A Study on the Accuracy of OCR Engines for Source Code Transcription from Programming Screencasts

Abdulkarim Khormi[1,2], Mohammad Alahmadi[1,3], Sonia Haiduc[1]
[1]Florida State University, Tallahassee, FL, United States
{ khormi, alahmadi, shaiduc } @cs.fsu.edu
[2]Jazan University, Jizan, Saudi Arabia
[3]University of Jeddah, Jeddah, Saudi Arabia

ABSTRACT

Programming screencasts can be a rich source of documentation for developers. However, despite the availability of such videos, the information available in them, and especially the source code being displayed is not easy to find, search, or reuse by programmers. Recent work has identified this challenge and proposed solutions that identify and extract source code from video tutorials in order to make it readily available to developers or other tools. A crucial component in these approaches is the Optical Character Recognition (OCR) engine used to transcribe the source code shown on screen. Previous work has simply chosen one OCR engine, without consideration for its accuracy or that of other engines on source code recognition. In this paper, we present an empirical study on the accuracy of six OCR engines for the extraction of source code from screencasts and code images. Our results show that the transcription accuracy varies greatly from one OCR engine to another and that the most widely chosen OCR engine in previous studies is by far not the best choice. We also show how other factors, such as font type and size can impact the results of some of the engines. We conclude by offering guidelines for programming screencast creators on which fonts to use to enable a better OCR recognition of their source code, as well as advice on OCR choice for researchers aiming to analyze source code in screencasts.

CCS CONCEPTS

• **Software and its engineering** → *Documentation*; • **Computer vision** → *Image recognition*;

KEYWORDS

Optical Character Recognition, Code Extraction, Programming video tutorials, Software documentation, Video mining

ACM Reference Format:
Abdulkarim Khormi[1,2], Mohammad Alahmadi[1,3], Sonia Haiduc[1]. 2020. A Study on the Accuracy of OCR Engines for Source Code Transcription from Programming Screencasts. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387468

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387468

1 INTRODUCTION

Nowadays, programmers spend 20-30% of their time online seeking information to support their daily tasks [6, 11]. Among the plethora of online documentation sources available to them, programming screencasts are one source that has been rapidly growing in popularity in recent years [17]. For example, YouTube alone hosts vast amounts of videos that cover a large variety of programming topics and have been watched billions of times. However, despite their widespread availability, programming screencasts present some challenges that need to be addressed before they can be used by developers to their full potential. One important limitation is the lack of support for code reuse. Given that copy-pasting code is the most common action performed by developers online [6], programming screencasts should also strive to support this developer behavior. However, very few programming screencast creators make their source code available alongside the video, and online platforms like YouTube do not currently have support for automatic source code transcription. Therefore, developers often have no choice but to manually transcribe the code they see on the screen into their project, which is not only time consuming, but also cumbersome, as it requires pausing-and-playing the video numerous times [23].

Researchers have recently recognized this problem and proposed novel techniques to automatically detect, locate, and extract source code from programming tutorials [1, 2, 14, 19, 20, 24, 32, 34]. While different approaches to achieve these goals were proposed, they all have one component in common: an *Optical Character Recognition (OCR) engine*, which is used to transcribe the source code found in a frame of the screencast into text. To ensure the most accurate extraction of source code from programming screencasts, one needs to carefully choose the OCR engine that performs the best specifically on images containing code. However, no empirical evaluation currently exists on the accuracy of OCR engines for code transcription. Previous work that analyzed programming screencasts [14, 25, 32] simply picked one OCR engine, namely Tesseract[1], and applied it to programming screencast frames. However, without validating its accuracy or comparing it to other OCR engines on the task of code transcription, there is no evidence that Tesseract is the best choice for transcribing source code. Choosing a suboptimal OCR engine can lead to noise that is difficult to correct, and therefore code that is hard to reuse without expensive post-processing steps or manual fixing. To this end, there is a need for an empirical evaluation that examines the accuracy of different OCR engines for extracting source code from images, with the goal of finding the engine with the lowest risk of code transcription error.

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

In this paper, we address this need and present a set of experiments evaluating six OCR engines (Google Drive OCR, ABBYY FineReader, GOCR, Tesseract, OCRAD, and Cuneiform) applied to images of code written in three programming languages: Java, Python, and C#. We evaluate the OCR engines on two code image datasets. The first one contains 300 frames extracted from 300 YouTube programming screencasts (100 videos per programming language, one frame per video). The evaluation on this dataset is relevant to researchers working on the analysis of programming screencasts and software developers wanting to reuse code from screencasts. The experiments on this dataset allow us to determine the best OCR engine for source code extraction from screencasts, therefore reducing the risk of error in the OCR results and increasing the potential for reuse of the extracted code. The second dataset is composed of 3,750 screenshots of code (1,250 per programming language) written in an IDE using different font types and font sizes. The code was extracted from a variety of GitHub projects, then locally opened in a popular IDE, where a screenshot of the code region was taken. This dataset, unlike the first one, allowed us to manipulate the *font* used to write the code in the IDE and change its size and type, in order to observe how these factors impact the accuracy of OCR engines. The results on this dataset are relevant for creators of programming screencasts, revealing font types and sizes that enable better OCR on the code in their screencasts. At the same time, researchers can use these results as guidelines for identifying screencasts that lend themselves to OCR the best.

Our results show that there are significant differences in accuracy between OCR engines. Google Drive OCR (Gdrive) and ABBYY FineReader performed the best, while the default choice in previous work, Tesseract, was never in the top three best performing OCR engines. We also found that applying OCR on code written using the DejaVu Sans Mono Font produced the best results compared to other fonts such as Consolas, Arial, Courier New, and Times New Roman. Moreover, font size impacted the OCR accuracy significantly for some OCR engines such as Tesseract. However, ABBY FineReader and Google Drive OCR worked the best regardless of the font size.

The rest of the paper is organized as follows. Section 2 introduces the OCR engines and the font types used in our study, Section 3 describes the empirical study we performed on two datasets, Section 4 presents the results we obtained, Section 5 discusses related work, Section 6 discusses threats to validity, Section 7 discusses the results, and Section 9 concludes the paper and describes future work.

2 BACKGROUND

This section introduces important background information about OCR engines and font types.

2.1 OCR Engines

Optical Character Recognition (OCR) is the electronic conversion of images of typed, handwritten, or printed text into machine-encoded text. OCR has been used in many fields, for various applications, including data entry from printed records, forms, and documents; recognition of text on road signs, etc. The main advantages of using OCR technology is the fact that it transforms an image into a format that is searchable, editable, and easy to store electronically.

OCR has also been recently adopted by researchers in software engineering for extracting source code from programming video tutorials [14, 24, 32]. It brings great promise in this area of research, allowing the text found in programming tutorials, including the source code, to be extracted, indexed, searched, and reused. While researchers in software engineering have only used one OCR engine thus far (*i.e.*, Tesseract), there are several other OCR engines that use different computer vision algorithms, leading to high-quality text extraction and real-time performance. In this paper, we investigate and compare six well-known OCR engines: Google Docs OCR, ABBYY FineReader, GOCR, Tesseract, OCRAD, and Cuneiform.

