mathematics MDPI

Article

Optimizing OCR Performance for Programming Videos: The Role of Image Super-Resolution and Large Language Models

Mohammad D. Alahmadi * and Moayad Alshangiti

Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 23890, Saudi Arabia; mshangiti@uj.edu.sa
* Correspondence: mdalahmadi@uj.edu.sa

Abstract: The rapid evolution of video programming tutorials as a key educational resource has highlighted the need for effective code extraction methods. These tutorials, varying widely in video quality, present a challenge for accurately transcribing the embedded source code, crucial for learning and software development. This study investigates the impact of video quality on the performance of optical character recognition (OCR) engines and the potential of large language models (LLMs) to enhance code extraction accuracy. Our comprehensive empirical analysis utilizes a rich dataset of programming screencasts, involving manual transcription of source code and the application of both traditional OCR engines, like Tesseract and Google Vision, and advanced LLMs, including GPT-4V and Gemini. We investigate the efficacy of image super-resolution (SR) techniques, namely, enhanced deep super-resolution (EDSR) and multi-scale deep super-resolution (MDSR), in improving the quality of low-resolution video frames. The findings reveal significant improvements in OCR accuracy with the use of SR, particularly at lower resolutions such as 360p. LLMs demonstrate superior performance across all video qualities, indicating their robustness and advanced capabilities in diverse scenarios. This research contributes to the field of software engineering by offering a benchmark for code extraction from video tutorials and demonstrating the substantial impact of SR techniques and LLMs in enhancing the readability and reusability of code from these educational resources.

Keywords: OCR (optical character recognition); code extraction; programming screencasts; image quality; pre-processing techniques; postprocessing techniques; large language models (LLMs); source code denoising; video programming tutorials; empirical study in software engineering

MSC: 68T45

check for updates

Citation: Alahmadi, M.D.; Alshangiti, M. Optimizing OCR Performance for Programming Videos: The Role of Image Super-Resolution and Large Language Models. *Mathematics* **2024**, *12*, 1036. https://doi.org/10.3390/math12071036

Academic Editor: Ivan Lorencin

Received: 29 February 2024
Revised: 18 March 2024
Accepted: 27 March 2024
Published: 30 March 2024

1. Introduction

Developers often turn to a variety of internet sources, including forums, blogs, and Q&A websites like StackOverflow, to find support and knowledge for their programming tasks. A significant part of their online activity—estimated between 20% and 30% of their time—is dedicated to searching for reusable code snippets that they can directly incorporate into their projects, highlighting the importance of these resources in their daily development work [1,2]. Video programming tutorials have become a popular source for developers seeking step-by-step visual instructions, offering a unique educational experience by demonstrating programming concepts and practices [3–6]. However, the code showcased in these videos is embedded within the video frames, making it inaccessible for direct reuse. As a result, developers are often required to manually transcribe the code from the videos, a process that is not only time-consuming but also prone to errors [6,7].

To reuse and enable developers to copy–paste the source code embedded in video frames, we need to apply optical character recognition (OCR) to the code frames. This method extracts the textual content from the images, facilitating the transformation of pixelated code into a format that developers can easily integrate into their projects. Over the last decade, many efforts have been dedicated to facilitating the process of extracting source code from video programming tutorials [7–16]. Several approaches have been proposed to clean the extracted code, such as by (i) using statistical language models [10,16] for Java language, (ii) verifying the extracted tokens using a dictionary-based approach [14,15,17], and (iii) using the CodeT5 model [18] to correct Java code [13]. Other work focused on developing an ensemble technique that improves OCR performance on specific domains, such as extracting texts from (i) geology papers using Softmax regression and SVM [19], (ii) raster images with Arabic script using SVM [20], and (iii) an IAM handwritten dataset using CRNN, LSTM, and CTC [21].

Given the advancements in OCR technology for extracting source code from video programming tutorials, a critical limitation of these previous efforts remains unaddressed: (i) a study on the impact of image quality on OCR performance, (ii) the use of appropriate image pre-processing techniques to enhance code recognition, and (iii) the selection of the most suitable method to extract the source code from video frames. First, considering that video programming tutorials are typically uploaded to YouTube (www.youtube.com) in varying resolutions such as 360p, 480p, 720p, and 1080p, the efficacy of a specific OCR method in extracting source code from low-quality video frames is still uncertain. Yet, previous research has primarily relied on the Tesseract (https://github.com/tesseract-ocr/tesseract), accessed on 1 February 2024, OCR engine for code extraction from images, without a comprehensive evaluation of its effectiveness across different image quality levels [8,9,12,16]. The second crucial aspect overlooked by prior studies is the potential of image pre-processing to boost OCR performance for code extraction from programming tutorials, aiming to eliminate noise that could originate from images with noisy pixels. Yet, previous work has extracted the code from images, then created a custom model for a specific programming language (e.g., Java) to detect and correct errors [10,13,16], without initially addressing image quality or employing image denoising techniques. Finally, previous work has relied on off-the-shelf OCR engines such as Tesseract [7,22] and Google Vision [14,15,23] for extracting source code without investigating the state-of-the-art vision-based large language models (LLMs), which could significantly enhance code extraction performance dramatically.

In this paper, we investigate the impact of image quality and advanced deep-learning techniques on code extraction from video programming tutorials. We explore the effectiveness of optical character recognition (OCR) engines and large language models (LLMs) across various video qualities and programming languages. Our comprehensive empirical analysis compares performance across different resolutions and evaluates the enhancement offered by image super-resolution (SR) techniques. The findings demonstrate the potential of SR to significantly improve OCR accuracy, especially in lower-resolution videos, and highlight the superior performance of LLMs in diverse scenarios.

