 3 introduces our approach and its components, Section 4 describes the empirical evaluation we performed, Section 5 presents an overview of the related work, and finally Section 6 concludes the paper.

2 Background

This section introduces the main image analysis approaches and architectures used in our work.

2.1 Image Feature Extraction

Feature extraction algorithms have been proposed and practically used in a wide range of computer vision applications. The features extracted for an image have to be robust and distinctive for that image. There are two main phases in extracting features from an image. First, distinctive regions or *key-points* of the image are identified. These key-points could be corners, blobs, etc. Second, for each key-point, a feature vector or *descriptor* is assigned, which is robust to noise. In our approach, we use image feature extraction in order to compare consecutive images in a video based on their features and determine if they contain duplicate information or not (see Section 3.1).

In this paper we extract the features of the video frames using the Scale Invariant Feature Transform (SIFT), which has been successfully used in image recognition applications (Lowe 1999, 2004). SIFT outperformed several other descriptors in terms of recognition and matching tasks due to its invariance to rotation and scaling (Mikolajczyk and Schmid 2005). SIFT's algorithm has four main steps: approximating key-point locations, refining key-point positions, assigning orientations to key-points and finally extracting features for each key-point. SIFT uses a Difference of Gaussians (DoG) to find the key-points in images. DoG is obtained by computing the difference of a Gaussian blur of an image with two different scaling factors. Once the DoG is found, the images are searched for local extrema, which represent the set of potential key-points. This set is then refined by eliminating any low-contrast and edge key-points. SIFT then assigns the remaining key-points an orientation and a vector of descriptors that are invariant to rotations, scale, and illumination. SIFT considers a small region around each key-point and divides it into cells. A gradient orientation histogram is built inside each cell. These gradient orientation histograms are then sorted bearing in mind the dominant orientation of the key-point. This then gives us a descriptor which is invariant to rotations. To make it invariant to scale, the size of the window needs to be adjusted according to the scale of the key-point. To make it illumination invariant, the histogram entries are weighted by gradient magnitude. At the end of this process, SIFT provides the key-points and their feature vectors that will be used to compare one image to another.

2.2 Object Detection Based on CNN

Computer vision has seen numerous advances in recent years in feature extraction, image recognition, image classification, etc. One category of remarkable advances has been in the area of object detection, where recent algorithms have shown impressive performance in finding objects inside an image or a scene. Most state-of-the-art object detection methods utilize Convolution Neural Networks (CNNs) to not only classify an object into a specific category but also to find a precise location of that object inside an image (LeCun et al. 1999; Hu et al. 2015). We therefore make use of this kind of architectures in our approach for identifying and locating the code editing window in a screencast frame. In particular, we make use of both types of existing object detectors based on CNN architectures: region-based and region-free.

2.2.1 Region-Based Object Detection

Region-based object detection is based on dividing an image into *potential regions* that are the most likely to contain an object. This eliminates the unnecessary computational cost of running a detector on the entire image, as done in other approaches such as the sliding window algorithm. The features of the potential regions are extracted using a CNN and eventually classified by the output layer of the network using a softmax unit.

There have been several object detection methods that show promising results by using a selective search method to generate a number of regions for an image (Uijlings et al. 2013). The selective search method is based on *image segmentation*, first introduced by (Felzenszwalb and Huttenlocher 2004). The image segmentation algorithm divides an image into regions based on pixel intensity, color, motion, etc. Then, proposed regions are created using the selective search method (Uijlings et al. 2013). Selective search combines different regions together for the object detection task (i.e., finding region proposals). As an example, the results of image segmentation and selective search on a frame extracted from a

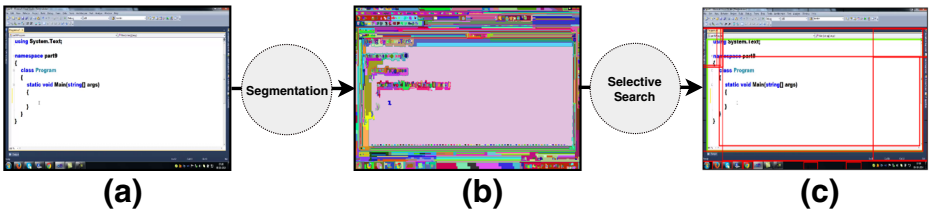


Fig. 2 Code frame segmentation (**b**) and selective search (**c**) results

programming video tutorial are shown in Fig. 2. In this case, out of the detected potential regions, the green rectangle contains the code. Region-based CNN algorithms or R-CNN then adjust the potential regions during the learning process.

R-CNN performs features extraction based on CNN in each proposed region that is created using the selective search method (Girshick et al. 2013). Forward propagation is performed on each of the potential regions that output the extracted features using the output dense layer. A Support Vector Machine is then used to classify each region based on the input features. R-CNN then uses linear regression to the bounding box to produce a tighter box on the object. Although R-CNN has been influential in the object detection field, the performance of R-CNN was its biggest issue. Running a CNN classifier in each proposed region makes the overall process of finding the location of an object very slow.

The performance issue was later addressed by *Fast R-CNN* (Girshick 2015). Fast R-CNN overcomes the computational cost of running the CNN on each proposed region by instead running the CNN on the entire image just once. In other words, Fast R-CNN extracts the entire set of features of the image only once, which reduces the computational time significantly. Fast R-CNN uses selective search to propose a number of regions. Each Region of Interest can be represented by the features that were extracted previously using a CNN. The Region of Interest pooling layer then changes the size of the proposed regions into a fixed size. These fixed feature vectors are then fed into a fully connected layer that outputs the predicted class and the bounding box of the object.

Faster R-CNN is the third iteration of R-CNN that aims to further increase the performance of the neural detector (Ren et al. 2015). Region Proposal Network was proposed as an alternative to the selective search algorithm used in previous iterations in order to overcome the performance issues. Instead of using selective search to find the region proposal which is expensive, Region Proposal Network is used to create the region proposal and predict the bounding box of the object, as well as the Objectness score.[1] Faster R-CNN outperformed all R-CNN variations and has been inspiring several follow-up applications (Shrivastava and Gupta 2016; Kim et al. 2016).

Faster R-CNN predicts each region using multiple fully-connected layers. This makes the prediction of each patch or proposed region expensive. Region-based Fully-Convolutional Networks (R-FCN) were therefore proposed as a further performance improvement of Faster R-CNN (Dai et al. 2016). We evaluate the use of both Faster R-CNN and R-FCN region-based approaches for detecting the location of the code region in screencasts in our empirical evaluation.

[1] "Objectness" indicates if a box contains an object.

2.2.2 Region-Free Object Detection

In this section, we explain two popular real-time object detector models, Single Shot Detection (SSD) (Liu et al. 2016) and You Only Look Once (YOLO) (Redmon et al. 2015), which we also experiment with in our evaluation to determine the best model for detecting code regions. Since these models have a single feed-forward convolutional network, they predict the class and the bounding box in one shot. In other words, there is no need for classifying different proposals/regions, which makes the process of finding the location of an object much faster. When comparing region-free detection to region-based detection, there is a trade-off between speed and accuracy (Huang et al. 2017). The region-based approaches are more accurate but slower than the region-free ones, as also shown later in the results of our evaluation.

SSD uses a single network for object localization. It generates a set of fixed-size bounding boxes and assigns a prediction score and box offset for each of the boxes using convolutional filters. Then, it adjusts the box accordingly to cover the object location.

YOLO is a unified CNN architecture which not only detects the position of the bounding box of each object present in an image but also predicts its class. Thus, YOLO looks at an input image just once and performs both detection and classification of multiple objects in one shot. Although YOLO can detect objects of multiple classes, in our case we only consider a single class: the code region. YOLO is fast since it does not require a complex pipeline and can make predictions at 150 frames per second, which makes it highly applicable to real-time video streams. It analyzes the entire frame during training and test time and encodes contextual information about classes as well as their appearance. YOLO also learns the generalizable representations of objects, so it is less likely to crash when given unexpected inputs.

3 Approach

In this section we describe our approach, which has two main components: duplicate frame elimination and code editing window localization. For convenience, we include a summary of the notations and acronyms we use in Table 1.

Table 1 The list of main terminology and acronyms used in this paper

Symbol	Description
Faster R-CNN	Faster Region-based Convolutional Neural Networks (region-based object detector)
R-FCN	Region-based Fully-Convolutional Networks (region-based object detector)
SSD	Single Shot Detector (region-free object detector)
YOLO	You Only Look Once (region-free object detector)
SIFT	Scale Invariant Feature Transform (feature extraction algorithm)
OCR	Optical Character Recognition
FVC	Fully Visible Code frames (i.e., frames where the code is present and not obstructed)
PVC	Partially Visible Code frames (i.e., frames where the code is present, but obstructed by an overlapping window)
NC	No Code frames (i.e., frames where there is no code present)

3.1 Duplicate Frame Elimination

Programming screencasts contain a lot of redundant frames, as identified by previous work (Ellmann et al. 2017). This can significantly impact the performance of the analysis of video tutorials. Therefore, the first step in our approach focuses on the elimination of duplicate frames from a programming screencast.

Starting from the set of frames extracted from a video (one per second), we use SIFT to compare neighboring frames and remove the redundant or duplicate ones, leaving a set of frames that contain unique information. In order to determine which frames to keep, we first extract the key-points from each frame and obtain their feature vectors using SIFT. Then, given two neighboring frames f_1 and f_2, for each key-point $k_{f_1,i}$ in frame f_1, we find the best matching key-point $k_{f_2,j}$ and the second-best matching key-point $k_{f_2,l}$ in f_2 based on the Euclidean distance between their features. If the Euclidean distance between $k_{f_1,i}$ and $k_{f_2,j}$ is smaller than 75% of the distance between $k_{f_1,i}$ and the $k_{f_2,l}$ (i.e., the best matching point is significantly closer than the second-best match) then the pair of key-points $(k_{f_1,i}, k_{f_2,j})$ is considered a strong match and added to the set $m_{1,2}$ of matching key-point pairs for f_1 and f_2. This process is called a "ratio test" and was introduced by Lowe (2004). The threshold of 75% for the distance was determined empirically, based on testing different values. This process is repeated for all key-points in f_1, and at the end, the number of matched key-points in $m_{1,2}$ represents the similarity measure between frames f_1 and f_2 (i.e., the more similar key-points in $m_{1,2}$, the more similar the two frames are). In other words, we obtain the similarity percentage between two frames by dividing the total number of matched key-points by the total number of key-points.

To determine which frames to keep for a video, we employed the following procedure. Consider a video and its frames $V = \{f_1, f_2, \ldots, f_n\}$, and $s_{i,i+1}$ as the similarity value between a pair of consecutive frames f_i and f_{i+1}. We compare f_i with its successive frame f_{i+1}, and if $s_{i,i+1}$ is more than 80%, we keep f_{i+1}. We keep f_{i+1} rather than f_i, as we want the latest version of the frame which might contain an important piece of code that was added. At the end of this process for the entire video, what is left are frames that contain considerably different information. These are the set of frames we use in the next step of our approach.

3.2 Code Editing Window Localization

Our goal is to accurately detect the location of the main code editing window in videos, regardless of its size, location, background color, etc. This is an important restriction since the size and the location of the main code editing window is not the same for all videos and could also change within the video itself. As opposed to previous work which was susceptible to errors due to these variables (Ponzanelli et al. 2017; Yadid and Yahav 2016), our approach is based on robust feature extractors and object detectors as introduced in Section 2.

The deep architectures for object detection first propose regions of interest to identify possible object locations and then classify those regions. The regions of interest are proposed based on low dimensional feature maps generated by a set of convolutional layers. The choice of the feature map generating network affects the speed and accuracy of the final object detection. Different types of feature extraction networks, with their own advantages exist. These are mostly convolutional so as to maintain translational invariance.