Google Drive[2] is a file storage and synchronization service developed by Google, which interacts with Google Docs, a cloud service allowing users to view, edit, and share documents. Google Drive provides an API, which allows converting images into text using the Google Docs OCR engine. The text extraction process is performed through a cloud vision API that Google uses in the back-end with a custom machine learning model in order to yield high accuracy results.

ABBYY FineReader[3] is a commercial OCR software company that offers intelligent text detection for digital documents. It can process digital documents of many types, such as PDF or, in our case, a video frame containing source code. FineReader uses an advanced algorithm to preserve the layout of the image during the text extraction process. It offers a paid plan, in which users can create a new pattern to train the engine. However, we used the built-in default patterns of the free trial version of FineReader 12 in our work. FineReader offers a software development kit (SDK)[4] to facilitate the OCR extraction process for developers. We used the SDK in our study to enable the efficient processing of our datasets.

GOCR[5] is a free OCR engine that converts images of several types into text. Its fast performance, accuracy, and simplicity make GOCR very practical [10]. It does not require training and handles 20-60 pixels height of single-column sans-serif fonts. Its performance typically degrades in low-quality images, hand-typed texts, and heterogeneous fonts [3]. We used GOCR 0.51 in our study.

Tesseract[6] is an open-source OCR engine developed at HP and released to Google in 2005. Tesseract detects blobs in images that form text lines [28], which in turn are divided into words by detecting the spaces between them [29]. Recognition is performed through two phases that work as a pipeline: (i) recognize each word individually and (ii) run an adaptive classifier to optimize the recognition performance further. Tesseract has been used in several works that are dedicated to code extraction from video programming tutorials [14, 24, 32]. We used version 4.0 in our study.

OCRAD[7] is a free OCR engine supported by the GNU Project. It performs feature extraction on the input image and uses the extracted features to produce text. Like other OCR engines, OCRAD analyzes the layout to recognize text blocks. It typically achieves its best performance when applied to characters of a minimum height of 20 pixels. OCRAD also includes an option to scale the

[2] https://www.google.com/drive/
[3] https://www.abbyy.com/
[4] https://www.abbyy.com/resp/promo/ocr-sdk/
[5] http://jocr.sourceforge.net/
[6] https://github.com/tesseract-ocr/tesseract
[7] http://www.gnu.org/software/ocrad/

words to the proper character size. It has several built-in filters that include, but are not limited to: *letters-only* to discard any number from the input and *text-block* to discard any noise identified outside the detected text lines. We used OCRAD 0.24 in our study.

Cuneiform[8] is an open source OCR engine originally developed by Cognitive Technologies. It supports multiple languages and it preserves the document structure. The format of the extracted text can be HTML, rtf, text, and others. Cuneiform does not require any learning algorithm. We used Cuneiform 1.1 in our study.

2.2 Font Types

In our study, we make use of several fonts in order to observe if the font type and size have an impact on the accuracy of OCR engines. In this section, we present some background information on fonts and their categorization, in order to set the context for our study.

Type families or *typefaces* can be divided into two main categories based on their *features*: Serif and Sans-Serif. *Serif* fonts attach a small line or stroke at the end of each larger stroke in a letter or symbol. *Sans-Serif* fonts exclude such features; note that *Sans* is a French word meaning "without" (*e.g.*, without the ending features on strokes). In addition to the feature-based categories, typefaces can be classified into two other categories based on the *width* they assign to characters. *Proportional* typefaces assign a width to each character based on its shape. For example, the letter (*M*) is wider than the letter (*I*). *Monospaced* fonts, on the other hand, assign a fixed width to each character. While there are many other categories of typefaces [8], we limit our analysis in this paper to the ones presented below. Each typeface or type family also includes several individual fonts. We describe briefly below some of these fonts and mention the ones we used in our study.

Serif types. This category includes two widely used fonts: *Times New Roman* and *Garamond*. *Serif* is very commonly used in electronic books and other digital documents such as magazines and newspapers. In our study, we focus on ***Times New Roman*** from this category, as it is the most commonly used font of this type in electronic documents.

Sans-serif types. The most common fonts belonging to this category are *Helvetica* and *Arial*. U.S. advertisements prioritize the use of *Helvetica* as it became the default typeface [27]. *Arial* has become a very popular font since it was the default font in several Microsoft products such as Microsoft Word. In our study, we focus on ***Arial*** from this category, as it is the most commonly used font of this type in electronic documents.

Monospaced types have been designed since the early age of computers for typewriters. The popularity of *Monospaced* types continued and this type of fonts has been favored in different tasks such as programming and writing tabular data [30]. Popular fonts belonging to this category include ***Courier New***, ***DejaVu Sans Mono***, and ***Consolas***. We consider all three of these fonts in our study since they represent the default fonts in the most popular IDEs for the programming languages used in our study, as explained later in Section 3.1.2 and Table 1.

[8]https://www.linuxlinks.com/cuneiform/

3 EMPIRICAL STUDY

In this section, we present the methodology, research questions, dataset, and results of the study on the accuracy of OCR engines in the context of source code extraction from programming screencasts and images of source code. We first introduce the two datasets we used and the procedure we followed for the data collection. We also make our datasets available in our replication package[9].

3.1 Datasets and Data Collection

In order to establish the OCR accuracy for each engine, we first manually collect the ground truth data consisting of images of source code and the text representing the actual source code extracted from those images. We use two types of images, resulting in two distinct datasets. First, we used YouTube programming screencasts to extract static video frames containing source code, similar to previous work on source code extraction from programming screencasts [14, 25, 32]. By studying the performance of OCR engines on this type of data, we hope to help researchers in this line of research by finding and recommending the best OCR engine for code extraction from videos. Our second dataset consists of screenshots of different code snippets extracted from open source software systems and open in an IDE. We vary the size and type of the source code font such that we can observe the impact of these factors on the performance of the OCR engines and therefore recommend the best settings for future screencast creators. In the next subsections we explain in detail each dataset and the process we followed to obtain it.

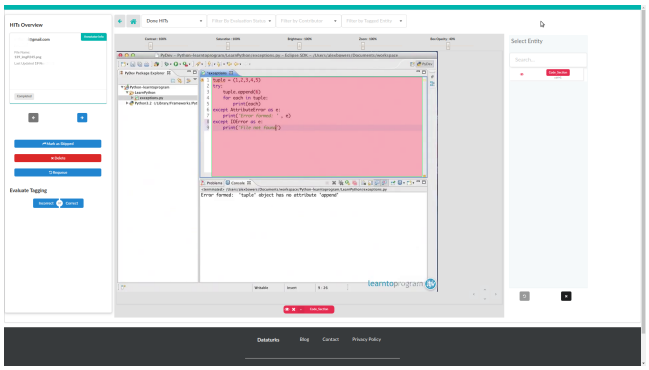

Figure 1: A screenshot of DataTurks, the cloud annotation tool used for the YouTube dataset annotation process.