To sum up, we make the following noteworthy contributions:

- We introduce state-of-the-art large language vision models (LLMs) specifically optimized for extracting source code from video programming tutorials, demonstrating superior performance in code extraction from video frames;
- We utilize advanced image super-resolution (SR) methods to enhance the quality of video frames containing source code. These improvements significantly aid optical character recognition (OCR) engines in accurately extracting source code from low-resolution video frames;
- Our work includes a thorough empirical analysis comparing the performance of two leading OCR engines and two LLMs in terms of their accuracy in extracting code from programming tutorials across various video qualities;
- We conduct a detailed study of the effectiveness of image super-resolution (SR) techniques in generating high-quality code images from their low-quality counterparts, helping OCR engines to better extract code;
- We include a benchmark (included in our online appendix (https://zenodo.org/records/10823097), uploaded on 15 March 2024, that contains (i) a collection of

100 video programming tutorials across four distinct quality levels, (ii) cropped images showcasing the code section for each quality level, (iii) ground-truth source code, (iv) the models utilized for super-resolution enhancement, and (v) extensive results, including scripts for each phase of our study.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. This is followed by Section 3, where we provide an overview of our empirical study. Section 4 is dedicated to presenting our results and key findings. The potential threats to the validity of our study are addressed in Section 5. Finally, we conclude our work in Section 6.

2. Related Work

2.1. Analyzing Video Programming Tutorials

A considerable amount of research has focused on the analysis of programming screencasts, with various approaches being explored to categorize and extract meaningful content from video frames. Notably, convolutional neural networks (CNNs) have been widely applied, as seen in the use of a CNN classifier to differentiate video frames that contain code from those that do not [10,11,22,24,25]. However, a significant limitation of employing CNN classifiers is their heavy reliance on large datasets of labeled frames for effective training [26]. Similarly, Zhao et al. [27] developed `ActionNet`, leveraging a sophisticated CNN architecture to identify various actions within programming screencasts [28], different from earlier methods like VT-Revolution [3] that lacked automated action recognition. Furthermore, the research landscape also includes efforts aimed at enhancing video content accessibility, such as video tagging for better organization [29], linking existing source code to relevant video segments [30], classifying video comments for insights [31], and segmenting videos to improve searchability and navigation [32,33].

It is important to note that much of the existing research utilizes the source code displayed in videos for various purposes, including identifying actions [27,34,35], classifying the source code [14], and tagging videos [29]. This underscores the necessity of selecting the most effective OCR (optical character recognition) engines and LLMs (large language models) tailored for extracting code from videos of varying quality, thereby facilitating the reuse of code for diverse applications. In our study, we focus on this preliminary and critical step by conducting an empirical study of OCR and LLMs combined with computer vision techniques for code extraction from video programming frames.

2.2. Extracting Code from Video Programming Tutorials

Considerable efforts have been dedicated to extracting source code from programming screencasts. Ponzanelli et al. [8] introduced CodeTube, a method for segmenting video tutorials, leveraging code extracted from video frames using Tesseract to find relevant video segments through code similarity analysis. Khandwala and Guo [7] developed Codemotion, designed to retrieve source code and its dynamic modifications from tutorials. Both methodologies, CodeTube and Codemotion, utilize the indexed code to refine video search capabilities. Yadid and Yahav [16] presented ACE, a technique aimed at denoising code extracted from video tutorials using statistical language models after applying the Tesseract engine. Similarly, Malkadi et al. [13] utilized a Bert-based model to denoise the extracted code. Recent work used Google Vision OCR to extract source code from videos for restructuring complete Android XML files [14], extracting meta information [15], and linking code to the relevant video fragment [30].

The work most aligned with our research is Malkadi et al.'s [12] investigation into the efficacy of OCR engines for extracting code from images. While this study comprehensively evaluated six OCR engines, it primarily focused on high-quality screenshots, overlooking the critical role of image quality on OCR accuracy. Furthermore, it did not explore the potential of large language models (LLMs), which possess the capability to surpass the performance of conventional OCR technologies. In our research, we conduct a more extensive evaluation, taking into account the impact of image quality and integrating LLMs to enhance the accuracy of code extraction.

2.3. LLMs in Software Engineering

Recently, the adoption of large language models (LLMs) for tackling software engineering tasks has notably increased. Xu et al. [36] conducted a comprehensive review of large language models' (LLMs) effectiveness in coding contexts, revealing their significant potential for code analysis and development tasks. Chaaben et al. [37] delved into the capabilities of few-shot learning for enhancing model completion processes, showcasing the substantial impact of prompt learning in such scenarios. Kang et al. [38] examined the role of LLMs as few-shot testers, particularly for bug reproduction, thereby paving new avenues for automated software testing. Sobania et al. [39] focused on ChatGPT's proficiency in automatic bug fixing, illustrating its adeptness in pinpointing and rectifying coding errors, suggesting a promising direction for debugging. Akli et al. [40] applied few-shot learning innovatively to forecast the categories of flaky tests, underscoring LLMs' ability to enhance the reliability of test predictions. Lyu et al. [41] introduced Chronos, a novel tool employing zero-shot learning to identify library vulnerabilities from reports, further broadening LLMs' utility in security applications. Le and Zhang [42] ventured into log parsing using few-shot learning, highlighting LLMs' flexibility in interpreting system logs crucial for diagnostics. Nashid et al. [43] explored how strategic prompt selection could optimize code-related learning tasks, emphasizing the nuanced role of prompts in few-shot learning's success. Lastly, Siddiq et al. [44] investigated zero-shot prompting for assessing code complexity through GitHub Copilot, demonstrating LLMs' effectiveness in gauging coding task challenges.

While the literature has demonstrated the power of large language models (LLMs) in different software engineering task, our study uniquely employs vision-based large language models (LLMs) to extract source code, leveraging their advanced capabilities to understand and interpret visual data.

3. Empirical Study

In this section, we outline the research questions (RQs) driving our empirical study, elaborating on the procedures and criteria we followed to (i) collect programming videos from YouTube and (ii) manually select and transcribe a frame that showcases a code snippet. We then describe our methodology for extracting source code from these images, followed by the RQs, where we explain our motivation and methodology for each RQ. Last, we present the evaluation metrics we used to report the results.

RQ$_1$ *How does the quality of video programming tutorials influence the performance of OCR engines in accurately extracting source code, and can LLMs enhance this accuracy?*

RQ$_2$ *To what extent do variations in programming language syntax challenge the code extraction capabilities of OCR engines and LLMs?*

RQ$_3$ *Would the application of deep learning-based image enhancement techniques improve the code extraction performance in lower-quality video tutorials?*

Figure 1 presents a comprehensive summary of our study. In the sections that follow, we engage in a thorough examination of each aspect highlighted in this overview.