When using more traditional supervised machine learning algorithms such as Naive Bayes, Supervised Vector Machines, Decision Trees, etc. input features need to be hand-engineered from the raw data, such that they describe the characteristics of the data points in a way that can help the learners discriminate between them. This leads to each data point being represented by a d-dimensional feature vector which, together with the class label of the data point are used as input for the machine learning algorithm. With the advent of deep learning techniques, however, the preliminary step of feature engineering is no longer needed, since deep neural networks have a large number of parameters distributed across many layers, allowing them to learn higher level features from the raw data itself through a series of multiple non-linear projections. Therefore such deep learning classifiers directly take in the raw data with just the class labels, without the need to extract any features from the data beforehand. In our case, the input data are the images themselves and their labels are the class name (e.g., Fully Visible Code) accompanied by the coordinates of the code editing window in the image. The code editing window is spatially located within an image (i.e., it has a set of coordinates within the image) and with the help of convolutional layers, the neural networks used in our approach can learn to extract relevant spatial features that map to the code editing window coordinates.

In this paper, we used five different configurations of object detectors and backbone feature extractor networks, which have been proven to perform the best in object detection tasks. Three of these configurations use region-free object detectors, while two use region-based ones.

The region-based approaches we use are Faster R-CNN with the Inception Resnet V2 feature extractor (Szegedy et al. 2016) and R-FCN with the Resnet-101 feature extractor (He et al. 2015). Our configurations for these are defined as follows. The *batch size* is the total number of training samples that will be passed through the network at once. The batch size can be adjusted based on the computer hardware performance such as the GPU memory as well as the input image size. We set the batch size to 16 for both networks based on the limitations of our hardware configuration. We then resized the input image dimensions to the default values of 600 and 1024 pixels. The resized image has the same aspect ratio for both models. One of the most important parameters that impact the training performance is the *optimizer*. The optimizer updates the weights after each epoch by backpropagation (one forward and backward pass of every training sample through the neural network). For general object detection, the recommended optimizer is the momentum, which is a version of the Stochastic Gradient Descent (SGD) (Qian 1999). Therefore, we trained the models using SGD with momentum optimizer. We used the default values for the hyper parameters of the network layers such as a stride value of 16.

The region-free approaches we use are: SSD with the Resnet50-FPN feature extractor (He et al. 2015; Lin et al. 2017), SSD with the Inception V2 feature extractor (Ioffe and Szegedy 2015), and YOLO with its own Darknet feature extractor (Redmon et al. 2015). The default configurations for these are as follows. We used a batch size of 24 for SSD. All input images to the SSD models were resized to have the default square aspect ratio of 640×640 and 300×300 for SSD with Resnet50-FPN and SSD with Inception V2, respectively. We kept the default optimizers for both SSD approaches, which are SGD with momentum and RMSprop. For YOLO, the input images were resized to the default 416×416 pixels. We trained the model on a total of 4,000 epochs since the per-batch average loss stopped oscillating after this. We used a batch size of 16 with subdivision of 8, which are the default values for the YOLO architecture. We used RMSprop gradient decent optimizer to minimize the loss of the network in the training process.

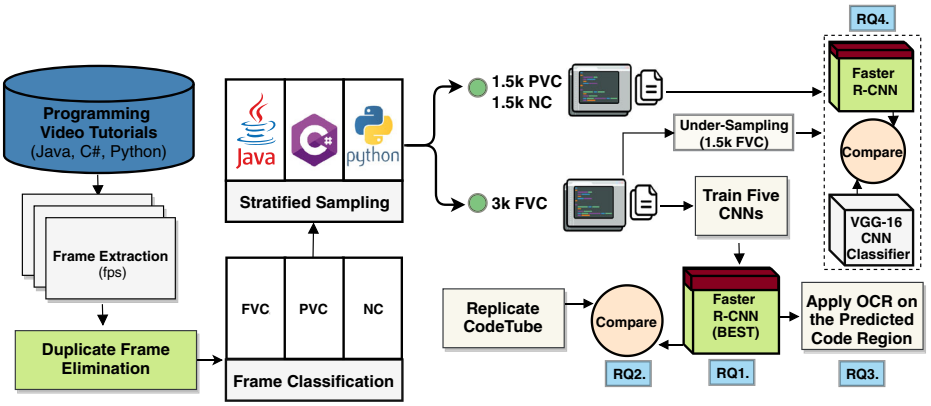


Fig. 3 An overview of our empirical evaluation

All our implementation is based on the Tensorflow API.[2] In Tensorflow, each object detector model is defined by a configuration file which is mainly used in the training process.

4 Empirical Evaluation

Figure 3 shows the general overview of our empirical evaluation. We started by manually collecting a set of programming video tutorials from YouTube, focusing on topics related to three programming languages, namely Java, C#, and Python. For each video, we extracted a frame every second. This resulted in a large set of frames, containing many duplicates. We therefore, applied our approach for duplicate frame elimination described in Section 3.1 in order to remove the redundant information and obtain a more diverse set of frames in our dataset. Each remaining frame was then manually classified into one of the three main categories: Fully Visible Code (FVC), Partially Visible Code (PVC), and No Code (NC). A total of 3,000 FVC frames, 1,500 PVC, and 1,500 NC frames were selected for the evaluation. Each frame was than annotated with the class name (FVC, PVC, or NC) and the bounding box information. We trained five different object detector configurations (as described in Section 3.2) to determine which models work the best in finding the code editing window inside a FVC frame. This answers our first research question. We then used the best model and compared its prediction accuracy with CodeTube (Ponzanelli et al. 2017) to answer RQ2. Additionally, we applied OCR to the predicted code region to show the percentage of noise that was removed using our approach and compared this with the noise reduction obtained by CodeTube (RQ3). Using the best model determined in RQ1, we also showed that we can successfully classify an input frame into one of the FVC, PVC, and NC categories. We compared our approach in this multi-classification task with a CNN-based classifier that was used by a previous work in classifying video frames (RQ4).

In the following subsections, we describe in detail our data collection process, the research questions we address and the results.

[2] https://github.com/tensorflow/tensorflow

4.1 Data Collection

4.1.1 Video selection

We created our evaluation dataset by manually selecting a diverse set of programming screencasts hosted on Youtube on topics involving three different programming languages, namely Java, C#, and Python. We generally looked for videos where the code is readable on the screen, without specifically restricting the quality of the videos to any particular threshold. We targeted videos whose length does not exceed 40 minutes, as a means to limit the number of frames considered from one single video in our dataset and increase diversity. In order to further increase variety and include videos from more authors, we limited the number of videos we collected from any given Youtube channel to five. When a channel we considered contained more than five eligible videos, we randomly selected videos from it one by one, until we found five that met our criteria. We further aimed to include videos covering a wide variety of topics for each programming language and also considered videos where the code was written in a variety of IDEs such as Netbeans, Pycharm, Visual Studio, IntelliJ, Atom, etc. and a variety of code editors such as vim, Notepad++, etc. Moreover, we included different background colors for each IDE and code editor (i.e., white and black). More specifically, to achieve diversity in the set of videos we considered, we searched on YouTube using a variety of queries that included: (i) the programming language name "Java", "Python" or "C#", (ii) one of the words "tutorial" or "lesson", (iii) optionally one of the words: "beginner", "intermediate", "advanced", (iv) optionally the name of an IDE or a code editor: "Eclipse", "Netbeans", "Intellij", "Pycharm", "Atom", "Visual Studio", "Notepad++", "vim", (v) and words referring to important programming concepts, such as "loop", "array", "lists", etc. We used these queries as needed to increase the diversity in our dataset gradually. The number of results obtained for each query varied, but we generally focused on the top results obtained for each query. It is generally very challenging to collect the same number of videos for each IDE for a particular programming language, as some IDEs are more common than others. For example, we noticed that in the screencasts on Youtube, (i) the majority of C# videos contain Microsoft Visual Studio, (ii) Eclipse and Netbeans are the most popular IDEs for Java, and (iii) PyCharm is the most popular IDE for Python. Furthermore, the majority of videos feature a white background in the code editing window since it is the default in many IDEs and the most common background color that developers use in screencasts. While we cannot claim we achieved complete parity between IDEs and background colors, while being also representative of the videos hosted on YouTube, we did our best to include videos featuring various IDEs and background colors for each language.

Another aspect we looked at when it came to IDEs was their layout. IDEs usually have at least three different sections: the main editing window, the file explorer and the output window. We made sure to find videos using differently sized and shaped parts in the layout. We also included videos that had different numbers of sections in their layout.

To keep track of all these properties of the videos, we created a simple database for storing this information about the selected videos. The two authors in charge of data collection kept this database up to date and checked it frequently to ensure diversity was being achieved as much as possible, given the limitations described above.

Two of the authors collected a dataset of 450 videos from YouTube following the guidelines mentioned above, with 150 videos for each of the three popular programming languages considered (Java, C#, Python). It took a total of about 18 hours to gather these videos. To ensure the validity of our predefined criteria for each video, and the diversity of

Table 2 Statistics about the collected videos

Programming language	Min.	Mean.	1st Qu.	Median	3rd Qu.	Max.
Java	73s	352s	205s	313s	424s	1789s
C#	86s	475s	251s	366s	590s	2215s
Python	129s	517s	321s	490s	652s	1773s

our dataset, the two authors validated each of the videos together after the initial collection process. During the validation process, the videos that did not adhere to the criteria were replaced with new ones.

After the video selection was completed, we used the youtube-dl[3] API tool to download the videos to our server. Then, we wrote a script based on the FFMPEG[4] tool to check the total number of seconds for each video. The detailed statistics about the collected videos are shown in Table 2. On average, Python has the highest number of seconds per video with a mean of 517s. In accordance with our criteria of keeping the length under 40 minutes, the maximum length was that of a C# video that was ∼37 minutes long. The total duration of the collected videos was relatively similar across the three programming languages.

4.1.2 Duplicate Frame Elimination

We extracted the frames of each of the 450 videos at the rate of one frame per second using the FFMPEG tool. This resulted, as expected, in many duplicate frames. Therefore, we applied our duplicate frame elimination approach (see Section 3.1) in order to reduce the redundant frames.

The results of the duplicate frame removal step are shown in Fig. 4. For example, the total number of extracted frames from Python programming tutorials was 77,535. During duplicate removal, more than 80% of the Python frames were found to be redundant and removed. The mean value of the number of frames kept per Python video was reduced from ∼517 to only ∼95, with a median of 77 as shown in Table 3. Only 14.8% and 17.4% frames were selected to be kept (i.e., were not duplicates) from Java and C#, respectively. Ultimately, the number of total extracted frames was significantly reduced from 201,574 to only 34,470 frames. Our dataset consists of only the selected (i.e., remaining, non-duplicate) frames which we use in our classification and annotation tasks.

Since previous work (Khandwala and Guo 2018; Zhao et al. 2019; Ponzanelli et al. 2017; Yadid and Yahav 2016) has used a different approach, namely pixel-by-pixel similarity to identify and remove frames containing duplicate information in programming screencasts, in the next few paragraphs we compare and contrast our approach based on SIFT to the pixel-by-pixel approach for detecting duplicate frames in two ways. We first provide a high-level discussion on the limitations of the pixel-by-pixel similarity approaches compared to SIFT and provide some examples to that respect. Then, in order to see how different the results provided by the two approaches really are in practice, we perform an analysis of the similarity and the differences between the set of remaining frames in the videos of our dataset after applying SIFT and pixel-by-pixel, respectively. One note is that, while there

[3] https://github.com/rg3/youtube-dl
[4] https://www.ffmpeg.org/

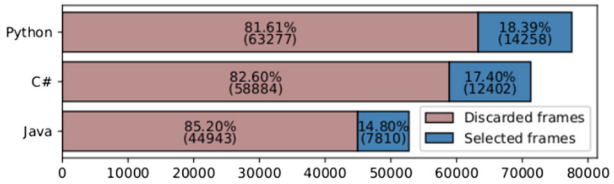


Fig. 4 The results after the duplicate frame elimination using SIFT

are several ways to compute pixel-by-pixel similarity between two frames, for our comparison we make use of the Structural SImilarity Index (SSIM) (Wang et al. 2004), which is the approach that has been used in previous work on analyzing programming screencasts (Khandwala and Guo 2018; Zhao et al. 2019).