3.1.1 ***YouTube Screencasts Dataset***. For our first dataset, we turned to YouTube, where vast amounts of programming screencasts on a variety of topics are hosted. For our study, we were interested in programming screencasts displaying code written in three programming languages: Java, C#, and Python. For each programming language, we selected a total of 100 screencasts, aiming to establish a diverse dataset in terms of the creators of the videos, the IDEs used to write the code, the color of the background, and the topics addressed in the videos. In total, we selected 300 videos following these diversity criteria and used the YouTube-dl[10] tool to

[9]https://zenodo.org/record/3743394
[10]https://github.com/ytdl-org/youtube-dl

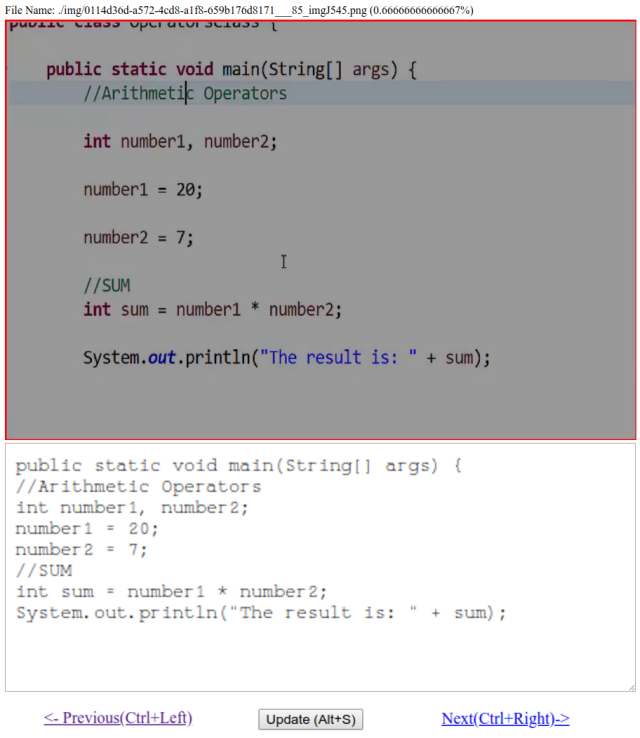

Figure 2: A screenshot of the web-based tool used for the code transcription.

download them. For each video, we then extracted one frame per second using the FFMPEG[11] tool. We then picked one representative frame from each video and included that frame in our dataset, for a total of 300 frames. The representative frame for each video was selected such that it contained a non-trivial amount of code and the code was fully visible on the screen (*i.e.*, not obstructed by other windows or popups). We selected only one frame per video since many of the frames in the same video may display the same piece of code and also share the same image features such as IDE, font type and size, background color, etc. Also, since creating the ground truth for this dataset involved manually transcribing the code from the screen, it was a very effort-intensive task that limited the number of frames we could consider. Therefore, we aimed to maximize the variety of the frames and of the extracted code by selecting only one frame per video and considering more videos, rather than choosing multiple frames from the same video.

Since our goal was to observe the performance of OCR engines for the transcription of source code, we wanted to avoid transcribing noise and focus on applying the OCR engines only on the code snippets. In order to ensure this, one of the authors manually annotated each of the 300 frames with a bounding box that encapsulated only the code snippet. For streamlining this task and minimizing errors, the author used the DataTurks[12] cloud annotation tool (see Figure 1). Another author then validated each annotated bounding box, and, in case of disagreement, the two authors discussed the conflicts until a final bounding box was agreed upon. Then, each image was cropped to the code bounding box using the mogrify[13] tool.

To evaluate the performance of the OCR engines, we need the ground truth source code for each video frame. Two of the authors and two other programmers divided the 300 images and manually transcribed the code in each using an in-house web-based tool (Figure 2). The tool was developed to facilitate the code transcription process. Our tool displays one image at a time from our dataset, with an editing box where the annotators can type the code they see embedded in the image. We added a few options to navigate between frames using the *previous* and *next* buttons. Once the code is fully written, the annotators can save or update the transcribed code using a button. For improving the user experience, we added keyboard shortcuts (*i.e.*, *Ctrl+Left*, *Ctrl+Right*, and *Alt+S* for *previous*, *next*, and textitsave, respectively). To ensure that the transcribed code is correct, one of the authors validated each transcribed source code snippet and corrected the ones containing mistranscriptions. Out of the 300 source code files, 56 were corrected. The corrected mistranscriptions include typos, missing semicolons and parentheses, and case sensitive errors. Another author again validated all corrected files and found no further corrections needed.

3.1.2 ***Screenshots Dataset.*** Our second dataset consists of screenshots we took of code opened in an IDE. This dataset allows us to have control over the font that the source code is written in and manipulate its size and type. This, in turn, allows us to see the impact that changes in these factors have on the performance of OCR engines in extracting the code. Given that IDEs support different font sizes and types, it is important to ensure we study screenshots of the source code with various font configurations.

In this study, we considered the default fonts of the top five most popular IDEs [7] at the time of our data collection (see Table 1), as well as the most popular fonts used in electronic documents. Specifically, we used three *monospaced* fonts (Courier New, DejaVu Sans Mono, and Consolas) and two standard *proportional* fonts (Times New Roman and Arial). Note that, Courier New and Times New Roman are Serif fonts, whereas DejaVu Sans Mono, Consolas, and Arial are Sans-Serif fonts. Concerning the font size, we used different font sizes that range from 10 to 14 points for each font.

To obtain the source code for this dataset, we used the GitHub API to download the top 50 projects in GitHub in terms of star ratings for each of the three programming languages we consider in this study: Java, C#, and Python. We then selected one source code file from each of the total 150 projects we downloaded, such that each file has a size of at least 0.5 KBytes (to ensure it contains enough source code). We then opened each file in an IDE and took a screenshot of a part of the file. In order to determine which part of the file to display on the screen and capture in our screenshots, we needed to consider a few things. First, given the font sizes we consider in our study (10 to 14 points), the default code window size in an IDE can display up to 35 lines without having to scroll up or down. Second, the first few lines of a lot of source code files could be very similar (*e.g.*, import package statements), even across different projects. Therefore, in order to ensure variety in our dataset, it is

[11] https://www.ffmpeg.org/
[12] https://dataturks.com/
[13] https://imagemagick.org/script/mogrify.php

Table 1: Default font settings in Linux and Windows for the top five IDEs.

IDE	Linux		Windows	
	Font Type	Font Size	Font Type	Font Size
Android Studio	DejaVu Sans Mono	12	Courier New	12
Visual Studio	Courier New	14	Consolas	10
Eclipse	Courier New	10	Consolas	10
NetBeans	Courier New	13	Courier New	13
PyCharm	DejaVu Sans Mono	12	Consolas	13

important to vary the position in the file at which we capture the source code. We, therefore, chose 35 consecutive lines that were found at random positions in each of the 150 files considered.