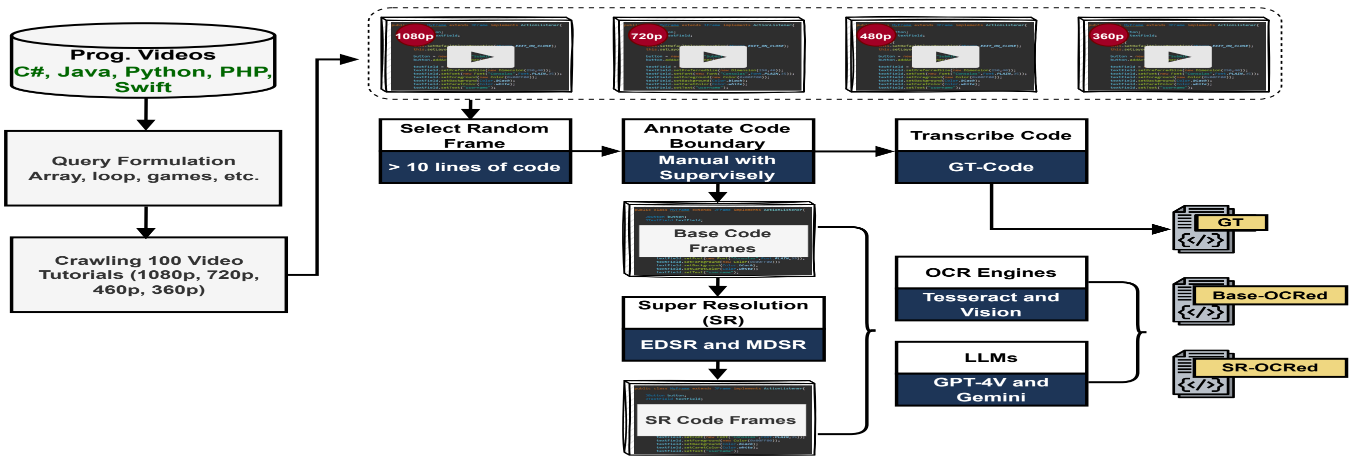


Figure 1. An overview of our empirical study on OCR and LLM accuracy across different video programming qualities using super-resolution techniques.

3.1. Videos: Dataset Collection

We manually selected programming screencasts from YouTube, focusing on ensuring a broad spectrum of content. Our dataset includes 100 videos, with 25 each from Python, Java, C#, and PHP, to support generalizability across different programming languages. We aimed for diversity in topics, employing search queries that combined the programming language with "development" and specific topics like *loops*, *arrays*, and *games*. Additionally, we have chosen programming video tutorials that introduce different levels of code complexity and variety. These include tutorials on programming fundamentals, GUI development, database management, asynchronous coding, API usage, animation, and more. This approach ensures a comprehensive representation of various programming concepts and practices, reflecting the broad syntax and structural diversity inherent in different programming languages. To assess the impact of video quality on OCR performance, we chose videos available in 360p, 480p, 720p, and 1080p resolutions, excluding any that did not meet these criteria. Our collection also captures various IDEs and settings, including Visual Studio and Android Studio, and different background themes to explore how these factors affect code extraction accuracy. Our careful selection of videos aims to support a detailed study of how well code can be transcribed from a variety of programming tutorials.

Table 1 shows a comprehensive summary of our programming screencasts across five programming languages: C#, Java, Python, PHP, and SWIFT, focusing on both the duration of videos in seconds and the number of code lines displayed and manually captured. C# screencasts exhibit a broad duration range from 266 to 14,400 s, with an average length of 1752 s, and show between 13 and 24 lines of code, averaging 20 lines per video. Java videos vary in length from 350 to 1188 s, averaging 816 s, and feature 11 to 19 lines of code with an average of 15 lines. Python tutorials present the most considerable variation in duration, ranging from 343 to 1107 s, with an average of 667 s, and include between 10 and 23 lines of code, averaging 14 lines. PHP videos span from 265 to 1178 s, with an average duration of 774 s, and display 11 to 25 lines of code, with an average of 19 lines. Lastly, SWIFT screencasts have durations ranging from 284 to 25,517 s, with a significant average of 3066 s, and feature 10 to 20 lines of code, averaging 15 lines. This detailed analysis highlights the diversity in our video programming dataset in terms of duration and lines of code.

In our dataset, the background color of the development environment (DE) within the programming screencasts is an important factor for analyzing the accuracy of OCR engines. Of the 100 videos we collated, 54 feature a black or dark background, while 46 have a white or light background. This distinction is critical, as it allows us to assess the impact of background color on the OCR's ability to accurately recognize and extract the source code. The variation in background color will enable us to draw more comprehensive conclusions about the performance of OCR technology under different visual conditions commonly encountered in programming tutorials.

Table 1. Overview of the dataset: detailed statistics of programming screencasts collected from YouTube for evaluating $RQ_{1,3}$.

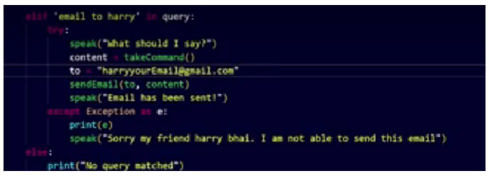
Prog. Lang.	**Duration (Seconds)**			**Lines of Code**		
	Min.	**Max.**	**Average**	**Min.**	**Max.**	**Average**
C#	266	14,400	1752	13	24	20
Java	350	1188	816	11	19	15
Python	343	1107	667	10	23	14
PHP	265	1178	774	11	25	19
SWIFT	284	25,517	3066	10	20	15

3.2. Dataset Labeling: Code Transcribing from Video Frames

In this section, we detail our approach to dataset creation and labeling. Our process involved selecting 100 unique frames, each from a distinct programming tutorial video, and with a minimum of 10 lines of code. For every selected frame, we captured images at four different resolutions: 360p, 480p, 720p, and 1080p. This approach yielded a total of 400 images. We then manually transcribed the source code from these images, creating a ground truth dataset that pairs each image with its corresponding code. Below we explain the process in more detail.