Our decision to use SIFT rather than pixel-by-pixel approaches for detecting the similarity between frames is mainly motivated by the fact that pixel-by-pixel approaches are very sensitive to noise and lead to inaccurate results. There are two main situations that exemplify this, and we discuss each of them in detail below.

First, two frames containing the same code snippet, but at different locations on the screen should be recognized as being duplicates/identical. This kind of frame pairs are a very common occurrence in programming tutorials, since a tutor often scrolls up or down a code section in an IDE. As an example, in Fig. 5a and b, we have two frames with exactly the same code snippet, just shifted by four lines. Therefore, only one frame should be kept for analysis purposes, otherwise duplicate information will be extracted and processed. Since the pixel-by-pixel algorithm compares the value of each pixel in the first frame to the value of the pixel at the same position in the second frame, it will consider the two frames quite dissimilar, since the location of all the original pixels has changed (i.e., the tutor has scrolled-down). Consequently, the pixel-by-pixel SSIM similarity value between the first and the second frame is only 68% as shown in Fig. 5f, where a black pixel indicates a match and a white pixel indicates the two pixels do not match between the two frames. On the other hand, SIFT is invariant to translation and it does not compare the same pixel location but rather features of the images. Therefore, SIFT finds the matching features regardless of their pixels positions. An example of this is shown in Fig. 5e where a line is drawn between each pair of matched key-points between the two frames and a similarity of 91% is obtained with SIFT. Therefore, when using SIFT the first frame would be removed since it is very similar to the second one. The same frame, however, would not be removed when using SSIM.

Table 3 Statistics of our dataset after applying the duplicate frame elimination compared to the original number of the extracted frames

	Prog. lang.	Min.	Mean	1st Qu.	Median	3rd Qu.	Max.
Original	Java	73	352	205	313	424	1789
	C#	86	475	251	366	590	2215
	Python	129	517	321	490	652	1773
Selected	Java	1	52	18	32	57	505
	C#	8	83	35	59	112	582
	Python	3	95	48	77	127	451

The second situation when pixel-by-pixel approaches are challenged is the opposite of the first, namely when the code location does not change on the screen, but the code itself does. This can happen when the tutor switches between open files in the IDE and it is a common occurrence in programming screencasts. An example can be seen in Fig. 5c and d, where frame 4 contains a completely different code snippet than frame 3. Thus, the two frames should have a low similarity and thereby both frames should be kept. In the case of pixel-by-pixel similarity, however, the two frames are actually found to be extremely similar (93% SSIM similarity). This is due to the fact that the pixel-by-pixel approach finds all the white background pixels as well as those in the other areas of the IDE such a the file hierarchy and output to be matching (the black pixels in Fig. 5h). Since all pixels are given the same weight, the similarity found in the background trumps the relatively small dissimilarity found in the code snippet. SIFT, on the other hand uses key-points as the basic comparison unit and is not as influenced by the background. In fact, a similarity of only 61% is returned by SIFT, which indicates that we should keep both frames. If using pixel-by-pixel on the other hand, the 93% similarity means that the first frame would be incorrectly removed.

We now proceed to compare the two algorithms in practice, by applying the two algorithms separately on the frames in our dataset. Formally, for each V = $\{f_1, f_2, \ldots, f_n\}$ we compare each pair of consecutive frames f_i and f_{i+1} using SIFT and SSIM with a similarity threshold of 80%, meaning that a frame that has a similarity greater than 80% to its successor will be removed. The output of this phase is two sets, SIFT = $\{f_j, f_{j+1}, \ldots, f_s\}$ and SSIM = $\{f_k, f_{k+1}, \ldots, f_t\}$ for each video in our dataset, containing the frames remaining after SIFT and SSIM have been applied, respectively. Since the total number of videos in our study is 450, we have a total of 450 sets for SIFT and SSIM each.

Our aim is to investigate the similarity and the differences between the remaining frames using SIFT and SSIM. We compute the similarity between the two sets for all videos using the Jaccard Index (Jaccard 1912) as shown in Eq. (1). A value between 0.0 and 1.0 is returned in which 1.0 means the two sets are identical. As shown in Table 4, we compute the average Jaccard Index between the SIFT remaining frames and the SSIM remaining frames

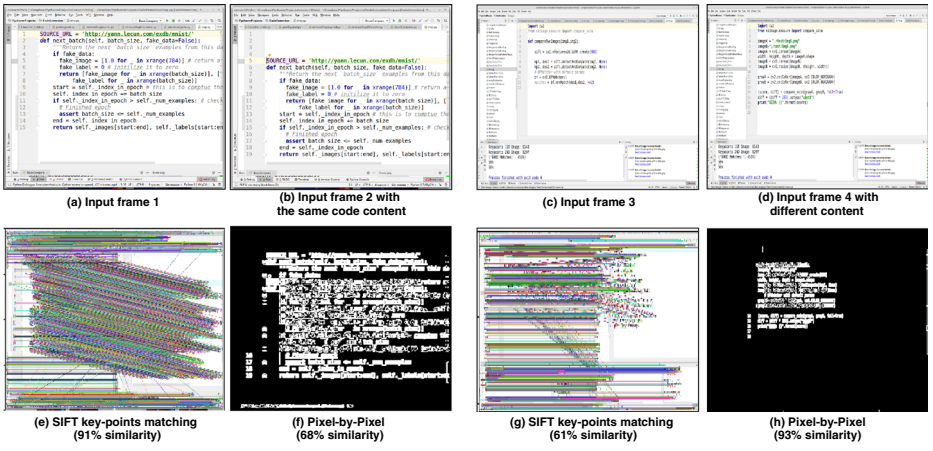


Fig. 5 Results of comparing two frames that contain identical code (frame 1 and frame 2) and two frames containing completely different code (frame 3 and frame 4) using SIFT and the Pixel-by-Pixel (SSIM) algorithms

Table 4 A comparative analysis between the remaining number of frames after applying SIFT and SSIM on our dataset

Prog. Lang.	Jacc(SIFT,SSIM)	| SIFT-SSIM |/|SIFT|	| SSIM-SIFT |/|SSIM|
Java	0.14	0.84	0.05
C#	0.18	0.81	0.05
Python	0.11	0.87	0.04

for each programming language. On average, the similarity between the two sets are only 18%, 14% and 11% for C#, Java, and Python, respectively. In light of this low similarity, we were also interested to know how many of the frames in each of the two sets are unique to that set. The last two columns in Table 4 show these numbers. Specifically, 81-87% of the frames retained by SIFT were only retained by SIFT and not SSIM. On the other hand, only 4-5% of the frames retained by SSIM were only retained by SSIM and not by SIFT. This indicates that a lot of the frames removed by SSIM were retained by SIFT, likely due to the second situation described above, where two frames are considered very similar by SSIM due to the match in their background, not in the actual code being displayed (see Fig. 5c,d, and h for an example), but SIFT can capture the difference in the code and keep both frames. In total, there were 34,470 frames kept by SIFT across our dataset, while only 4,979 were kept by SSIM. While SSIM leads to a much greater reduction in analysis time by reducing the amount of frames to be analyzed, we argue that this can also lead to a significant loss in meaningful information, as described above. We therefore argue that SIFT is better suited to analyze programming screencasts, where the changes in the appearance of frames can be subtle due to the background color and other sections of the IDE remaining the same between frames, even though completely different code may be shown.

$$Jaccard(SIFT, SSIM) = \frac{|SIFT \cap SSIM|}{|SIFT \cup SSIM|} = \frac{|SIFT \cap SSIM|}{|SIFT| + |SSIM| - |SIFT \cap SSIM|} \quad (1)$$

4.1.3 Manual Frame Classification and Annotation

As we are interested in the frames that contain code, in our next step we manually classified all the frames remaining after removing duplicates (34,470 frames) into three categories:

- **Fully Visible Code (*FVC*).** A frame contains fully visible code if the main editing window is not obstructed in any way and every line of code is clear and readable.
- **Partially Visible Code (*PVC*).** A frame contains a partially visible code if the main editing window is partially obstructed by another window, such as code completion suggestions.
- **No Code (*NC*).** A frame is considered to contain no code if it does not contain any fully visible nor partially visible code. These could be frames from a video where the tutor is explaining a topic on slides or board, is searching in their browser, or introduction and exit scenes, etc.

To make the data classification process more manageable, we created a web-based classifier which displayed only a frame at a time and provided radio buttons for each of the FVC, PVC and NC categories, which a person classifying the image could select. Navigation buttons allowed the person doing the classification to move to the previous or to the next image. The tool stores the classification results in an SQLite database.

The data classification task was performed by the first two authors. Each author classified the remaining frames from half (225) of the videos and confirmed the classes selected by the other author for the other half of the videos. For the frames where there was a disagreement upon the selected class, the authors reached an agreement after a discussion. The frames that needed discussion were all involving partially visible code. For example, there were a few cases where the cursor was blocking a small part of the code (i.e., usually one character). These frames were initially classified as FVC by one author and PVC by the other author. After a discussion, the two authors agreed to consider these frames as FVC since the missed area was insignificant and the missing characters could be easily guessed or reconstructed. However, when the disagreement involved frames with a small popup box that blocks part of the code, after consultations, the authors agreed to consider these as PVC since reconstructing the code from these frames would be challenging. The overall agreement level between the two authors in selecting the classes was $\sim$ 99.7.%

The detailed classification results are shown in Table 5. Since our goal is to determine the code region on the screen that will allow the correct extraction of code, we are particularly interested in the FVC class which contributes 65%, 47%, and 62% to the overall dataset for Java, C#, and Python, respectively. In particular, in order to obtain the needed training data for our code editing window detection approach, we need to manually annotate FVC frames with the coordinates of their code editing windows. Since this is a more intensive task than just classifying the frames into FVC, PVC, and NC, annotating the whole set of FVC frames was unfeasible. Instead, we decided to sample a set of 3,000 FVC frames from the total of 19,704 and use this subset for the annotation phase.

In order to ensure diversity and uniformity in our FVC frame sample, which is important when training machine learning approaches, we applied a form of stratified sampling as follows. First, to ensure diversity across programming languages, we decided to include 1,000 frames for each of the three programming languages in our sample. Then, for each programming language, we included a balanced number of frames from each video in that set and selected the frames to include from each video randomly. In more details, given a set of videos for a programming language $V = \{v_1, v_2, \ldots, v_{150}\}$, for each v_i, we select a random FVC frame f_i from it and add it to the selected set. Once a frame has been included from each video, we repeat the process, until the size of the selected set equals 1,000 FVC frames for that programming language. Thus, our dataset consists of 3,000 FVC frames randomly picked from each video of every programming language.

Once the 3,000 FVC frames were selected, we proceeded to manually annotate each of them with the location of their corresponding code editing window. We define the location of a code editing window in a frame as a *bounding box* (*x, y, w, h*), where (*x, y*) is the center of the quadrilateral or box where the code is located and *(w, h)* are the width and height of the box. The bounding box is often represented in terms of (*x, y, w, h*) in object detection literature (Redmon et al. 2015). We used a cloud online image annotation service,

Table 5 Classification results (FVC=Fully Visible Code, PVC=Partially Visible Code, NC=No Code)

Prog. language	FVC frames	PVC frames	NC frames	Total	FVC/Total
Java	5,079	667	2,064	7,810	0.65
C#	5,799	1,311	5,292	12,402	0.47
Python	8,826	1,590	3,842	14,258	0.62
Total	19,704	3,568	11,198	34,470	

Dataturks,[5] for enabling the annotation of the frames with this information. A total of ten Computer Science graduate students participated in the annotation process. We recorded a video that explains the process of the annotation with a few examples and shared it with all of our participants. A total of 300 FVC frames were assigned to each participant. Since each frame was annotated only once, one of the authors manually verified each annotation and re-annotated the frame that had inaccurate bounding box (e.g., any frames that did not cover the code entirely, or covered more than only the code). Less than 20 frames out of the 3,000 required re-annotation. The annotation process involved showing one frame at a time, after which a bounding box could be drawn on the frame to delimit the code section. The location of the bounding box identified by the participants was then automatically determined and saved as (x, y, w, h) as explained above. The participants had an option to skip annotating a frame for any reason, such as if it was not clear for them where to draw the box. We make our dataset available online.[6]

4.2 Research Questions, Methodology, and Results

4.2.1 RQ1. Which deep architecture model performs the best at localizing the code editing window in video programming tutorials?