Taking the screenshots for our dataset by hand would require a very intensive human effort. Specifically, the following steps would need to be performed for each of the 150 files considered:

(1) Open the file in the IDE.
(2) Change the font type and size five times each, for a total of 25 combinations.
(3) For each combination of font type and size, scroll the file to the randomly determined position inside the file.
(4) Take a screenshot.

Therefore, taking the 3,750 screenshots by hand is unfeasible. We thus developed a tool called AUTOScR to automate the process described above for taking the screenshots. The IDE used by AUTOScR is Microsoft Visual Studio Code, which offers all the font configurations we needed. AUTOScR uses the *xdotool* tool[14] to simulate keyboard input programmatically and the *scrot* tool[15] to record the current window into a PNG image file. We developed AUTOScR to automatically change the font type of Microsoft Visual Code as follows. AUTOScR opened the settings window by sending the key combination *Ctrl+comma*. Then, AUTOScR types the word "font" by sending the keys "*f o n t*" and sends the *tab* key four times to set the focus on the font family field and insert the font type. After that, for each font type, AUTOScR changes the font size of the code editor to each value in the range from 10 to 14 points. For each font size, AUTOScR sets the focus to the font size field using the same procedures except sending the key *tab* six times instead of four times. Finally, AUTOScR opened the source code files at randomly selected positions in Microsoft Visual Code one by one. After opening a file (*e.g.*, f1.java), AUTOScR took a screenshot of the active window into the corresponding image file (*i.e.*, f1.png). Finally, AUTOScR closed the active window by sending the key *Ctrl+W*.

A total of 3,750 screenshots of code written in three different programming languages (Java, C#, and Python) were taken using our tool. After we took the screenshots, we cropped all images to only the code area using *mogrify*[16]. Moreover, since the OCRAD OCR engine accepts only pbm, pgm, or ppm files, we created a ppm copy of all images using *mogrify*. In the case of this second dataset, getting the ground truth source code was trivial, since it was the same source code we took the screenshots of.

[14]https://www.semicomplete.com/projects/xdotool/
[15]https://github.com/dreamer/scrot
[16]https://imagemagick.org/script/mogrify.php

3.2 Accuracy Metric

We evaluate the performance of the six OCR engines by comparing the text they extract from our two datasets to the ground truth source code for each. In previous work, edit-based similarity metrics have been used to measure the similarity between two strings in OCR-related studies [13, 31]. In our study, we use a normalized version of the Levenshtein distance metric [15] (LD), which is an edit-based similarity metric, to capture the difference between the OCRed text and the ground truth source code. The Levenshtein distance counts the number of insert, substitute, and delete operations required to change one string to another. We normalized LD to obtain an accuracy score that ranges between zero and one, as defined in Equation 1. The Normalized Levenshtein distance (NLD) then represents the similarity percentage between two texts t_1 and t_2, where an NLD of zero means the two texts are completely dissimilar, and an NLD of one indicates that they are identical.

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

One known limitation of LD is that it also counts the number of inserted spaces and blank lines, as they are considered regular characters. However, for many programming languages, including C# and Java which we considered in our study, the spaces and blank lines do not have any effect on the behavior of the source code. With this in mind, given that OCR engines can introduce spaces in the original text during the transcription process, we ignore the leading and trailing spaces and blank lines in the texts for most of our analysis.

One exception, on the other hand, is Python, where white spaces indicate groups of statements and are important to the compilation of Python scripts. Therefore, we introduced a separate analysis in our study on the effect that removing or keeping white spaces and blank lines has on the accuracy of OCR engines in transcribing Python code.

3.3 Research Questions and Methodology

In this study, we are interested in observing the accuracy and performance of OCR engines when applied to images containing source code. To evaluate the six different OCR engines, we used the two datasets collected in Section 3.1. For screenshots, we varied the font type and size according to the five font types and five sizes mentioned in Section 3.1.2. Mind that for the frames we extracted from videos, we cannot vary and cannot accurately predict the font type and size. Therefore, we do not study these factors in the context of video frames.

Our aim is to answer the following research questions:

RQ1: What are the *accuracy* scores of the OCR engines for extracting code from *video frames*?
To answer this research question, we applied each of the six OCR engines to the 300 video frames extracted from programming screencasts and then compared the text they extracted against the ground truth source code using the NLD score while ignoring spaces and blank lines.

RQ2: What are the *accuracy* scores of the OCR engines for extracting code from *screenshots*?
To answer this research question, we applied each of the six OCR engines to the 3,750 source code screenshots obtained from the 150 source code files imported from GitHub by applying all the 25 combinations of font size and type one by one and then considering the aggregated results. We then compared the text that the OCR engines extracted from these screenshots to the ground truth source code using the NLD score, while ignoring spaces and blank lines.

RQ3: To what extent does the *font type* affect the code extraction performance from *screenshots*?
To answer this question, we computed the average NLD accuracy score for OCR engines when applied to screenshots with respect to the different font types, while ignoring spaces and blank lines.

RQ4: To what extent does the *font size* affect the code extraction performance from *screenshots*?
To answer this question, we computed the average NLD accuracy score for OCR engines when applied to screenshots with respect to the different font sizes, while ignoring spaces and blank lines.

RQ5: To what extent does the *programming language* have an effect on the accuracy scores of the OCR engines?
To answer this research question, we computed the average NLD score for each OCR engine applied to both screenshots and video frames with respect to the three programming languages considered, while ignoring spaces and blank lines.

RQ6: What is the effect of *the space and blank line trimming step* on the accuracy of the OCR engines on *Python* code?
As noted in Section 3.2, spaces and blank lines may only have an impact on Python code among the three programming languages considered in our study. However, the impact of this aspect is unknown. In this research question, we aim to quantify this impact on the accuracy of the OCR engines by comparing their results on Python code with or without applying the space and blank line trimming step on the extracted text and ground truth.

RQ7: What is the *time performance* of OCR engines when extracting source code from *video frames* and *screenshots*?
To answer this research question, we computed the running time of each OCR engine during the extraction process using the *system call time*. The system call time computes the real-time from start to finish of the process. This is all elapsed time, including time slices used by other processes. We performed the extraction for all engines in the same conditions, using a machine with an Intel Core i7-7820HQ Quad-Core 2.90GHz processor and 32GB RAM with a high-speed Internet connection (200 Mbps).

4 RESULTS

This section presents the results for each of our research questions. The full results of our study are available in our replication package[17].

4.1 OCR Engines and Video Frames

To answer this research question, we computed the overall accuracy for each engine in terms of its NLD on extracting source code from video frames. The results are shown in Figure 3. Google Drive API and ABBYY FineReader had the best overall accuracy, 95%, and 93%, respectively. GOCR and Tesseract followed, with accuracies of 67% and 57%, respectively. Finally, OCRAD and Cuneiform scored the lowest of all OCR engines, with accuracies of 26% and 20%, respectively.