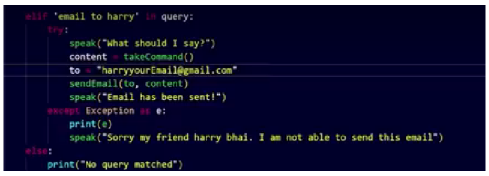
First, for the creation of our dataset, we meticulously selected video frames from the collected tutorials, focusing on frames that displayed a minimum of 10 lines of code, excluding blank lines. For each video representing different quality levels—360p, 480p, 720p, and 1080p—we extracted a single frame that met this criterion. Figure 2 showcases a sample of a frame in our dataset with the four different qualities. The selected frames were then manually cropped using consistent coordinates (*xmin, xmax, ymin, and ymax*) to ensure that only the code was displayed, omitting any surrounding elements. More specifically, we used the Supervisely (https://supervisely.com/), accessed on 1 February 2024, tool to annotate the code editing window with a bounding box for the selected frame from each video (i.e., we opt for an image quality of 1080p, as they are identical to the other images with different qualities). From these annotated images, we gathered all the annotations in JSON format for a total of 100 images. Given that the dimensions (width and height) of images can vary depending on their quality settings, we standardized the bounding box annotations across different image qualities. To achieve this standardization, we implemented a scaling mechanism for the bounding box annotations. This mechanism involves adjusting the coordinates of the bounding box to align with the corresponding dimensions of images of a different quality. All images in our dataset were cropped to the code bounding box and named with their video ID and corresponding quality level. This process yielded a comprehensive set of 400 images, which serves as the basis for our dataset.

In the transcription phase of our study, three professional software developers independently transcribed the code from each image. We provided them with the images, instructing them to accurately transcribe the code. After transcription, we compared the outputs from all three developers to ensure consistency. The transcription accuracy was approximately 88% as per the kappa statistic, indicating a high level of agreement among the transcribers. The main differences we found were in the transcription of non-code elements, like “0 references”, and the occasional inclusion of extra whitespace characters.



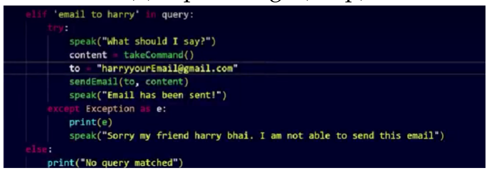
```
elif 'email to harry' in query:
    try:
        speak("What should I say?")
        content = takeCommand()
        to = "harryyourEmail@gmail.com"
        sendEmail(to, content)
        speak("Email has been sent!")
    except Exception as e:
        print(e)
        speak("Sorry my friend harry bhai. I am not able to send this email")
else:
    print("No query matched")
```

(**a**) Input Image (360p)



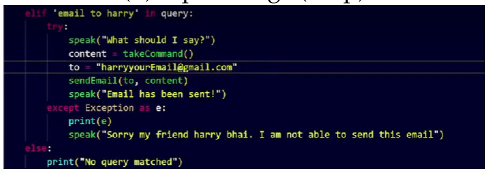
```
elif 'email to harry' in query:
    try:
        speak("What should I say?")
        content = takeCommand()
        to = "harryyourEmail@gmail.com"
        sendEmail(to, content)
        speak("Email has been sent!")
    except Exception as e:
        print(e)
        speak("Sorry my friend harry bhai. I am not able to send this email")
else:
    print("No query matched")
```

(**b**) Input Image (460p)



```
elif 'email to harry' in query:
    try:
        speak("What should I say?")
        content = takeCommand()
        to = "harryyourEmail@gmail.com"
        sendEmail(to, content)
        speak("Email has been sent!")
    except Exception as e:
        print(e)
        speak("Sorry my friend harry bhai. I am not able to send this email")
else:
    print("No query matched")
```

(**c**) Input Image (720p)



```
elif 'email to harry' in query:
    try:
        speak("What should I say?")
        content = takeCommand()
        to = "harryyourEmail@gmail.com"
        sendEmail(to, content)
        speak("Email has been sent!")
    except Exception as e:
        print(e)
        speak("Sorry my friend harry bhai. I am not able to send this email")
else:
    print("No query matched")
```

(**d**) Input Image (1080p)

Figure 2. Visual representation of images with varying resolutions within our Python dataset, spanning from 360p to 1080p. These images showcase the diverse quality levels found in our dataset, reflecting the range of available resolutions.

Consequently, our dataset comprises 100 manually transcribed code segments corresponding to 100 video frames, establishing a robust ground-truth for our empirical evaluation. Furthermore, each video in our collection is represented by four frames spanning a quality that ranges from 360p to 1080p. These multi-quality frames are important to our experiments, where we apply OCR engines to extract the code and then assess its accuracy in terms of qualities against the ground-truth. This approach allows us to evaluate the impact of video quality on OCR performance and explore advancements in image and text processing techniques to improve video quality.

3.3. OCR Engines and Large Language Models

To extract source code from images, we need to carefully select an OCR engine that is both effective in decoding images containing source code and resilient against the distortions present in low-quality images. We note that some videos uploaded to YouTube might be presented in a low quality, and thereby, the code presented in those videos might not be visible to extract and reuse. In this section, we explain our methodology in extracting the source code from our image dataset using both OCR engines and large language models (LLMs) as follows.

First, similar to a previous study, we utilized the most two popular OCR engines for extracting source code from images [8,12,13]. Namely, we used the open-source Tesseract (https://github.com/tesseract-ocr/tesseract), which was released by Google and accesssed on 1 Feburary 2024, given that previous researchers have relied on it for extracting source code from images. Furthermore, we utilized Google Vision, as it was shown that it performed very well in extracting source code from images [12,30].

Second, in our study, we broadened the scope beyond traditional OCR engines for extracting source code from images by exploring the capabilities of large language models (LLMs), specifically focusing on the recent developments in multimodal LLMs. A prime example of this advancement is the introduction of GPT-4 with Vision (GPT-4V) by OpenAI. This model stands out as the state of the art in the computer vision field, offering the unique ability to process both images and text within a unified framework. To adapt this technology for our purpose, we converted all images from our dataset into base-64 format. This conversion was essential for utilizing the “gpt-4-vision-preview” model effectively. We then crafted a prompt instructing the model to “act as an OCR that extracts the code from the image”, ensuring that the model’s focus was finely tuned to our specific need for accurate code extraction. The OCRed code was subsequently stored locally on our server for further analysis and validation.