Motivation: The main purpose of this work is to propose an approach that is able to precisely locate the code editing window regardless of the programming language, the code editing tool used, the location and the size of the code region, etc. This research question aims to experimentally determine which of the five deep architectures presented in Section 3.2 is best at performing this task.

Methodology: To address this question, we trained the five deep object detectors with the annotated FVC frames. In the object detection literature, it is common to train the network using only images from the positive class. For example, if the goal is to identify cars in images, an object detector would be trained only with images that contain cars. The detector will then simply fail when an object (e.g., a car) cannot be identified in an image, such as in the case of images belonging to the negative class. This is done out of practicality since there can be an infinite number of examples in the negative class (e.g., an infinite number of pictures without cars in them) and therefore training on all types of images in this class is impossible and not needed. In our case, the positive class is represented by the FVC frames annotated with the location of a code editing window. Thus, we trained the five different network architectures on the FVC frames and their corresponding main code editing windows. We split the dataset into training, validation, and testing sets. The validation and the testing sets are 10% of the dataset each, while the remaining 80% of the 3,000 annotated FVC images were used for training. We performed a 10-fold cross-validation in our experiments. While the validation sets were mainly used to tune the model hyperparametes and avoid overfitting, the testing sets used to evaluate the final model on unseen data. Each fold had different test images that were randomly generated and were different from the validation sets and the images in the other folds. We computed the training and prediction time for each model during our experiments. We conducted all the experiments using Tensorflow on a machine with an Intel Xeon 3.40GHz processor, 128GB RAM, and a GeForce GTX 1080 GPU with 8 GB of memory.

[5] https://dataturks.com/

[6] http://malahmadi.sa/roi/

Evaluation metrics: *Intersection over Union (IoU)* is a metric commonly used for measuring the success of an object detection task and has been widely used in several competitions such as PASCAL VOC Challenge, ImageNet Large Scale Visual Recognition Challenge, and MS COCO (Everingham et al. 2010; Russakovsky et al. 2015; Lin et al. 2014). IoU measures how well the predicted object area matches the area of the ground truth by dividing the intersection of the two areas to their union. In our case, the ground truth is represented by the area selected by the participants to contain the code editing window, while the prediction of the code editing window is given by the five deep learning architectures we employ. While IoU gives an indication of how good a prediction of a code editing window is, it does not give a clear indication of whether to consider a prediction successful or not. For this purpose, we use IoU thresholds and consider a prediction successful if its IoU is above the threshold. For example, setting the IoU threshold to 0.8 means that we consider any prediction that achieves an IoU higher than 0.8 when compared to the ground truth as a successful prediction. Using IoU thresholds allows us to compute further performance metrics such as average precision and accuracy (defined below).

Average Precision (AP) is another metric we use to report the success of the predictions. We report AP at different IoU thresholds, as it is the standard in different competitions and the state-of-the-art work in object detection (Dai et al. 2016; Shrivastava and Gupta 2016). Each object detector predicts the location of the code box with a confidence score. We compute the AP by sorting the confidence scores in descending order. Then, for each predicted code box, we consider the prediction to be true if its IoU is greater than the current IoU threshold. We compute the precision for each prediction starting from the image with the highest confidence score. In this context, the precision is the ratio of the correctly predicted bounding boxes to the total number of predictions. As the AP might vary based on the chosen IoU threshold, we compute the AP at different IoU thresholds (0.5 to 0.9 with a step size of 0.1).

The *Accuracy* is the last metric we report for this research question. It is measured as the total number of correct predictions divided by the total number of predictions. We also report the accuracy at different IoU thresholds.

Results: Table 6 shows the AP at different IoU thresholds for each model tested. In total, each model has been trained and tested 10 times using 10-fold cross validation. The results are based on 50 different experiments for each of the five object detectors, which took a total of 71 hours to run. As shown in Table 6, the average precision and the accuracy for all models are relatively high at the .5, .6, and .7 IoU thresholds. This means that each model is reliably predicting the code editing window with an IoU of at least 50% (i.e., at least a 50% overlap with the ground truth). We can also observe that when we increase the IoU thresholds gradually, the performance of all approaches starts to significantly decrease, with the exception of Faster R-CNN and SSD-Resnet. This shows the importance of evaluating each model at different IoU thresholds, where we can finally find the best model that could reliably predict a code editing window with a high overlap with the ground truth. At an IoU threshold of 0.9, Faster R-CNN maintains an AP of 93.3% with 94.4% accuracy. This is indicative that Faster R-CNN is the best performing model for our dataset.

Additionally, when we fed our 300 frames into the models to make the prediction in each fold, we computed the prediction time and averaged the result. Although YOLO performed the worst in terms of AP and accuracy for prediction compared to other models, the prediction time was the fastest with an average of $\sim$ 6 fps as shown in Table 6. In general, the

Table 6 The results of 10-fold cross validation in terms of Average Precision (AP) and Accuracy (Acc.) at different Intersection over Union (IoU). Also, the prediction time for each input image in seconds

Detector	Feature extractor	IoU@.5		IoU@.6		IoU@.7		IoU@.8		IoU@.9		Prediction time (s)
		AP	Acc.	AP	Acc.	AP	Acc.	AP	Acc.	AP	Acc.	
Faster R-CNN	Inception Resnet(V2)	97.7	98.1	97.6	98.1	97.5	98.0	97.1	97.8	93.3	94.4	1.2
R-FCN	Resnet-101	98.5	98.8	98.2	98.8	98.0	98.5	97.2	92.7	66.4	76.3	0.85
YOLO	Darknet	97.5	98.1	94.2	96.3	84.9	90.0	51.0	66.3	3.4	15.7	0.11
SSD	Resnet50 FPN	98.7	99.1	98.0	98.7	97.1	97.7	95.5	96.0	87.3	88.9	0.77
SSD	Inception (V2)	99.1	99.4	89.9	99.3	98.6	98.8	94.2	95.9	31.4	50.5	0.73

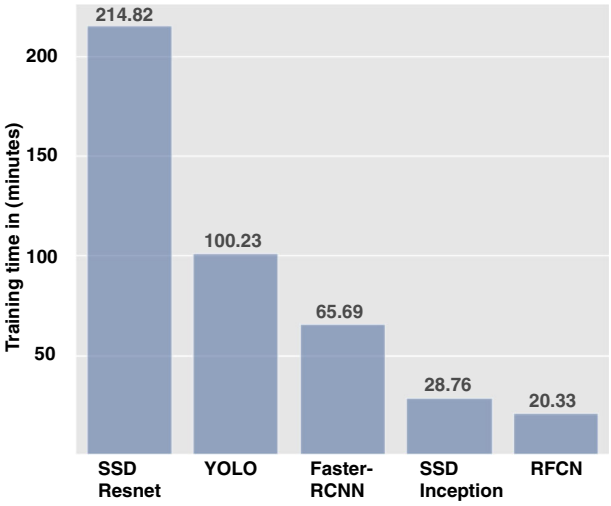


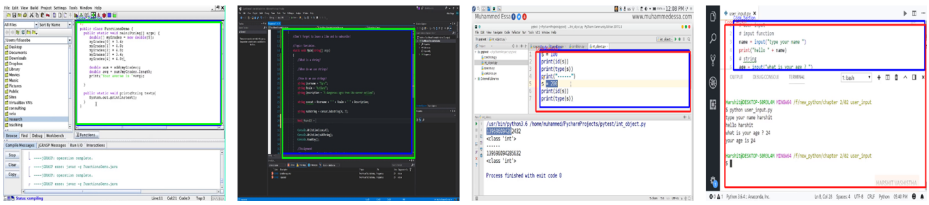
Fig. 6 Training time in (minutes) for each object detector we used during our 10-fold cross validation

prediction times of YOLO and SSD are faster than those of Faster R-CNN and R-FCN since they are region-free approaches. Since for the scope of this research the accuracy is more important than the speed, we chose Faster R-CNN which is the most accurate approach and can still make a code editing window prediction at a rate of $\sim$ one fps.

Figure 6 shows the training time of the models using the 10-fold cross validation for a total of 4,000 iterations where all models have converged. The choice of the backbone network contributes toward the training time as we can see with SSD using Resnet with 101 layers and SSD using Inception. We observed that Faster R-CNN took about an hour during the training process. Szegedy et al. (2016) shows that training with residual connections accelerates the training of Inception networks significantly and proposed Inception-ResNet-v2 that we utilized as a feature extractor for Faster R-CNN. Another important factor that plays an important role in the training performance is the localization task. While R-FCN is faster to train, it is less accurate compared to Faster R-CNN.

Figure 7 shows four examples of code box predictions made by Faster R-CNN compared to the ground truth bounding boxes. The blue bounding box represents the ground truth and the green bounding box is our model's correct prediction. We generally noticed that Faster R-CNN remains very accurate even with different combinations of IDE layouts we tested. For example, in Figs. 7a and b, the prediction and ground truth are almost exact matches. In Fig. 7c, the prediction achieves an IoU of 86%, but still manages to cover almost the same code section, and nothing else. However, due to the high IoU threshold of 90%, it is marked as an incorrect prediction, since it falls below the threshold. For the last Fig. 7d, the model incorrectly predicted both the code bounding box and the terminal bounding box as a single code bounding box. The color of the terminal and its format looked similar to the code bounding box, so we assume this is what made the model to incorrectly predict the code bounding box. More example predictions can be found online in our replication package.[7]

[7] http://malahmadi.sa/roi/



(a) IoU = 0.97 (b) IoU = 0.93 (c) IoU = 0.86 (d) IoU = 0.25

Fig. 7 Ground truth bounding box (in blue) compared to predicted bounding box (in green) for correct prediction or (in red) for incorrect prediction

4.2.2 RQ2. How does our approach compare to previous work that identified the code editing window in video frames?

Motivation: Finding the code editing window/bounding box is the first step towards extracting source code from video tutorials. Previous work has mostly made use of simple edge detection techniques for this purpose, which have known limitations that limit their application (more details in Section 5). CodeTube (Ponzanelli et al. 2017) is the only previously proposed approach that not only employs an edge detection technique, but also offers an alternative approach for the cases when edge detection fails. Therefore, we compare the performance of our approach based on Faster R-CNN with CodeTube.

Methodology: We replicated the approach proposed by CodeTube (Ponzanelli et al. 2017) for locating the code bounding box as follows. First, using edge detection based on OpenCV,[8] we detected a number of quadrilaterals in a code frame. Then, we applied OCR on each quadrilateral using Tesseract-OCR[9] (as done in CodeTube) and found the one that contains code in it. The result of this process is accurate when the main code editing window is detected as a quadrilateral. For many of the frames, however, the code editing window is not detected, especially when the bounding box is unclear. In this case, we applied the second approach proposed by CodeTube: we divided the image into fixed-size sub-images, applied OCR on them and analyzed the results. The top-left and bottom-right sub-images with at least one English and/or Java keywords defined the overall code region.

We compared the replicated approach from CodeTube with our best model from **RQ1**, namely Faster R-CNN. In other words, we compared the code bounding boxes produced by the two approaches to our testing dataset and observed which ones are closer to the ground truth. Since the CodeTube approach was defined and evaluated only on Java programming videos in previous work (Ponzanelli et al. 2017), we only use the Java FVC frames in our testing set (100 frames in total) for this part of our evaluation, for a fairer comparison.