Surprisingly, our results show that the only OCR engine employed in previous work on the extraction of source code from video frames [14, 25, 32], Tesseract, was only fourth in terms of accuracy, and performed worse than Google Drive API, ABBYY FineReader, and GOCR. *This indicates that previous approaches may benefit from adopting a more accurate OCR engine for this particular task, such as Google Drive API or ABBYY FineReader.*

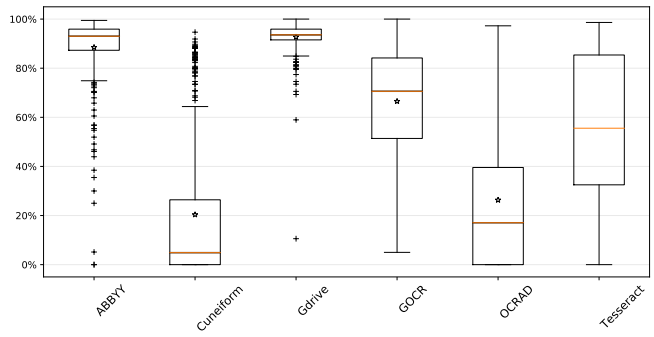

Figure 3: Boxplots of the OCR engines' accuracies measured using the Normalized Levenshtein Distance on video frames

4.2 OCR Engines and Screenshots

Figure 4 shows the overall accuracy of each OCR engine in extracting source code from screenshots. ABBYY FineReader and Google Drive API outperformed the four other OCR engines, scoring an overall accuracy of 96% and 94%, respectively. Additionally, GOCR and Tesseract had an accuracy of 78% and 60%, respectively. At long last, OCRAD and Cuneiform obtained again the lowest scores, with an accuracy of 32% and 18%, respectively.

We notice that overall the accuracies of all OCR engines are higher for screenshots than for video frames. This is to be expected given that in general the screenshots were of higher quality compared to video frames, which varied in quality. However, we observe that the trend in accuracies remains the same among the six OCR engines, with Google Drive API and ABBYY FineReader again scoring the best, while Tesseract remains fourth. This offers additional support for the idea that *Tesseract is not the best OCR engine choice*

[17] https://zenodo.org/record/3743394

for extracting source code from images, but rather *Google Drive API and ABBYY FineReader would represent much better choices, since they achieve almost perfect accuracies on both video frames and screenshots.*

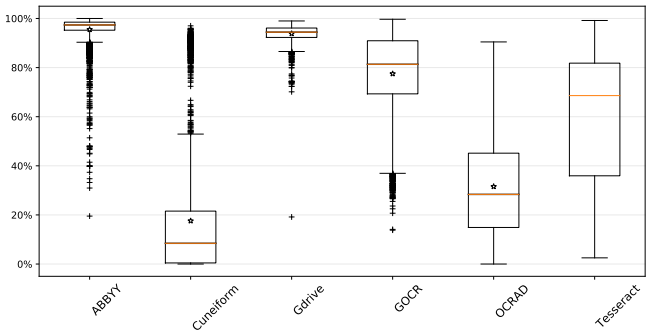

Figure 4: Boxplots of the OCR engines' accuracies measured using the Normalized Levenshtein Distance on screenshots

4.3 OCR Engines and Font Types

Figure 5 shows the accuracy of applying OCR engines on screenshots that contain source code written using different font types. The font DejaVu Sans Mono performed the best among all fonts for four out of the six engines. The exceptions were Google Drive API and ABBYY FineReader, where the font Consolas was the best. Across all engines, DejaVu Sans Mono achieved the highest average accuracy of 71%, followed by Consolas with 67%. The average accuracy across all OCR engines of Arial, Courier New, and Times New Roman were 63%, 60%, and 54%, respectively.

Notably, ABBYY FineReader and Google Drive obtained an accuracy of over 90% for all fonts. GOCR scored its best accuracy for DejaVu Sans Mono and its worst accuracy for Times New Roman. Times New Roman also had the worst accuracy among all fonts for Tesseract, with 20% less than the average of the other five fonts. OCRAD scored less than 50% accuracy for all fonts; however, it showed better results for DejaVu Sans Mono, with 11% higher accuracy than the average of all fonts. Finally, Cuneiform scored the worst among all OCR engines, performing particularly poorly with the font Courier New, which achieved an accuracy 11% lower than the average of all fonts for this engine.

Overall, we observe that for four out of the six engines, the choice of font type has an impact on their accuracy, with differences of at least 20% between their worst and their best performing fonts. In some cases the differences between fonts are very noticeable, such as Cuneiform with Courier New and Arial, where the accuracy triples between the two fonts. However, for the best performing engines ABBYY FineReader and Google Drive, the choice of font is not as important, as they perform almost perfectly for all fonts. Therefore, *the choice of OCR engine seems to be more important than the choice of font type.*

4.4 OCR Engines and Font Sizes

Figure 6 shows the accuracy of applying OCR engines on screenshots that contain source code written using different font sizes. Unsurprisingly, the results show that the accuracy increases with the increase in font size for all engines. The differences are substantial for the four least-performing engines, showing that size has a noticeable impact on these engines. The accuracy of GOCR was impacted by an average of 5.4% by increasing the size. Additionally, the average increases in the accuracy of Tesseract, OCRAD, and Cuneiform were 8.6%, 8.8%, and 7.3%, respectively. On the other hand, Google Drive and ABBYY FineReader were barely impacted by the change in size, as both of them scored more than 90% accuracy for all font sizes. This again supports the idea that *the choice of OCR engine is the most important, more important than the font size. Choosing a highly accurate OCR engine such as Google Drive and ABBYY FineReader seems to mitigate also this factor.*

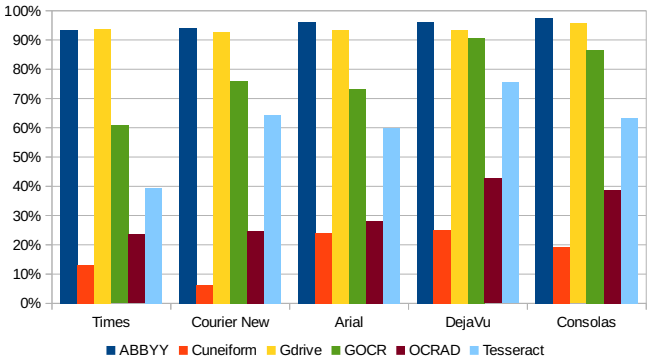

Figure 5: Accuracy of OCR engines for different font types

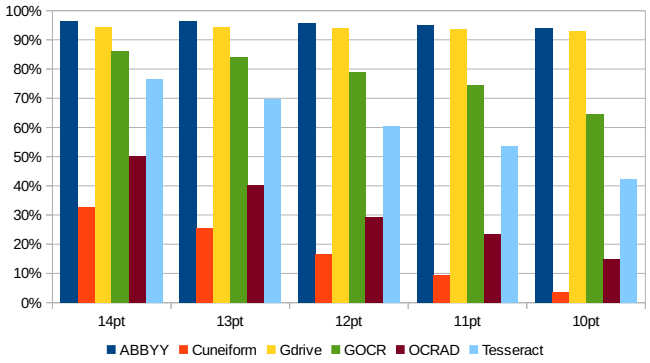

Figure 6: Accuracy of OCR engines for different font sizes

4.5 Programming Languages

Figure 7a shows the accuracy of the extracted code using the six OCR engines on video frames that contain code written in the three programming languages: Java, Python, and C#. The OCR engines scored the highest accuracy on Java source code, followed by C# and Python, with an overall accuracy across all OCR engines of 62% for Java and 56% for C# and Python. Google Drive obtained comparable results for all programming languages with a slight improvement for Python and Java (93% for Python and Java, and 92% for C#). ABBYY FineReader obtained its best accuracy for C#,