Third, we also delve into the capabilities presented by the Gemini-Pro-Vision model, Google’s forefront in large language vision models (LLMs), designed to interact with both text and visual modalities (e.g., images and videos). This exploration is motivated by

Gemini's recent enhancement to interpret images, marking it as a significant multimodal LLMs. In leveraging Gemini for our study, we transformed the images from our dataset into a compatible format with the help of the Gemini-API (https://makersuite.google.com/app/apikey), accessed on 1 February 2024, library in Python. We engaged Gemini by submitting prompts that directed the model to function as an OCR specifically tailored for extracting programming code from the images and saved the OCRed code locally on our server.

3.4. Image Super-Resolution (SR)

Dealing with image quality is a fundamental challenge in all image/video processing applications, including code extraction from programming tutorials. Factors such as lighting, shadows, bleeding, overexposure, contrast, and particularly resolution can significantly affect the quality of images extracted from video frames.

In the field of image processing, the problem of image super-resolution (SR), especially single-image super-resolution (SISR), has been a focus of extensive research for many years [45–48]. SISR seeks to reconstruct a high-resolution image from a single low-resolution counterpart, which becomes particularly relevant when considering the extraction of source code from video frames, where we assume the resolution of the image will directly impact the performance of code extraction.

Our research takes inspiration from the groundbreaking work presented in [45], where the authors introduce two advanced deep learning-based SR methods: the enhanced deep super-resolution network (EDSR) and the multi-scale deep super-resolution system (MDSR). These methods have set new benchmarks in the field, outperforming the existing state-of-the-art approaches. The EDSR achieves this through an optimized network structure, by discarding redundant modules found in traditional residual networks, and by increasing the model size. On the other hand, the MDSR innovatively tackles the challenge of handling multiple upscaling factors within a single model, offering a versatile solution for various super-resolution scales. These methods have demonstrated exceptional performance in benchmark datasets and were the winners of the NTIRE2017 Super-Resolution Challenge.

Exploring the application of these advanced deep learning-based techniques in our context could significantly enhance the quality of source code extraction from video tutorials, thereby improving the overall effectiveness of the code extraction process.

3.5. RQ$_1$: The Impact of Image Quality on Code Extraction

Motivation: The motivation behind this research question stems from the increasing reliance on video programming tutorials for learning and sharing coding practices. These tutorials, however, vary widely in quality, potentially affecting the effectiveness of OCR engines in accurately extracting source code from video frames. As the quality of video can significantly impact the accuracy of the extracted code (OCRed code), empirically evaluating its impact is crucial for developing and/or choosing more robust code extraction engines. Furthermore, the advent of large language models (LLMs) offers a new dimension to this challenge. LLMs, with their advanced understanding of context and content, could potentially enhance the accuracy of code extraction beyond the capabilities of traditional OCR engines. This research question aims to explore these dynamics, evaluating both the impact of video quality on OCR performance and the potential of LLMs to improve the extraction process.

Methodology: To address this research question, our methodology leverages the detailed dataset collection of programming screencasts and the manual transcription of source code, as outlined in Sections 3.1 and 3.2. We conducted a series of experiments—totaling 1600—by applying both traditional OCR engines, such as Tesseract and Google Vision, and cutting-edge large language models, including GPT-4V and Gemini, to the 400 video frames at each of the four resolution levels (360p, 480p, 720p, 1080p). This comprehensive empirical study allows us to not only measure the performance of OCR technologies in extracting source code from video frames under varying quality conditions but also evaluate the enhancements offered by LLMs in improving the accuracy of the extracted code. During our LLM experiments, we used zero-shot prompting for both GPT-4V and Gemini, where we did not present the models with any explicit examples and only asked both LLMs to extract code from the given image. For GPT-4V, we used the *gpt-4-vision-preview* model, configuring the role as *user* and the type as *image_url*. Each image was encoded as *base64_image* before passing it to the model for the code extraction. The responses were received in JSON format, parsed to extract the content of the GPT-4v reply, and subsequently saved as text files. As for Gemini, we employed the *gemini-pro-vision* model for code extraction from images. Notably, Gemini includes the programming language at the start of its reply. Consequently, we removed the initial line before saving each response to the corresponding text file.

3.6. *RQ$_2$: The Impact of Programming Language Syntax on Code Extraction*

Motivation: The motivation for this research question arises from the growing importance of accurate text extraction from programming video tutorials with diverse programming language code snippets. As programming languages vary significantly in syntax, structure, and conventions, existing OCR engines may encounter challenges in accurately interpreting and extracting code from these diverse sources. Additionally, with the emergence of powerful language models (LLMs), there is a potential opportunity to improve code extraction accuracy by leveraging the contextual understanding and language modeling capabilities of these models. Understanding the comparative performance of OCR engines and LLMs across a range of programming languages is essential for identifying their strengths, limitations, and potential synergies, thereby enabling future research on code extraction, cleaning, and reusability.

Methodology: To explore the comparative performance of OCR engines and LLMs across different programming languages, our methodology involves a structured evaluation process. We used the collection of programming screencasts and the manual transcription of source code, as outlined in Sections 3.1 and 3.2. We conducted a series of experiments, totaling 1600 experiments, across four distinct OCR engines (Tesseract, Google Vision, GPT-4V, and Gemini). For each programming language and OCR engine, we computed the token-based NLD for the four resolution levels (360p, 480p, 720p, 1080p). Through systematic evaluation and comparison, our study aims to examine the performance differences between OCR engines and LLMs when extracting code snippets from programming tutorial videos across different programming languages.

3.7. *RQ$_3$: The Impact of Applying Image Super-Resolution (SR) on Code Extraction Performance*

Motivation: The effectiveness of programming tutorials can be heavily compromised by low-resolution videos, where text becomes difficult to read, particularly in a low-resolution setting such as 360p. This not only hinders learning but also poses significant challenges for optical character recognition (OCR) systems used in automated code extraction. We assume the accuracy of an OCR systems will decline as the image resolution decreases. This assumption posses the question of whether advanced deep learning-based image super-resolution techniques, including the enhanced deep super-resolution network (EDSR) and the multi-scale deep super-resolution system (MDSR), offers a potential solution. These methods can substantially enhance the resolution of video frames, potentially improving OCR accuracy, thereby making programming tutorials more readable and learning more effective.