Evaluation metrics: We used the IoU metric between the predicted code section and the ground truth code section for both our approach and CodeTube. We then report the median IoU across all 100 Java testing frames for our approach as well as for CodeTube.

Results: As shown in Fig. 8, our approach significantly outperformed CodeTube. While the median IoU was only 0.4 for CodeTube (0.48 average IoU), our approach achieved a median IoU of 0.95 (0.93 average IoU). When analyzing the results in more detail,

[8] https://opencv.org/

[9] https://github.com/tesseract-ocr

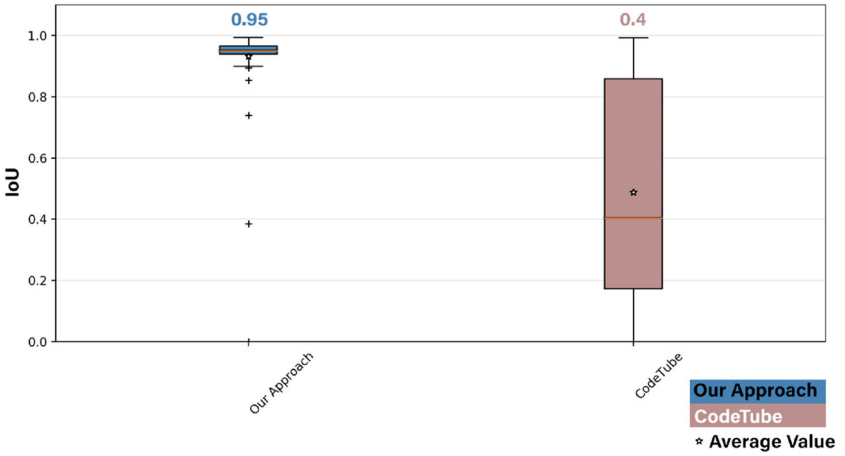


Fig. 8 Boxplots of the IoU metric for our approach and CodeTube with the median value reported above each boxplot

we noticed that using the CodeTube algorithm, 50 code regions were detected using the first approach (code edge detection), while the other 50 were detected using the second heuristic approach as the code edge detection approach had failed.

The CodeTube heuristic approach has some limitations that yield low accuracy. Basically, the fixed-size sub-images could cover only part of the code, that could lead to truncated identifiers and English words. Since CodeTube looks for Java or English words, the sub-image containing truncated code would not be valid. This could be resolved using a sliding-window approach with a predefined stride width and height. However, this would be computationally expensive as we would need to apply OCR on all sub-images. On the other hand, using our approach, we apply OCR to only one region (i.e., the code editing window). Our approach uses supervised learning which requires a label for each frame. This label is the code bounding box annotation, that we need to prepare before training the model. Unlike CodeTube, this process is time-consuming since it requires human effort to make the annotation. Yet, this is only a one-time process and once the model is built, it can be used for as many predictions as needed. Additionally, our approach is able to make a prediction at a rate of $\sim$ one fps as shown in Table 6. CodeTube requires $\sim$ 12 seconds to make a prediction for each frame. This is mainly because CodeTube applies OCR to 25 different boxes to make the prediction (i.e., 20% of the original width and height).

4.2.3 RQ3. Does our approach help OCR techniques to better extract code from video programming tutorials?

Motivation: The ultimate goal of our approach is to help OCR approaches extract code better from programming video tutorials by reducing the noise that would otherwise be extracted. In this research question, we investigate this assumption and study whether applying OCR on the code editing window predicted by our approach leads to better results compared to applying OCR on the entire frame. Additionally, we aim to compare the results of the code extraction from the bounding box detected by our approach with the one detected by CodeTube.

Methodology: To answer this question, we apply OCR on each frame in our testing dataset consisting of 300 FVC frames from Java, C#, and Python video tutorials. OCR is applied four times on each frame, namely:

- On the ground truth code bounding box. OCR is applied on the true location of the code bounding box in the frame and the result is stored as OCR_{GT}.
- On the predicted code bounding box. OCR is applied on the code bounding box predicted by our best approach, Faster R-CNN and the result is stored as $OCR_{Predicted}$.
- On the entire code frame. OCR is applied on the entire frame and the result is stored as OCR_{Entire}.
- On the CodeTube-predicted code frame. OCR is applied on the code bounding box predicted by CodeTube using the approach discussed in Section 4.2.2, and the result is stored as $OCR_{CodeTube}$.

We then compare how close $OCR_{Predicted}$, $OCR_{CodeTube}$, and OCR_{Entire} each come to the ground truth OCR_{GT}.

For this part of our evaluation we used Google Cloud Vision API for OCR,[10] which, unlike Tesseract-OCR used in CodeTube, does not need image preprocessing before extracting a clean text. Google API has also been shown to outperform Tesseract-OCR in previous studies (Moslehi et al. 2018).

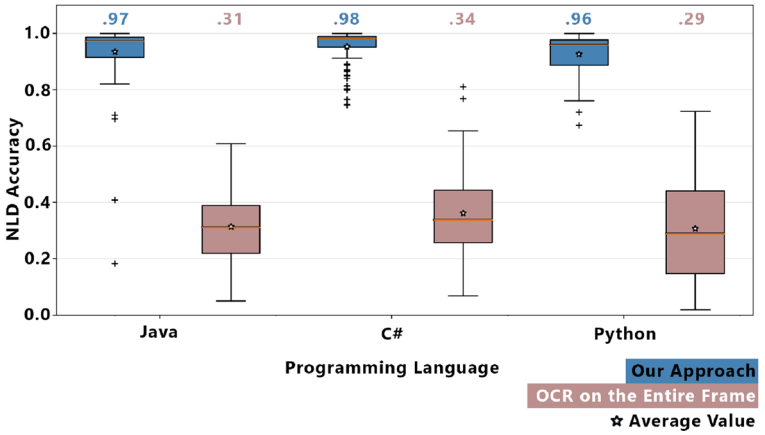
Evaluation metrics: We used the Levenshtein Distance (LD) metric to measure the similarity between OCR_{GT} and $OCR_{Predicted}$ as well as the dissimilarity between OCR_{GT} and OCR_{Entire} for each of the test frames. Additionally, we used LD to compare the dissimilarity of OCR_{GT} with both $OCR_{Predicted}$ and $OCR_{CodeTube}$. The Levenshtein distance measures the required number of edit operations (insertion, substitution, and deletion) to transform one string to another. The higher the Levenshtein distance, the more dissimilar the two texts are. However, it is hard to gauge what the exact value of the Levenshtein distance means. Therefore, we use the Normalized Levenshtein Distance (NLD), defined in Eq. (2), which instead indicates how similar two texts are to each other as a value between zero and one. An NLD of one indicates that two texts are identical.

$$NLD(t_1, t_2) = 1 - \frac{LD(t_1, t_2)}{max(len(t_1), len(t_2))} \tag{2}$$

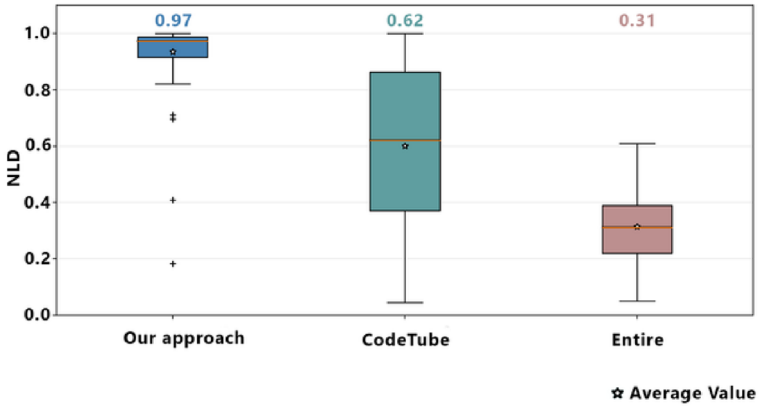
The Normalized Levenshtein Distance (NLD) has been used before in several fields including software engineering to gauge similarity between texts. Sun (2015) and Thummalapenta et al. (2010). When OCR_{GT} is exactly the same as $OCR_{Predicted}$, their NLD is one. The NLD is computed between each OCR_{GT} to each $OCR_{Predicted}$, $OCR_{CodeTube}$, and OCR_{Entire}. We used a total of 300 frames from our test set to compare $OCR_{Predicted}$ and $OCR_{CodeTube}$ to OCR_{Entire}. Since the approach and study in the CodeTube paper (Ponzanelli et al. 2017) focused only on Java programming, we excluded Python and C# from our evaluation, for a fair comparison (i.e., only the 100 Java frames were used from our test set).

Results: From the boxplots in Fig. 9, it is clear that $OCR_{Predicted}$ outperformed OCR_{Entire} and $OCR_{CodeTube}$, as its similarity to the ground truth is significantly higher, reaching a median of 97% for Java, 98% for C# and 96% for Python. On the other hand, OCR_{Entire} achieved a median similarity to the ground truth of only 31% for Java,

[10] https://cloud.google.com/vision/



(a) Results of 300 frames (Java, C#, Python)



(b) Results of 100 Java frames

Fig. 9 Boxplots of NLD with the median above each boxplot shown (i) in (blue) for NLD($OCR_{Predicted}$, OCR_{GT}), (ii) in (red) for NLD(OCR_{Entire}, OCR_{GT}), (iii) in (green) for NLD(OCR_{Entire}, $OCR_{CodeTube}$)

34% for C# and 29% for Python as shown in Fig. 9a. Although, CodeTube reduced a median of 32% noise comparing to OCR_{Entire}, the dissimilarity between $OCR_{CodeTube}$ to OCR_{GT} is about 38% on average. This indicates that the process of detecting the code bounding box by CodeTube is not as accurate as our approach. Overall, this shows that on average, applying our code localization approach based on Faster R-CNN to the frame and then applying OCR only on the area predicted by our approach leads to a 66% reduction in noise in the extracted code compared to applying OCR on the entire frame.

4.2.4 RQ4. Can our approach be used for discriminating between FVC, PVC, and NC frames?

Motivation: While our duplicate frame elimination step manages to remove a lot of the noise in a programming screencast, during the manual classification of frames, we

noticed that a lot of them contain only Partially Visible Code (PVC) due to the obstruction of the code editing window by other windows such as popups. These frames are not useful in the process of correctly extracting code and would therefore be best if they were automatically identified and removed from the analysis, instead of having to manually identify them. Since our Faster R-CNN approach proved to be very successful in identifying particular windows in a frame (i.e., the code editing window), we want to investigate in this research question, if the same approach could also be used to identify popups obscuring the code (or in other words to identify the PVC frames). Additionally, the frames that do not contain any written code should also be detected and removed, since they are not useful in localizing and extracting code in the first place. So, we want to train our model to also learn the features of the frames that belong to the no code (NC) class, and identify them correctly. We also compare our approach to VGG-16, an algorithm used in previous work for the detection of FVC, PVC, and NC frames (Ott et al. 2018a).

Methodology: We considered this problem as a multi-class classification problem where the three possible classes are Fully Visible Code (FVC), Partially Visible Code (PVC), and No Code (NC). Previous work has addressed this particular problem using a classifier based on VGG-16 (Ott et al. 2018a), which is a pre-trained network and robust feature extractor that won the ILSVRC-2014 competition in classification (Simonyan and Zisserman 2014). However, we also believe that our approach based on Faster R-CNN has the potential to perform very well and maybe even outperform VGG-16 on this task. While Faster R-CNN was used in our RQ1 for predicting the location of the code editing window, what was not maybe straightforward is that Faster R-CNN also implicitly classifies each object inside the predicted box. We can therefore use Faster R-CNN as a classifier and compare it with the VGG-16 classifier used in previous work.