followed by Java, then Python with an average accuracy of 92%, 89%, and 84%, respectively. GOCR had similar results for C# and Java (65% and 64%, respectively) and a worse result for Python with an average accuracy of 58%. Tesseract obtained its best accuracy for Python, followed by Java, then C# with an average accuracy of 64%, 59%, and 48%, respectively. OCRAD obtained its best accuracy for Java, followed by C#, then Python with an average accuracy of 36%, 23%, and 18%, respectively. Finally, Cuneiform obtained its best accuracy for Java, followed by Python, then C# with an average accuracy of 28%, 19%, and 14%, respectively.

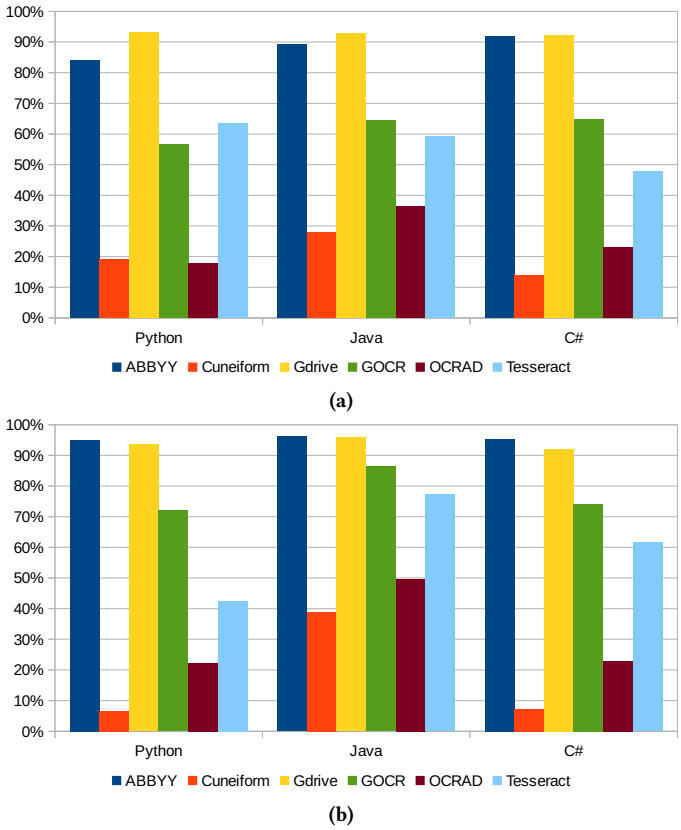

Figure 7: The accuracy of OCR engines for the different programming languages on (a) video frames and (b) screenshots

Figure 7b shows the accuracy of the extracted code using the six OCR engines on the screenshots of code written in the three programming languages. The OCR engines scored the highest accuracy overall on Java source code, followed by C#, then Python with an overall accuracy of 74%, 59%, and 55%, respectively, across all six engines. Overall, the accuracy on Java was the best for all OCR engines. On the other hand, the accuracy on Python was the lowest for all OCR engines except for Google Drive. Moreover, ABBYY FineReader and Google Drive obtained comparable results for all programming languages, with a slight preference for Java. On the other hand, the gap between the accuracy on Java and Python for the rest of the engines was substantial (*e.g.*, 35% for Tesseract, 32% for Cuneiform, and 27% for OCRAD).

Overall, the results again support the idea that *the right OCR engine will perform well no matter the language in which the source code is written.* This is especially true for screenshots, where both Google Drive and ABBYY FineReader achieved consistent accuracies of over 90% irrespective of the programming language. While the accuracy of ABBYY FineReader decreases for Python in the case of video frames, this could be partially explained by the variability in quality when it comes to video frames compared to the stable quality of the screenshots in our dataset.

4.6 OCR Engines without Trimming for Python

Some programming languages, such as Python, use whitespace to determine the grouping of statements. Therefore, we may need to consider not trimming the whitespace for such programming languages. Figure 8 shows the results of applying OCR engines on Python code with and without removing the whitespace and blank lines. The trimming step enhanced the accuracy of OCR engines overall by 6.43% on average. The accuracy of ABBYY FineReader and Google Drive increased by 14.04% and 13.06%, respectively when using trimming. Additionally, the accuracy of Tesseract increased by 6.02%. Finally, the trimming step did not substantially affect OCRAD, GOCR, and Cuneiform; their accuracy slightly increased by 2.64%, 2.37%, and 0.45%, respectively. Note that OCRAD and Cuneiform worked the worst in extracting the source code regardless of the trimming step.

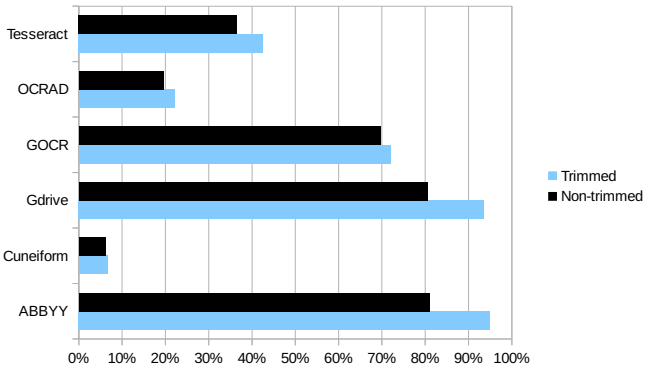

Figure 8: Comparing the accuracy of OCR engines on Python with and without trimming

4.7 The Time Performance of OCR Engines

Table 2 shows the code extraction time of the six OCR engines we used in this study. Roughly, it took less time to extract source code from video frames than screenshots for all OCR engines except Google Drive. Screenshots have higher quality (*i.e.*, more pixels) than video frames, which might slightly delay the processing and increase the code extraction time. Google Drive is the only OCR engine that extracts code in the cloud, which might cause the delay we observed in its extraction time. ABBYY and Google Drive performed the slowest for both video frames and screenshots. Note that they both produced the most accurate results as shown previously, but that might involve expensive image processing which

Table 2: The time performance results (in seconds of the six OCR engines in extracting source code from images

		ABBYY	Cuneiform	Gdrive	GOCR	OCRAD	Tesseract
Videos	Total time (sec)	753	40	1807	14	4	467
	Images	300	300	300	300	300	300
	Time/image	2.51	0.13	6.02	0.05	0.01	1.56
Screenshots	Total time (sec)	9528	957	21333	464	132	17262
	Images	3750	3750	3750	3750	3750	3750
	Time/image	2.54	0.26	5.69	0.12	0.04	4.60

could explain the delay in the extraction process. It took less than a second for Cuneiform, GOCR, and OCRAD to extract code from each image. Although Cuneiform and OCRAD were the fastest, they produced the least accurate results compared to other OCR engines. GOCR, on the other hand, managed to produce reasonable results in near real-time (*e.g.*, less than 12 milliseconds per image).