Methodology: To address the research question, we utilized a collection of programming screencasts, as detailed in Sections 3.1 and 3.2. Our approach involved the application of two super-resolution techniques, enhanced deep super-resolution (EDSR) and multi-dimensional super-resolution (MDSR), the details of which are available in the GitHub repository. We conducted a comprehensive set of experiments using four different optical character recognition (OCR) engines, Tesseract, Google Vision, GPT-4V, and Gemini, to compare their effectiveness in extracting code from videos.

We systematically assessed and compared the performance of each OCR engine under varying conditions at three different video resolutions: 360p, 480p, and 720p. Additionally, we investigated the impact of using super-resolution (SR) techniques on OCR accuracy. This involved comparing the OCR results obtained without SR (Tesseract-base) against those enhanced through the application of SR at two different scales, $2\times$ and $4\times$. Our goal is to determine whether the incorporation of SR techniques could significantly improve the accuracy of code extraction from low-resolution video screencasts.

3.8. Evaluation Metrics

To evaluate the accuracy of OCR engines, LLMs, and the processing steps, we used the well-established Levenshtein distance (LD) metric [49]. LD quantitatively evaluates the dissimilarity between two text sequences by calculating the minimum number of operations—insertions, substitutions, and deletions—needed to transform one sequence into the other. In our empirical evaluation, each video V in our dataset is represented as $V = \{f_{1080p}, f_{720p}, f_{480p}, f_{360p}\}$, where $f_{quality}$ denotes a frame of a specified quality. For each frame of a specific quality, we extracted the source code using a specific engine, represented as $OCR_{Engine} = \{OCRed_{1080p}, OCRed_{720p}, OCRed_{480p}, OCRed_{360p}\}$, where $OCRed_{q}uality$ represents the OCRed code using a specific OCR engine applied on a video frame with a specific quality. To compare the extracted code against the manually transcribed ground truth, we employed the normalized Levenshtein distance (NLD). The NLD metric refines LD by normalizing it to a value between zero and one, offering a measure of similarity between the OCR-extracted code and the ground truth. This normalization is crucial for our evaluation, providing a standardized scale for accuracy assessment across different resolutions. The NLD is calculated as follows:

$$NLD(gt_{code}, OCRed_{code}) = 1 - \frac{LD(gt_{code}, OCRed_{code})}{max(len(gt_{code}), len(OCRed_{code}))} \quad (1)$$

where gt_{code} is the manually transcribed ground-truth code from our input code images and $OCRed_{code}$ is the extracted code using an OCR engine or LLM.

4. Empirical Results

4.1. RQ$_1$: The Impact of Image Quality on Code Extraction

Tables 2 and 3 offer a comprehensive analysis of OCR engines (Tesseract and Google Vision) and large language models (GPT-4V and Gemini) in extracting source code from video tutorials, employing token-based and character-based normalized Levenshtein distance (NLD) metrics across varying video resolutions (1080p, 720p, 480p, 360p). The analyses highlight GPT-4V's superior performance, achieving the highest NLD scores in both token and character-based evaluations, demonstrating its exceptional accuracy in code extraction across all video qualities. Gemini and Google Vision also produce cleaner code than Tesseract, showcasing strong capabilities in code extraction, closely matching GPT-4V's performance. In contrast, Tesseract's accuracy markedly decreases with declining video quality, highlighting its sensitivity to noise. However, Google Vision, GPT-4V, and Gemini exhibit outstanding resilience to lower resolutions and noise, maintaining consistent and robust performance. This resilience makes them significantly more reliable for extracting readable and reusable code from video tutorials, especially in conditions where traditional OCR engines like Tesseract underperform due to noise and reduced clarity.

Table 2. Evaluation using token-based normalized Levenshtein distance (NLD) of OCR engines (Tesseract and Vision) and large language models (GPT-4V and Gemini) across different video qualities (1080p, 720p, 480p, 360p).

Quality	Tesseract		Google Vision		GPT-4V		Gemini	
	Median	Average	Median	Average	Median	Average	Median	Average
1080p	0.87	0.85	0.93	0.91	0.97	0.95	0.94	0.91
720p	0.83	0.81	0.93	0.91	0.97	0.95	0.93	0.91
480p	0.62	0.60	0.91	0.89	0.96	0.93	0.93	0.89
360p	0.25	0.31	0.88	0.83	0.93	0.89	0.91	0.86

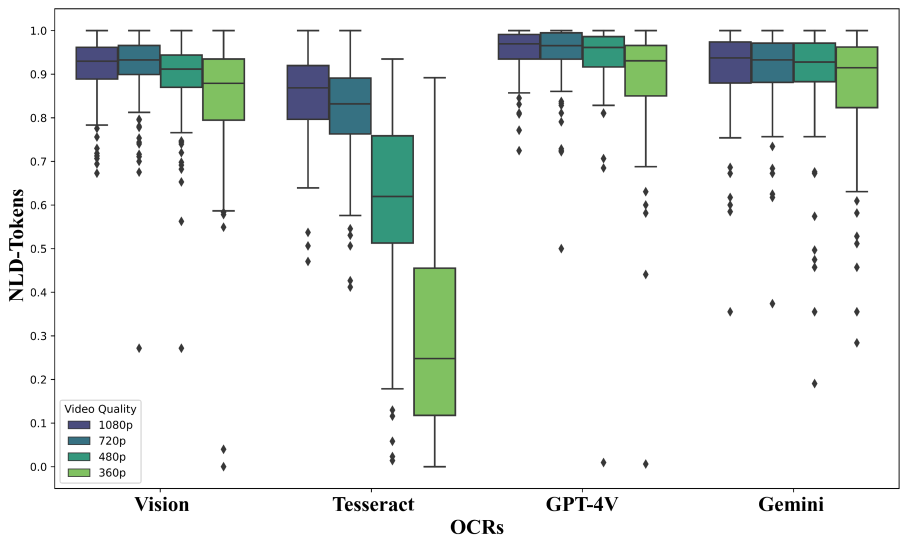
Table 3. Evaluation using character-based normalized Levenshtein distance (NLD) of OCR engines (Tesseract and Google Vision) and large language models (GPT-4V and Gemini) across different video qualities (1080p, 720p, 480p, 360p).