Our dataset contains frames from three different classes: FVC, PVC, and NC. Since we had already annotated 3,000 FVC frames from all the videos we collected for each programming language, we downsampled the total number of FVC frames to 1,500 to bring it to the same number of frames as PVC and NC. The downsampling process was mainly used to balance our dataset across all classes. We used a similar approach to the one described in Section 4.1.3. Basically, we randomly chose one FVC frame from each video and repeated this process again by selecting another random frame until we had a total of 500 FVC frames for each programming language. A total of 1,500 PVC frames and NC frames were randomly selected using this same stratified sampling approach. The PVC frames were then also equally distributed using the Dataturks annotation tool to a total of 10 participants, which annotated each PVC with the box defining the obstructing window. All annotations were verified by one of the authors who re-annotated the ones with an inaccurate bounding box or the skipped ones (i.e., an annotator could skip annotating a frame). While in the case of FVC and PVC frames, it is relatively straight-forward what needs to be annotated (i.e., the bounding box of the code and of the obstructing window, respectively), annotating the NC frames presents a few options. There are three main approaches we envisioned for training the model with the negative class (NC class in our case). The first approach would be to train the model with only positive examples (FVC and PVC frames) and the model would just return a True Negative when asked to classify NC frames with no visible code. In the second approach, all three classes (FVC, PVC and NC) would be used to train the model and we would just annotate the NC frames with zeros as the bounding box coordinates. The third approach is similar to the second approach but rather than setting the bounding box to zeros, we consider the entire image as a single bounding box and annotate the NC frames with its coordinates (i.e., the

bounding box is the entire image). Although using the first approach we do not need to collect negative examples for training, we need to set a confidence threshold to identify the best class that describes the detected object if any. A value less than a pre-defined threshold would indicate that there was no object in the image, hence, it belongs to the negative class (e.g., NC class). Identifying the best threshold needs several experiments and is based on heuristics that might fail in some cases. Using the second approach would work the same way as the first one, since the model would not learn anything from an all-zeros bounding box. The third approach, however, would allow the model to explicitly learn the spatial features of the NC frames and make a prediction accordingly without using any heuristic such as a confidence threshold. Additionally, annotating these frames would not require any human effort since a script could automatically generate the annotation for each of these frames with the bounding box information. Eventually, our model should output the class name and the bounding box information for each input frame where we use the bounding information only for the frames with the FVC and the class name to identify the PVC and the NC frames.

For our competitor, VGG-16, we also used our dataset with the three classes: FVC, PVC, and NC. The label for each image is only the class name since VGG-16 is basically a classifier that uses the features of the entire image. VGG-16 was trained using back-propagation, and a sophisticated gradient decent optimizer along with techniques to prevent overfitting. In particular, we monitor the validation loss in each epoch, and stop the training when the validation loss is not improving after a total of five epochs. Additionally, we save the best model that achieves the best performance on the validation set during the training of our network. Decreasing the learning rate over time improves the network performance as the network would "fine-tune" the learned weights (Russakovsky et al. 2015). Therefore, we automatically reduce the learning rate by a factor of 1×10^{-1} if we observe that the validation loss is decreasing for a total of 3 epochs. We experimentally found that by fine-tuning our networking using these techniques, the overall classification performance for the model increased. The VGG-16 network was implemented using Python with Keras[11] API.

We split our dataset into training (80%), validation (10%), and testing (10%). Both VGG-16 and Faster R-CNN were trained using 10-fold cross-validation for the described multi-class classification task.

Evaluation metrics: To evaluate the performance of classifying FVC, PVC, and NC frames, we used the following standard metrics in our experiment (Zimmermann et al. 2007).

Precision measures the ability of the classifier to correctly identify positive samples. In particular, it measures the percentage of correctly classified frames in a specific class over all frames classified as that class. Formally, it is defined as $P = \frac{T_p}{T_p+F_p}$, where T_p is the number of true positives and F_p is the number of false positives. A high precision indicates that a classifier is returning a small number of false positives.

Recall, also known as True Positive Ratio, measures the percentage of correctly classified frames in a specific class over the actual number of frames in that class (e.g., the total number of correct FVC predictions over the total number of FVC frames). It is computed as $R = \frac{T_p}{T_p+F_n}$, where T_p is the number of true positives and F_n is the number of false negatives.

[11] https://github.com/keras-team/keras

F-score is a measure that uses a combination of precision and recall to indicate the effectiveness of the classifier. Formally, F_1 score is the harmonic mean of precision and recall, defined as $F = 2 \cdot \frac{P \cdot R}{(P+R)}$.

Accuracy measures the proportion of correctly classified frames. Formally, *Accuracy* is defined as $Acc = \frac{T_p+T_n}{T_p+F_n+F_p+T_n}$.

Results: Table 7 shows the results of our 10-fold cross-validation experiments using the VGG-16 and Faster R-CNN models. We observed that both models achieved a high precision and recall for predicting the frames that belong to the NC class. Since the FVC frames and the PVC frames look very similar to each other, the accuracy for predicting these frames is less than that of the NC frames. Faster R-CNN outperformed VGG-16 in predicting the frames that belong to the FVC and PVC classes, with an F-Score of .96 and .95, respectively. The recall score of the PVC class using VGG-16 is .86, which means that the model classified some PVC as a false negative (i.e., only 1,291 out of 1,500 were correctly classified as a PVC). In order to better understand where VGG-16 failed in the classification of PVC frames, we analyzed the results in detail and found that the problems were in the cases of dark IDE backgrounds or when there was not enough contrast between the obstructing popup window and the IDE background. Faster R-CNN, on the other hand, was able to better handle these cases. Not only was the overall accuracy of Faster R-CNN better than VGG-16, but also Faster R-CNN can determine the exact location of the code region in the FVC frames. Therefore, Faster R-CNN is more practical for our problem since we can classify and localize at the same time.

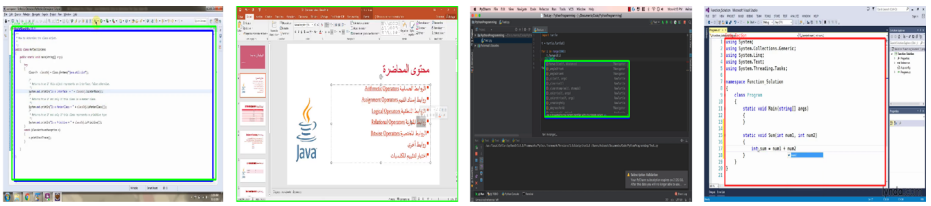
In Fig. 10, we show some of the prediction examples that are made by our model for the frames that belong to the NC and PVC classes. The content of Figs 10a and b are a FVC frame and a PowerPoint slide, respectively and were correctly classified by our approach into the FVC and NC categories, respectively. Figure 10c shows an example of predicting the PVC frame where the model correctly classified and located the obstructing window. We also show an example of incorrect prediction in Fig. 10d where the model incorrectly classified the input frame as a FVC frame instead of PVC, since the size of the popup box was very small.

4.3 Threats to Validity

The threats to *internal validity* in our study include the two main tasks for labeling our data set. First, two of the authors classified the remaining video frames in our dataset after removing the duplicates. Each author had to classify frames from a set of 225 videos. There is a chance that some of the frames were incorrectly labeled. To mitigate this threat, each

Table 7 The classification results of the 10-fold cross-validation experiments for the Fully Visible Code (FVC), Partially Visible Code (PVC), and No Code (NC) classes using VGG-16 and Faster R-CNN

Class	VGG-16			Faster R-CNN		
	Precision	Recall	F-Score	Precision	Recall	F-Score
FVC	0.86	0.91	0.88	0.94	0.98	0.96
PVC	0.90	0.86	0.88	0.99	0.91	0.95
NC	0.98	0.99	0.99	0.95	0.99	0.97
Overall accuracy		0.92			0.96	



(a) Correct FVC prediction (b) Correct NC prediction (c) Correct PVC prediction (d) Incorrect FVC prediction

Fig. 10 An example of prediction results for the frames with FVC, NC, and PVC using Faster R-CNN

author had to verify the frames labeled by the other author from the other 225 videos. The authors discussed any conflict regarding the labels and reached a common agreement. Second, for each of the FVC frames and PVC frames, we needed the ground truth code bounding box. Therefore, we recruited 10 computer science students to annotate the frames with this information. Sometimes, the participants may not be sure about the location of the code to annotate, so they might draw an inaccurate bounding box. To mitigate this issue, we had an option for them to skip a frame. For each skipped frame, one of the authors later annotated it. Additionally, we created a simple video demonstration about the process of annotating frames with a few examples for our participants. Lastly, all the annotated frames were verified by one of the authors.

Construct validity is threatened in our study by the measurements we used to answer our four research questions. In general, we mitigated this threat by using well-established metrics from the fields of object identification in images and image classification (Dai et al. 2016; Shrivastava and Gupta 2016; Russakovsky et al. 2015). For measuring the accuracy of our approach for **RQ1**, we had to carefully set a reasonable IoU threshold. A low IoU threshold increases the overall average precision and accuracy, while a high one can drastically reduce them. In our evaluation process, we followed the standard procedure from the literature in the field and computed the average precision and accuracy at different IoU thresholds.

For **RQ2**, we only used the 100 Java frames from our testing set in order to have a fair comparison with the previous work we compared against, CodeTube (Ponzanelli et al. 2017), which was proposed and evaluated only for Java screencasts. We also made sure to apply the same OCR engine that was used in CodeTube in the step for finding the code region.

The construct threats in **RQ3**, concern the evaluation metrics we used to show the similarity between two texts. The similarity between two texts could be measured using several metrics, based on character or token. We used the Normalized Levenshtein Distance string metric, which has been previously used in software engineering for measuring text similarity.

In our experiments to answer **RQ4**, the construct threat is mainly related to the effectiveness of the performance metrics that were used for the classification task. We mitigated the threat of selecting the measurement metrics by using well-established measurements in the machine learning field. That is, we evaluated our approach by using *Precision*, *Recall*, *F-Score*.

Regarding the threats to *external validity*, our results may not be generalizable to all the software development videos available or all of the code editing windows present in them. However, we aimed to make our approach and findings more generalizable by including

videos covering three different programming languages, different IDEs and background colors, various layouts, etc., for a total of 450 videos. To ensure even more diversity, we also limited the maximum number of selected videos from any channel to five.

5 Related Work

Nowadays, social media plays an essential role in developers' daily tasks, as they use it to communicate with other developers, acquire new information and skills, solve problems, etc. Storey et al. (2014). Programming screencasts, such as those hosted on YouTube are becoming more and popular and recent studies have started analyzing and leveraging this source of documentation. MacLeod et al. (2015) and MacLeod et al. (2017) performed a set of interviews with developers in order to study their motivation for creating programming screencasts and found that sharing knowledge they gained while performing a programming task was their main goal. Bao et al. (2017) analyzed the content of screen-captured videos to produce time-series HCI data automatically. Bao et al. (2018) proposed an approach to record the workflow of programming screencasts and display it to the watchers. Consequently, a timeline of the workflow operations are displayed with an option to navigate to a specific action. Moslehi et al. (2018) proposed an approach to link source code files to the corresponding screencasts by leveraging the GUI text as well as the audio transcripts. Ellmann et al. (2017) analyzed a set of videos from YouTube and found that video programming tutorials are more static than other types of videos, namely that the content shown on screen changes less often. This indicates that, when analyzing programming screencasts, one important performance gain may be to first remove the duplicated information that appears across several consequent frames in the video. Based on this previous work, we designed the first step of our approach (described in Section 3.1) to remove duplicate frames.

Other related works aim to create a useful resource from video contents and comments. As there are millions of programming video tutorials on YouTube, tagging video content is beneficial for developers to find the intended video. Escobar-Avila et al. (2017) and Parra et al. (2018) proposed an automatic tagging system for software development video tutorials. Comments could also provide video creators with useful information when accurately analyzed. But not all the comments are useful for the narrators; therefore, Poché et al. (2017) proposed an approach based on machine learning to classify comments into relevant and irrelevant.