5 RELATED WORK

5.1 Analyzing Programming Video Tutorials

Over the last few years, researchers have started analyzing video programming tutorials with the goal of extracting, classifying, and leveraging the information contained in them to support developers in their daily tasks. We present an overview of these works below.

Parra *et al.* [21] proposed an approach to tag video tutorials in order to make it easier for developers to identify relevant videos. Poche *et al.* [22] analyzed and classified video comments based on their relevance to the content of the video they correspond to. Moslehi *et al.* [18] automated the process of linking source code files to the videos that exemplify the features that the source code files implement. Bao *et al.* [4] proposed VT-Revolution, a video tutorial authoring system that records all actions taken by developers within the video. Zhao *et al.* [33], on the other hand, automated the process of detecting actions in existing video tutorials by applying computer vision and deep learning algorithms.

Ott *et al.* [20] trained a convolutional neural network (CNN) to recognize frames that contain code in programming video tutorials. The authors also used a CNN to predict the programming language (Java or Python) of code captured in images using only image features, without having to extract the image contents [19]. In our previous work [2], we proposed an approach to locate the editing window of the source code within video frames, therefore enabling noise removal before OCR is being applied. While our long-term goal is similar to these previous works (*i.e.*, correct extraction of source code from video tutorials), the empirical study we present in this paper focuses on identifying the best OCR engine for the process of code extraction.

The most related works to our study are the few approaches that have utilized OCR engines in order to extract the source code appearing on the screen. Ponzanelli *et al.* [26] proposed an approach that identifies various types of fragments in video tutorials based on their contents and allows developers to search for individual video fragments instead of full videos. One of the types of fragments identified was "code implementation", in which the authors applied OCR (Tesseract) on each frame to find and extract code. Yadid *et al.* [32] also used Tesseract on programming video frames to extract source code and proposed two main heuristics to post-process and fix OCR errors. Khandwala *et al.* [14] extracted source code using Tesseract and made it available to developers while watching the video. In this study, we found that Tesseract did not produce a high-quality source code transcription. We also found that Google Drive and ABBYY FineReader performed far better and, as opposed to Tesseract, were not impacted by factors such as font size, font type, or programming language.

5.2 Empirical Evaluation of OCR Engines

Manually transcribing text from a large set of images is an overwhelming and impractical task which requires intensive human effort and time. OCR engines have therefore been very helpful in this regard, as they are able to automatically extract text documents from images which are then easy to search and process. OCR engines have been successfully applied in many different areas, such as machine learning [5, 16], biomedical informatics [9], and computer vision [12].

Empirical studies evaluating OCR engines have been performed in different fields. Tafti *et al.* [31] performed experimental evaluations on the performance of Google Docs, Tesseract, ABBYY FineReader,and Transym. The authors used a synthesis dataset that includes various categories, such as license plates, barcodes, and PDF files. We focus our work on a completely different category, source code images taken from two sources: screenshots and screencasts. However, our results are similar in that we also found that Google Drive and ABBY FineReader perform the best, even though our accuracies are much higher than those in the study by Tafti *et al.* [31].

Helinski *et al.* [13] compared the accuracy of Tesseract and ABBYY FineReader. The comparison was limited to Polish historical printed documents, and both Tesseract and ABBYY were trained on a historical documents dataset. Dhiman *et al.* [10] compared the accuracy of Tesseract and GOCR using three different font types. However, they limited the maximum number of characters in each image to only 39.

6 THREATS TO VALIDITY

We faced two main threats to the *internal validity* of our results. First, our study involved the manual annotation of the code bounding box in each video frame. The bounding box must include all the source code and exclude any information outside the code editing box (*e.g.*, the line number displayed on the left of each line). To mitigate this threat, one author manually annotated each frame and another author validated each annotation. Second, we manually transcribed the ground-truth code, which had to contain no errors.

To alleviate this threat, four annotators manually transcribed the code, one author then validated and corrected the transcribed code, followed by another author who re-validated each transcribed code snippet.

Construct validity in our case regards the metric we used to evaluate the accuracy of the six OCR engines. We mitigate this threat by employing well-known and widely used edit-based metrics previously employed in many research disciplines to measure OCR accuracy [13, 31]. Note that since OCR engines make mistakes at the character level, measuring the edit distance is a more reliable assessment than token-based distance.

Concerning the threats to the *external validity*, the OCR evaluations we performed might be limited to certain programming languages, font types, font sizes, and image resolutions. To mitigate these threats, we assessed the performance of each OCR engine when applied on three different programming languages, five font types, five font sizes that range from 10 to 14, as well as on images of various quality (*e.g.*, screenshots had the highest quality, whereas the quality of the video frames varies).

7 DISCUSSION

Extracting noise-free source code from programming screencasts is a challenging problem which has been studied in several works over the last few years. The main steps deal with detecting which frames contain code [20], localizing the main code editing window [1, 2], choosing and applying an OCR engine to the code editing window, post-processing the extracted code to remove the noise and correct the code [32], and merging different extracted code segments to obtain the complete code [14]. Currently, the only step that has not been investigated at all in previous work is the choice of OCR engine and determining the best suited OCR engine for code extraction. Choosing the best OCR engine can reduce the effort and the errors involved in post-processing the extracted code, therefore getting us closer to reusable code. Addressing this gap is the main goal of the study we presented in this paper. We found that choosing the right OCR engine is extremely important as it can lead to a significant reduction in transcription errors.

One important factor in our study is the type of images we collected and evaluated the six OCRs on. Current works that extracted source code from programming screencasts use YouTube to collect the data [24, 25, 32]. Therefore, we evaluated the six OCR engines on frames taken from programming screencasts hosted on YouTube to help future researchers choose the best OCR that can be applied to video frames. **We found that Google Drive and ABBY FineReader are the most accurate, as well as the most robust OCR engines. We therefore highly recommend researchers using OCR on video tutorials to choose one of these two engines instead of Tesseract for the highest accuracy. We particularly recommend Google Drive if the programming tutorials feature Python code.**

We also explored the accuracy versus speed trade-off in the choice of OCR engine. A high-speed OCR engine is more efficient in extracting source code in real-time, which may be important for time-sensitive operations, whereas a more accurate OCR engine is desirable whenever the code extraction is done in a preliminary, offline step. Most of the previous approaches that extracted and used the source code found in videos so far have had two phases. The first phase is performed offline, and is where all the information (including source code) is extracted from programming screencasts and indexed or further processed. Then, during the second phase, this information is displayed to the developers or made available for searching. Therefore, based on the research performed so far in this direction, speed is not as crucial as accuracy. Therefore, in these applications, **it makes again more sense for researchers to choose the most accurate OCR engines, such as Google Drive or ABBYY FineReader. For applications that want to prioritize speed without significantly impacting the accuracy, we recommend using GOCR.**