Quality	Tesseract		Google Vision		GPT-4V		Gemini	
	Median	Average	Median	Average	Median	Average	Median	Average
1080p	0.96	0.92	0.96	0.94	0.99	0.96	0.97	0.93
720p	0.94	0.91	0.96	0.94	0.99	0.96	0.96	0.93
480p	0.82	0.77	0.95	0.94	0.98	0.94	0.96	0.91
360p	0.54	0.52	0.95	0.91	0.95	0.92	0.95	0.89

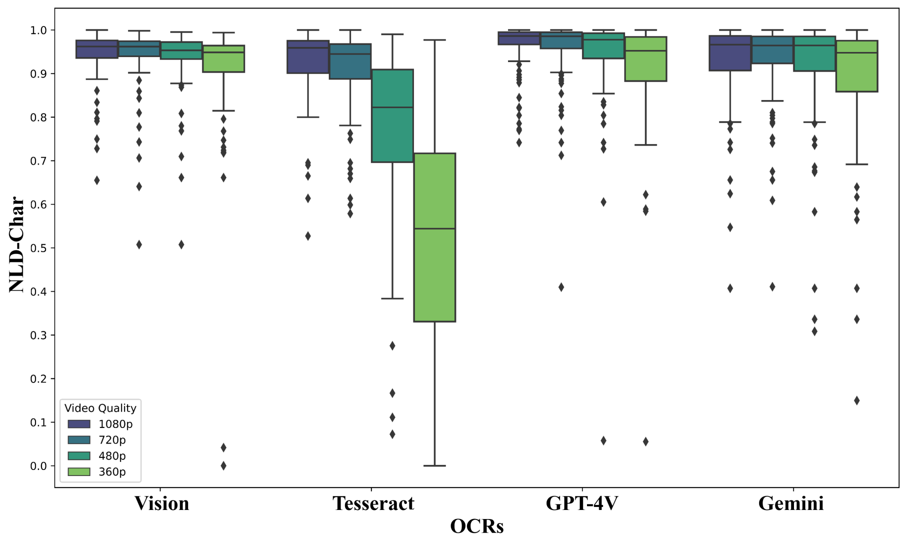
Figure 3 for both token-based and character-based normalized Levenshtein distance (NLD) reveals distinct performance patterns across the OCR engines and language models at various video resolutions. GPT-4V, Gemini, and Google Vision consistently maintain high NLD scores with little fluctuation, evidencing their reliable code extraction across resolutions. Tesseract, however, shows a marked decrease in NLD scores with declining video quality, especially at 360p, highlighting its vulnerability to noise and lower resolutions. This is further illustrated by its wide interquartile ranges and the presence of numerous outliers at lower resolutions, indicating a less-stable performance compared to the other technologies. In sum, while Tesseract's performance is notably affected by video quality, the other engines and models demonstrate a robust ability to accurately extract code from video tutorials, even in less-than-ideal conditions.

Based on the results, we noticed that by using large language models (LLMs) like GPT-4V and Gemini for code extraction from video tutorials, several factors contribute to the issues observed in their performance. One significant challenge is the tendency of these models to "autocomplete" code, an inherent behavior due to their training on predictive text generation. While this feature is valuable in many programming contexts, it can introduce inaccuracies when extracting code from videos, as the models may generate syntactically correct but contextually irrelevant code completions. To mitigate this, we directed the model with an engineered prompt such as *"Act as an OCR that extracts the code from the image without explanation or adding any other information"*.

Moreover, we observed a trend where the performance of all engines, including OCRs and LLMs, degraded as the video quality decreased. This degradation suggests that resolution plays a significant role in the accuracy of code extraction. Notably, the images used for model input were provided without any form of pre-processing, which could potentially enhance the models' ability to interpret and transcribe code accurately. The lack of pre-processing might have limited the engines' performance, especially in lower-resolution conditions where noise and artifacts are more prevalent. Therefore, there is a clear need to experiment with various image pre-processing techniques that could improve the visibility and clarity of code within videos.



(**a**) Token-based NLD



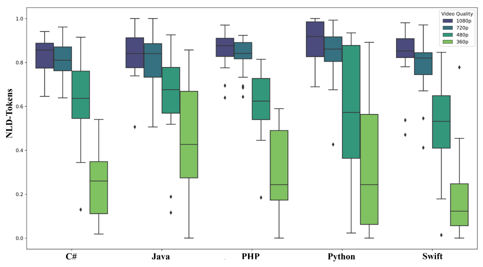
(**b**) Character-based NLD

Figure 3. Boxplots showing how well OCRs and LLMs worked on different image qualities, measured by NLD scores.

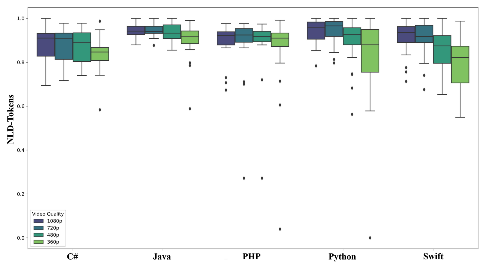
Answer to RQ$_1$: Vision-based LLMs such as GPT-4V demonstrate superior performance over conventional OCR engines like Tesseract in the task of code extraction, excelling across all resolution levels and showing particularly notable improvements in lower-resolution environments.

4.2. RQ$_2$: The Impact of Programming Language Syntax on Code Extraction

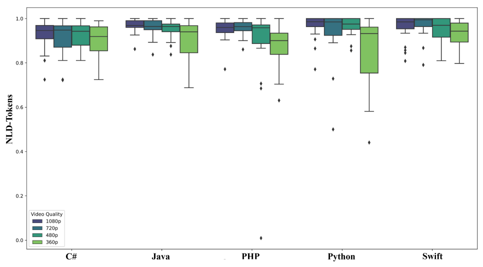
Figure 4 summarizes the results we obtained after applying the two OCR engines and the other two LLMs to each programming language for each quality. As depicted in Figure 4, Google Vision consistently achieves high accuracy across various programming languages and video qualities with minimal variability. Tesseract exhibits moderate performance, particularly excelling in higher-resolution videos. GPT-4V demonstrates robust performance, maintaining high accuracy levels across different video qualities. Gemini (Bard) also performs well, showing consistency in accuracy levels but with a slightly lower mean accuracy compared to GPT-4V. Overall, each OCR engine showcases unique strengths and areas for improvement, with GPT-4V standing out for its consistent high performance.