The most related works to our research are divided into three topics and discussed in details below: Duplicate Frame Elimination in Programming Screencasts, Code Bounding Box Detection and Code Extraction from Videos, and Identifying Partially Visible Code in Video Frames.

5.1 Duplicate Frame Elimination in Programming Screencasts

Previous work on analyzing programming video tutorials has made use of pixel-by-pixel metrics to compute the similarity between frames in order to determine duplicate information to remove. Yadid and Yahav (2016), uniformly sampled the frames from 40 videos at a rate of 30 frames per video. Then the authors discarded the frames that do not contain typed code. While the frames could be sampled at longer periods of time to reduce redundancy, this would inevitably lead to the loss of valuable information, as some of the frames skipped may contain new material. Ponzanelli et al. (2017) compared every two consecutive frames

using pixel matrices and removed one of the frames when the similarity was less than a certain threshold. Comparing two consecutive frames after extracting one frame per second ensures the diversity of the frame selection process. Thus, we used the same approach in our frame extraction and duplicate elimination processes. However, we used a different approach for comparing one frame to another, described in Section 3.1. Moslehi et al. (2018) applied a simple textual comparison between each subsequent frame. While this approach can successfully compare two frames that are textually rich, it will likely fail for frames with more visual and less textual content. Other works are summarized in Table 8 and mostly use pixel-based comparison between frames.

Pixel-based or pixel-by-pixel comparison between frames is very sensitive to noise and leads to inaccurate results. For example, a part of the code might be highlighted which would result in differences between the pixels in the frames (i.e., before and after the text highlighting). Also, when a cursor location changes, the pixel values of the cursor would change from one location to another. Unfortunately, pixels are not invariant of transformation, unlike features. There are several other cases such as image resolution, color changing, etc. that can add more noise to pixels. This is the primary reason that Moslehi et al. (2018) changed their approach from pixel-based comparison to textual comparison.

To avoid the limitations of pixel-by-pixel approaches, we leverage Scale Invariant Feature Transform (SIFT) which has been experimentally approved by a wide variety of applications to be very successful for extracting features (Lowe 1999, 2004). Although video resolution could affect the performance of SIFT in terms of extracting features, SIFT resolves this issue by firstly selecting a set of candidate key-points and then filtering out the ones with low contrast or those that were poorly localized. The remaining key-points are robust to noise, illumination, and occlusion. Additionally, we use SIFT to remove duplicates among neighboring frames from within a video and not across multiple videos. We assume that the tutors do not alter the color scheme or other visual properties of the IDE during the presentation. In other words, the resolution and the color scheme/contrast used by the IDE / source code editor are very likely to remain the same for neighboring frames extracted from the same video. SIFT is successfully able to resolve the scaling and rotation challenges of an image. In video programming tutorials, it is common that a tutor may have zoomed into a specific area of code to explain it, and this problem is addressed by the invariant scale property of SIFT. SIFT outperformed several feature detector algorithms and was applied to several object recognition applications, Juan and Gwun (2009). Section 2 describes the SIFT algorithm in more details.

5.2 Code Bounding Box Detection and Code Extraction from Videos

Most of the video programming screencasts involve a narrator writing code on-the-fly in a code editor such as an IDE. These IDEs have some windows in the form of quadrilaterals which can be detected using an edge detection algorithm such as Canny Edge Detector,

Table 8 Papers that analyzed the similarity between videos frames

Paper	Total Tutorials	Frame Rate	Frame Comparison
Ponzanelli et al. (2017)	150	one fps	Pixel-by-Pixel
Ellmann et al. (2017)	100	10 fps	Pixel-by-Pixel
Moslehi et al. (2018)	10	one fps	Textual-Based
Ott et al. (2018a)	40	one fps	Pixel-by-Pixel

Canny (1986). Therefore, several approaches used edge detection techniques along with some heuristics to find the main code editing window, as summarized in Table 9.

Yadid and Yahav (2016) applied a Canny Edge Detection algorithm to extract a set of contours from an IDE frame. The smallest contour that covers most of the code is assumed to be the main code region. The authors state that finding the main editing window is a challenging task; therefore they find the location of the main editing window for one frame of a video using Canny Edge Detection and use its location to extract the other code sections from the remaining frames. While this is convenient and computationally efficient, it can also be incorrect in many situations, when the window is moved, split, resized, overlapped, etc.

Recent work was proposed by Khandwala and Guo (2018) to automatically extract source code from screencasts and display it in a web-based tool. The approach combines different code sections shown in a video by finding the differences between frames and merging the code when the editing window is scrolled. Due to the fact that the edge-based algorithm that finds the main code segment is not very accurate, the code extraction process was poor. The authors resolved this issue by introducing a custom edge detector algorithm based on seven heuristic steps. However, these heuristics are not always applicable and are subject to error. We resolve this issue by applying a deep feature extractor on each image and train a model with images and their corresponding code segment locations. This does not rely on any heuristics and is generally applicable.

Ponzanelli et al. (2017) proposed CodeTube, which is a tool that enables developers to write a query, and relevant video fragments will be retrieved along with other related StackOverflow discussions. CodeTube identified the code editing window using a two-part approach. First, the authors used a tool to find all quadrilaterals of an IDE frame. In many cases the code section can be identified as a quadrilateral. If the first approach fails, the authors use a second approach based on frame segmentation. A frame is divided into sub-frames such that each sub-frame is 20% of the width and height of the original image. Then, OCR is applied to each sub-frame. A sub-frame is marked as a Java code if it contains a Java keyword and/or at least one English term. Finally, the minimum (x, y) and the maximum (x, y) of the sub-frames that contain code are considered the bounding box of the code section.

We experimentally found that edge detection algorithms have several limitations that would fail in detecting the code editing window in many cases. For example, there have to be explicit box boundaries to be detected by the edge detector. There are several cases where box horizontal and vertical lines are not clearly visible in videos frames. Furthermore, the code sometimes is not written on a quadrilateral which is a polygon with four edges. Figure 11 shows the result of applying Canny Edge Detector on a frame extracted from a video tutorial. The code bounding box was not detected in this case, as was the case in many other frames. On the other hand, our approach was successfully able to detect the code bounding box as shown in Fig. 11.

Table 9 Papers that find a code region in screencasts

Paper	# of Tutorials	Find code frame	Find code region
Yadid and Yahav (2016)	40	Manual	Edge Detector
Ponzanelli et al. (2017)	150	OCR-Based	Edge Detector
Khandwala and Guo (2018)	20	OCR-Based	Edge Detector
Our Approach	450	CNN	CNN+Localize

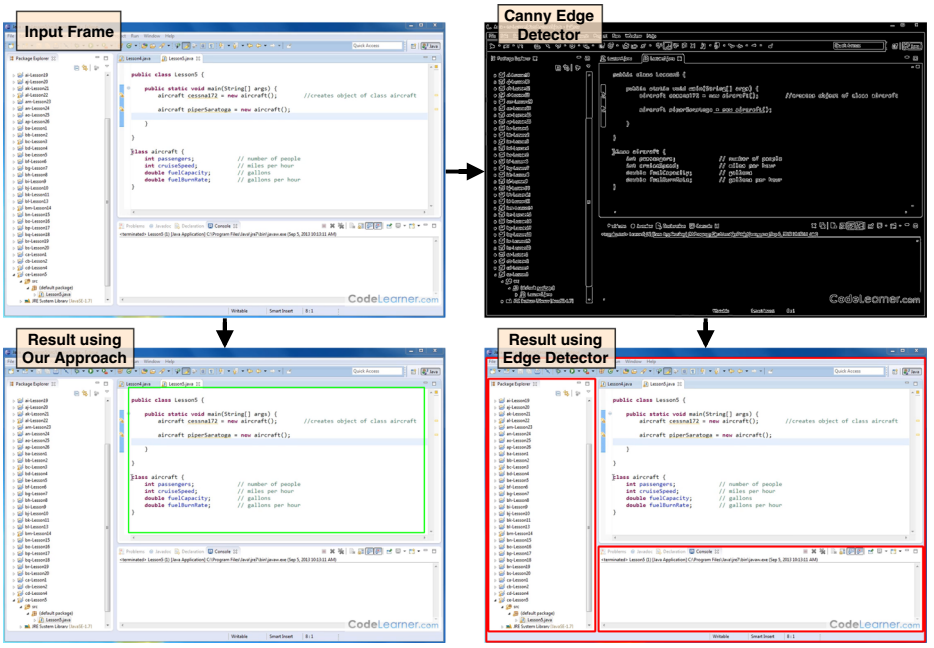


Fig. 11 Comparison between approaches in finding the code editing window

5.3 Identifying Partially Visible Code in Video Frames

Ott et al. (2018a) used a pre-trained neural network VGG-16 to classify frames from 40 Java programming videos into four categories. Two of these categories are Fully Visible Code (FVC) and Partially Visible Code (PVC). Their main goal is to classify the entire frame as containing code or not. On the other hand, our goal is not only classifying the frame but also detecting the accurate location of the code snippet inside the frame. This is related to our work as we aim to classify FVC and PVC to answer our **RQ4**. We used the VGG-16 classifier employed by Ott et al. (2018a) as a baseline to compare our Faster R-CNN approach against and found that VGG-16 is outperformed on all metrics.

A follow-up work by the same authors is to recognize the programming language based on a VGG network (Ott et al. 2018b). Java and Python programming languages were used in their experiment.

6 Conclusion

In this paper, we proposed a novel approach to improve the accuracy and drastically reduce the noise of code extraction based on OCR from programming screencasts by localizing the main code editing window inside the code frames. Our approach is based on advanced object detection models that are trained to automatically predict the main code editing window. We evaluated our approach on a set of frames extracted from videos on three different programming languages: Java, C#, and Python. A total of 450 videos were collected in our study.

We trained five neural networks architectures for our classification and localization tasks for finding the code region and found that Faster R-CNN performs the best, achieving an accuracy of 94% on average.

We also extracted the source code using OCR from a set of 300 frames and showed that applying our approach leads to a 66% reduction in noise and a 96% match with the ground truth source code.

Our long-term goal is to be able to extract the code from the predicted code region correctly. Our future work will move in this direction, focusing on correcting the noise caused by the OCR extraction and also improving upon previous work in merging code appearing in different video fragments in order to obtain a correct and complete program. We then aim to create a web-based system that analyzes programming screencasts and correctly extracts the complete embedded source code and makes it available for download.

Acknowledgements Mohammad Alahmadi was sponsored in part by the University of Jeddah. Abdulkarim Khormi was sponsored in part by Jazan University. Sonia Haiduc was supported in part by the National Science Foundation under Grant No. 1846142.