We also investigated screenshots of code in order to establish if font size or type have an influence over the quality of the automatic transcription. This is important in order to offer recommendations to video creators and to the best of our knowledge, this is the first paper that aims to do so. We found that the best OCR engines Google Drive and ABBY FineReader are able to extract the code with high accuracy irrespective of which of the five fonts and five sizes were chosen. While we studied these aspects on screenshots, which have a higher resolution compared to video frames, Google Drive and ABBY FineReader obtained very encouraging and robust results also on video frames. **Therefore, we are confident in recommending to video creators any of the five fonts and five sizes we studied, in order to ensure the best OCR code extraction from their videos.**

8 ACKNOWLEDGMENTS

Sonia Haiduc was supported in part by the National Science Foundation grants CCF-1846142 and CCF-1644285.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we performed an empirical evaluation of the accuracy and the speed of six optical character recognition (OCR) engines using two datasets collected from programming screencasts and source code screenshots. Additionally, we showed the impact of different factors such as font type and size on the quality of the extracted code. Although Google Drive and ABBYY FineReader performed the slowest compared to the other four OCR engines, we found that they produced the most accurate results when applied to programming screencasts and source code screenshots, and therefore, are well-suited for offline source code extraction. OCRAD and Cuneiform were the fastest engines for code transcription, but at the price of accuracy. Ultimately, we offer a set of guidelines for video creators and/or future researchers on which OCR engine (i) produces the best results, (ii) works in near real-time, and (iii) is more tolerant to different font configurations.

In future work, we plan to investigate OCR engines with models trained on our particular type of data. Also, we plan to experiment with applying computer vision algorithms for image denoising and smoothening on our input images to improve the quality of the code extraction. We will also focus on analyzing OCR errors and fixing them. This will provide a bigger picture on which OCR engine should we use for extracting source code with pre-processing and post-processing steps.

REFERENCES

[1] Mohammad Alahmadi, Jonathan Hassel, Biswas Parajuli, Sonia Haiduc, and Piyush Kumar. 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, Finland, 2–11. https://doi.org/10.1145/3273934.3273935

[2] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. 2020. Code Localization in Programming Screencasts. *Empirical Software Engineering* (2020), 1–37.

[3] Haslina Arshad, Rimaniza Zainal Abidin, and Waqas Khalid Obeidy. 2017. Identification of Vehicle Plate Number Using Optical Character Recognition: A Mobile Application. *Pertanika Journal of Science and Technology* 25 (2017), 173–180.

[4] Lingfeng Bao, Zhenchang Xing, Xin Xia, and David Lo. 2018. VT-Revolution: Interactive programming video tutorial authoring and watching system. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2802916

[5] Sameeksha Barve. 2012. Optical character recognition using artificial neural network. *International Journal of Advanced Research in Computer Engineering & Technology* 1, 4 (2012), 131–133.

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 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, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[7] Pierre Carbonnelle. [n.d.]. Top IDE index. https://pypl.github.io/IDE.html. Accessed: 2019-10.

[8] Rob Carter, Philip B Meggs, and Ben Day. 2011. *Typographic design: Form and communication*. John Wiley & Sons.

[9] Thomas Deselaers, Henning Müller, Paul Clough, Hermann Ney, and Thomas M Lehmann. 2007. The CLEF 2005 automatic medical image annotation task. *International Journal of Computer Vision* 74, 1 (2007), 51–58.

[10] Shivani Dhiman and A Singh. 2013. Tesseract vs gocr a comparative study. *International Journal of Recent Technology and Engineering* 2, 4 (2013), 80.

[11] Adam Grzywaczewski and Rahat Iqbal. 2012. Task-specific information retrieval systems for software engineers. *J. Comput. System Sci.* 78, 4 (2012), 1204–1218.

[12] Maya R Gupta, Nathaniel P Jacobson, and Eric K Garcia. 2007. OCR binarization and image pre-processing for searching historical documents. *Pattern Recognition* 40, 2 (2007), 389–397.

[13] Marcin Helnski, Miłosz Kmieciak, and Tomasz Parkoła. 2012. Report on the comparison of Tesseract and ABBYY FineReader OCR engines.

[14] Kandarp Khandwala and Philip J. Guo. 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, United Kingdom, 1–10. https://doi.org/10.1145/3231644.3231652

[15] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[16] Huei-Yung Lin and Chin-Yu Hsu. 2016. Optical character recognition with fast training neural network. In *2016 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 1458–1461.

[17] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 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, Italy, 104–114.

[18] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature location using crowd-based screencasts. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, Gothenburg, Sweden, 192–202. https://doi.org/10.1145/3196398.3196439

[19] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. 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*. 376–386.

[20] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. , 336-339 pages.

[21] Esteban Parra, Javier Escobar-Avila, and Sonia Haiduc. 2018. Automatic tag recommendation for software development video tutorials. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 222–232.

[22] Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Vesper, and Anas Mahmoud. 2017. Analyzing user comments on YouTube coding tutorial videos. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 196–206.

[23] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F Cohen. 2011. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 135–144.

[24] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch!: Extracting relevant fragments from software development video tutorials. ACM Press, 261–272. https://doi.org/10.1145/2884781.2884824

[25] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. 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, TX, 645–648.

[26] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, Sonia Cristina Haiduc, Barbara Russo, and Michele Lanza. 2017. Automatic identification and classification of software development video tutorial fragments. *IEEE Transactions on Software Engineering* (2017). https://doi.org/10.1109/TSE.2017.2779479

[27] Alice Rawsthorn. 2007. Helvetica: The little typeface that leaves a big mark. https://www.nytimes.com/2007/03/30/style/30iht-design2.1.5085303.html

[28] Ray Smith. 1994. A simple and efficient skew detection algorithm via text row algorithm. *Personal Systems Laboratory, HP Laboratories Bristol, HPL-94-113 December* (1994).

[29] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 2. IEEE, 629–633.

[30] Joel Spolsky. 2001. User Interface Design For Programmers. https://www.joelonsoftware.com/2001/10/24/user-interface-design-for-programmers/

[31] Ahmad P Tafti, Ahmadreza Baghaie, Mehdi Assefi, Hamid R Arabnia, Zeyun Yu, and Peggy Peissig. 2016. OCR as a service: an experimental evaluation of Google Docs OCR, Tesseract, ABBYY FineReader, and Transym. In *International Symposium on Visual Computing*. Springer, 735–746.

[32] Shir Yadid and Eran Yahav. 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, The Netherlands, 98–111.

[33] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: vision-based workflow action recognition from programming screencasts. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 350–361.

[34] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, Guoqiang Li, and Shanghai Jiao Tong. 2019. ActionNet: Vision-based workflow action recognition from programming screencasts. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*.