References

Alahmadi M, Hassel J, Parajuli B, Haiduc S, Kumar P (2018) Accurately predicting the location of code fragments in programming video tutorials using deep learning. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE'18. ACM Press, Oulu, pp 2–11. https://doi.org/10.1145/3273934.3273935. http://dl.acm.org/citation.cfm?doid=3273934.3273935

Bao L, Li J, Xing Z, Wang X, Xia X, Zhou B (2017) Extracting and analyzing time-series hci data from screen-captured task videos. Empir Softw Eng 22(1):134–174

Bao L, Xing Z, Xia X, Lo D (2018) VT-Revolution: Interactive programming video tutorial authoring and watching system. IEEE Transactions on Software Engineering, https://doi.org/10.1109/TSE.2018.2802916. http://ieeexplore.ieee.org/document/8283605/

Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR (2009) Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '09. ACM, New York, pp 1589–1598. https://doi.org/10.1145/1518701.1518944

Canny J (1986) A computational approach to edge detection. Ieee Transactions on Pattern Analysis and Machine Inteligence, pp 679–698

Dai J, Li Y, He K, Sun J (2016) R-FCN: Object detection via region-based fully convolutional networks. arXiv:160506409 [cs]

Ellmann M, Oeser A, Fucci D, Maalej W (2017) Find, understand, and extend development screencasts on youtube. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics, ACM, pp 1–7

Escobar-Avila J, Parra E, Haiduc S (2017) Text retrieval-based tagging of software engineering video tutorials. In: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17). IEEE, Buenos Aires, pp 341–343

Everingham M, Van Gool L, Williams CK, Winn J, Zisserman A (2010) The pascal visual object classes (voc) challenge. Int J Comput Vis 88(2):303–338

Felzenszwalb PF, Huttenlocher DP (2004) Efficient graph-based image segmentation. Int J Comput Vis 59(2):167–181

Girshick R (2015) Fast R-CNN. arXiv:150408083 [cs]

Girshick R, Donahue J, Darrell T, Malik J (2013) Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv:13112524 [cs]

Grzywaczewski A, Iqbal R (2012) Task-specific information retrieval systems for software engineers. J Comput Syst Sci 78(4):1204–1218

He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. arXiv:151203385 [cs]

Hu W, Huang Y, Li W, Zhang F, Li H (2015) Deep convolutional neural networks for hyperspectral image classification. J Sensors 2015:258,619–258,619. https://doi.org/10.1155/2015/258619

Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, et al. (2017) Speed/accuracy trade-offs for modern convolutional object detectors. In: IEEE CVPR, vol 4

Ioffe S, Szegedy C (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:150203167 [cs]

Jaccard P (1912) The distribution of the flora in the alpine zone. 1. New Phytologist 11(2):37–50

Juan L, Gwun O (2009) A comparison of sift, pca-sift and surf. International Journal of Image Processing (IJIP) 3(4):143–152

Khandwala K, Guo PJ (2018) codemotion: expanding the design space of learner interactions with computer programming tutorial videos. In: Proceedings of the Fifth Annual ACM Conference on Learning at Scale - L@S '18. ACM Press, London, pp 1–10. https://doi.org/10.1145/3231644.3231652. http://dl.acm.org/citation.cfm?doid=3231644.3231652

Kim KH, Hong S, Roh B, Cheon Y, Park M (2016) PVANET: Deep but lightweight neural networks for real-time object detection. arXiv:160808021

LeCun Y, Haffner P, Bottou L, Bengio Y (1999) Object recognition with gradient-based learning. In: Shape, Contour and Grouping in Computer Vision. Springer, London, pp 319–345. http://dl.acm.org/citation.cfm?id=646469.691875

Lin TY, Maire M, Belongie S, Bourdev L, Girshick R, Hays J, Perona P, Ramanan D, Zitnick CL, Dollár P (2014) Microsoft coco: Common objects in context. arXiv:14050312 [cs]

Lin TY, Dollár P, Girshick R, He K, Hariharan B, Belongie S (2017) Feature pyramid networks for object detection. In: CVPR, vol 2

Liu W, Anguelov D, Erhan D, Szegedy C, Reed S, Fu CY, Berg AC (2016) SSD: Single shot multibox detector. 9905:21–37, arXiv:151202325 [cs], https://doi.org/10.1007/978-3-319-46448-0_2

Lowe DG (1999) Object recognition from local scale-invariant features. In: Proceedings of the Seventh IEEE International Conference on Computer Vision, vol 2, IEEE, pp 1150–1157. https://doi.org/10.1109/ICCV.1999.790410. http://ieeexplore.ieee.org/document/790410/

Lowe DG (2004) Distinctive image features from Scale-Invariant keypoints. Int J Comput Vis 60(2):91–110

MacLeod L, Storey MA, Bergen A (2015) Code, camera, action: How software developers document and share program knowledge using youtube. In: Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC'15), Florence, pp 104–114

MacLeod L, Bergen A, Storey MA (2017) Documenting and sharing software knowledge using screencasts. Empir Softw Eng 22(3):1478–1507. https://doi.org/10.1007/s10664-017-9501-9. https://link.springer.com/article/10.1007/s10664-017-9501-9

Mikolajczyk K, Schmid C (2005) A performance evaluation of local descriptors. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(10):1615–1630

Moslehi P, Adams B, Rilling J (2018) Feature location using crowd-based screencasts. In: Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18. ACM Press, Gothenburg, pp 192–202. https://doi.org/10.1145/3196398.3196439. http://dl.acm.org/citation.cfm?doid=3196398.3196439

Ott J, Atchison A, Harnack P, Bergh A, Linstead E (2018a) A deep learning approach to identifying source code in images and video. In: Proceedings of the 15th IEEE/ACM Working Conference on Mining Software Repositories, pp 376–386

Ott J, Atchison A, Harnack P, Best N, Anderson H, Firmani C, Linstead E (2018b) Learning lexical features of programming languages from imagery using convolutional neural networks

Parra E, Escobar-Avila J, Haiduc S (2018) Automatic tag recommendation for software development video tutorials. In: Proceedings of the 26th Conference on Program Comprehension, ACM, pp 222–232

Poché E, Jha N, Williams G, Staten J, Vesper M, Mahmoud A (2017) Analyzing user comments on youtube coding tutorial videos. In: Proceedings of the 25th International Conference on Program Comprehension, IEEE Press, pp 196–206

Ponzanelli L, Bavota G, Mocci A, Di Penta M, Oliveto R, Hasan M, Russo B, Haiduc S, Lanza M (2016a) Too long; didn't watch!: Extracting relevant fragments from software development video tutorials. ACM Press, pp 261–272, https://doi.org/10.1145/2884781.2884824. http://dl.acm.org/citation.cfm?doid=2884781.2884824

Ponzanelli L, Bavota G, Mocci A, Di Penta M, Oliveto R, Russo B, Haiduc S, Lanza M (2016b) codetube: Extracting relevant fragments from software development video tutorials. In: Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE'16). ACM, Austin, pp 645–648

Ponzanelli L, Bavota G, Mocci A, Oliveto R, Di Penta M, Haiduc SC, Russo B, Lanza M (2017) Automatic identification and classification of software development video tutorial fragments. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2017.2779479. http://ieeexplore.ieee.org/document/8128506/

Qian N (1999) On the momentum term in gradient descent learning algorithms. Neural Netw 12(1):145–151
Redmon J, Divvala S, Girshick R, Farhadi A (2015) You only look once: Unified, real-time object detection. arXiv:150602640 [cs]
Ren S, He K, Girshick R, Sun J (2015) Faster R-CNN: Towards real-time object detection with region proposal networks. arXiv:150601497 [cs]
Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M et al (2015) Imagenet large scale visual recognition challenge. Int J Comput Vis 115(3):211–252
Shrivastava A, Gupta A (2016) Contextual priming and feedback for faster R-CNN. In: Leibe B, Matas J, Sebe N, Welling M (eds) Computer Vision – ECCV 2016, vol 9905. Springer International Publishing, Cham, pp 330–348. https://doi.org/10.1007/978-3-319-46448-0_20. http://link.springer.com/10.1007/978-3-319-46448-0_20
Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. arXiv:14091556 [cs]
Storey MA, Singer L, Cleary B, Figueira Filho F, Zagalsky A (2014) The (R) Evolution of social media in software engineering. In: Proceedings of the on Future of Software Engineering, FOSE 2014. ACM, New York, pp 100–116. https://doi.org/10.1145/2593882.2593887
Sun Y (2015) A comparative evaluation of string similarity metrics for ontology alignment. Journal of Information and Computational Science 12(3):957–964. https://doi.org/10.12733/jics20105420. http://www.joics.com/publishedpapers/2015_12_3_957_964.pdf
Szegedy C, Ioffe S, Vanhoucke V, Alemi A (2016) Inception-v4, inception-resnet and the impact of residual connections on learning. arXiv:160207261 [cs]
Thummalapenta S, Cerulo L, Aversano L, Di Penta M (2010) An empirical study on the maintenance of source code clones. Empir Softw Eng 15(1):1–34. https://doi.org/10.1007/s10664-009-9108-x. http://link.springer.com/10.1007/s10664-009-9108-x
Uijlings JR, Van De Sande KE, Gevers T, Smeulders AW (2013) Selective search for object recognition. Int J Comput Vis 104(2):154–171
Wang Z, Bovik AC, Sheikh HR, Simoncelli EP et al (2004) Image quality assessment: from error visibility to structural similarity. IEEE Trans Image Process 13(4):600–612
Yadid S, Yahav E (2016) Extracting code from programming tutorial videos. In: Proceedings of the 6th ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'16). ACM, Amsterdam, pp 98–111
Zhao D, Xing Z, Chen C, Xia X, Li G, Tong SJ (2019) Actionnet: Vision-based workflow action recognition from programming screencasts. In: Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)
Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the 3rd IEEE International Workshop on Predictor Models in Software Engineering (PROMISE'07), Washington, pp 9–15

Mohammad Alahmadi is currently a Ph.D. candidate in the Computer Science Department at Florida State University (FSU), Tallahassee, USA. He has received his M.Sc. in Computer Science from FSU and B.Sc. in Information Technology from King Abdul-Aziz University (KAU), Jeddah, Saudi Arabia. His current research interests include multimedia software documentation, software engineering, applications of deep learning and computer vision, mining of software repositories. Previously, he worked as a programmer in the General Authority of Civil Aviation (Jeddah, Saudi Arabia) before he joined Jeddah University to work as a teaching assistant, where he received a full scholarship to pursue his degree.

Abdulkarim Khormi is currently a Ph.D. student in the Computer Science Department at Florida State University (FSU). He received an M.Sc. degree in Computer Science from FSU and a B.Sc. degree in Computer Science from Jazan University (Jazan, Saudi Arabia).

Biswas Parajuli currently works as a Data Scientist at Cognitive GeoInterpretation Inc. He was born in Nepal and received his B.Tech in Computer Science (CS) degree from National Institute of Technology (Durgapur, India), and his M.S. and PhD degrees in CS from Florida State University (USA).

Jonathan Hassel is currently a Software Engineer for WebstaurantStore. He has a B.Sc. in Computer Science from Florida State University. His research interests include multimedia software documentation and machine learning.

Sonia Haiduc is an Associate Professor in the Computer Science Department at Florida State University. She received her Ph.D. and M.Sc. degrees from Wayne State University in 2013 and 2009, respectively and her B.Sc. degree from the Babes-Bolyai University in 2006. Her research interests are in software engineering, and in particular in software maintenance and evolution, program comprehension, and software documentation. Her research has been published in top journals and conferences in the field of software engineering and she is the recipient of several NSF grants, including the NSF CAREER Award. Her research has received several awards, such as the ACM Distinguished Paper Award and Most Influential Paper Award and her students have been awarded Gold and Silver Medals in ACM Graduate Research Competitions. She is also actively involved in the organization and review process of several conferences and journals.

Piyush Kumar is an Associate Professor in the Computer Science Department at Florida State University. He obtained his Ph.D. from Stony Brook University in 2004 and his undergraduate degree from IIT Kharagpur in 1999. His research has been supported by NSF, AFOSR, AMD, NASA, and FSU. He is part of the FSU Database and Compustat groups and is a senior member of the ACM. In 2007, Dr. Kumar received the NSF CAREER Award and, in 2010, he received the AFOSR Young Investigator Award. Dr. Kumar's research is primarily on the boundary of algorithms and the real world. His interests lie in applying rich theory of algorithms to the domains of artificial intelligence, computational geometry, computer graphics, pattern recognition, and machine learning. Dr. Kumar's work narrows the gap between theory and practice.

Affiliations

Mohammad Alahmadi[1] · Abdulkarim Khormi[1] · Biswas Parajuli[1] · Jonathan Hassel[1] · Sonia Haiduc[1] · Piyush Kumar[1]

Abdulkarim Khormi
khormi@cs.fsu.edu

Biswas Parajuli
parajuli@cs.fsu.edu

Jonathan Hassel
hassel@cs.fsu.edu

Sonia Haiduc
shaiduc@cs.fsu.edu

Piyush Kumar
piyush@cs.fsu.edu

[1] Florida State University, 600 W College Ave, Tallahassee, FL 32306, USA