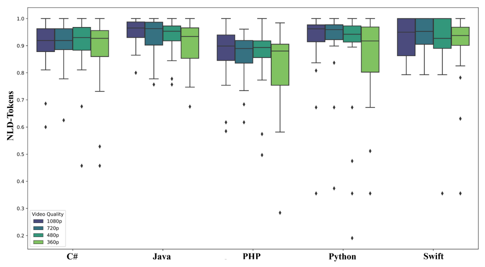
(**a**) Tesseract



(**b**) Google Vision



(**c**) GPT-4V



(**d**) Gemini

Figure 4. Boxplots showing how well OCRs and LLMs worked on different image qualities, measured by NLD-Token scores on different programming languages.

Answer to RQ$_2$: The fluctuating accuracy across programming languages highlights the impact of syntax and structure on OCR efficiency. For example, at lower resolutions, languages like Python exhibit greater variability in accuracy compared to C# for all approaches.

Notably, the performance in terms of programming language varies across all four OCR engines. While some languages consistently yield higher accuracy scores regardless of the OCR engine used, others show more variability. For example, C# and Java tend to exhibit relatively higher accuracy scores across all engines, while PHP and Python show more variability in performance. This suggests that certain programming languages may be better-suited for OCR tasks, potentially due to language-specific syntax or structural characteristics. However, further analysis would be needed to draw conclusive insights into the relationship between programming language and OCR performance.

4.3. RQ$_3$: The Impact of Applying Image Super-Resolution (SR) on Code Extraction Performance

We report our findings for the Tesseract OCR engine in Table 4, Vision in Table 5, GPT-V4 in Table 6, and finally, Gemini in Table 7. Our analysis reveals that image super-resolution (SR) enhances performance across all baseline models, with more observed improvements at lower resolutions, such as 360p compared to 720p, as depicted in Figures 5 and 6.

Table 4. Evaluation using token-based normalized Levenshtein distance (NLD) of **Tesseract** OCR engine on raw and enhanced images using super-resolution techniques across different video qualities (720p, 480p, 360p).

Quality	**Tesseract-Base**		**Tesseract-EDSR-×2**		**Tesseract-EDSR-×4**		**Tesseract-MDSR-×2**		**Tesseract-MDSR-×4**	
	Median	**Average**	**Median**	**Average**	**Median**	**Average**	**Median**	**Average**	**Median**	**Average**
720p	0.83	0.81	0.88	0.86	0.87	0.84	0.87	0.86	0.87	0.84
480p	0.62	0.60	0.83	0.80	0.83	0.80	0.83	0.80	0.83	0.81
360p	0.25	0.31	0.75	0.69	0.75	0.69	0.73	0.68	0.75	0.69

Table 5. Evaluation using token-based normalized Levenshtein distance (NLD) of **Vision OCR engine** on raw and enhanced images using super-resolution techniques across different video qualities (720p, 480p, 360p).

Quality	Vision-Base		Vision-EDSR-×2		Vision-EDSR-×4		Vision-MDSR-×2		Vision-MDSR-×4	
	Median	Average	Median	Average	Median	Average	Median	Average	Median	Average
720p	0.93	0.91	0.94	0.89	0.93	0.90	0.94	0.90	0.93	0.91
480p	0.91	0.89	0.92	0.89	0.92	0.89	0.92	0.89	0.92	0.89
360p	0.88	0.83	0.88	0.84	0.89	0.84	0.89	0.84	0.90	0.84

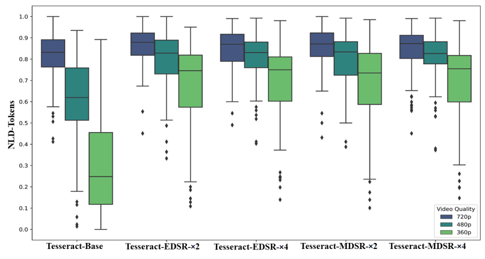
Table 6. Evaluation using token-based normalized Levenshtein distance (NLD) of **GPT-4V** on raw and enhanced images using super-resolution techniques across different video qualities (720p, 480p, 360p).

Quality	GPT-Base		GPT-EDSR-×2		GPT-EDSR-×4		GPT-MDSR-×2		GPT-MDSR-×4	
	Median	Average	Median	Average	Median	Average	Median	Average	Median	Average
720p	0.97	0.95	0.97	0.96	0.97	0.95	0.97	0.95	0.96	0.95
480p	0.96	0.93	0.96	0.94	0.96	0.95	0.97	0.94	0.96	0.95
360p	0.93	0.89	0.96	0.94	0.96	0.94	0.96	0.94	0.95	0.93

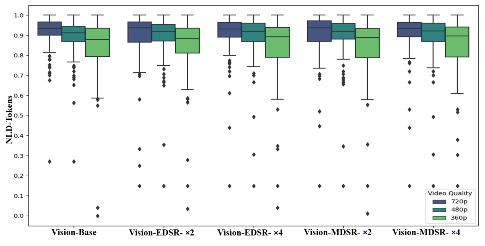
Table 7. Evaluation using token-based normalized Levenshtein distance (NLD) of **Gemini** on raw and enhanced images using super-resolution techniques across different video qualities (720p, 480p, 360p).

Quality	Gemini-Base		Gemini-EDSR-×2		Gemini-EDSR-×4		Gemini-MDSR-×2		Gemini-MDSR-×4	
	Median	Average	Median	Average	Median	Average	Median	Average	Median	Average
720p	0.93	0.91	0.94	0.91	0.94	0.90	0.94	0.90	0.94	0.91
480p	0.93	0.89	0.93	0.90	0.94	0.90	0.94	0.89	0.94	0.90
360p	0.91	0.86	0.92	0.88	0.93	0.88	0.93	0.89	0.93	0.88

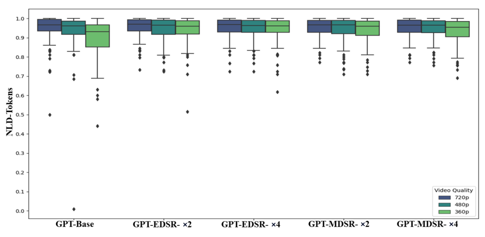
Answer to RQ$_3$: The effectiveness of super-resolution (SR) in enhancing OCR accuracy scales with the decrease in video quality, with the most significant gains, up to a 200% increase in performance, seen at the 360p resolution.



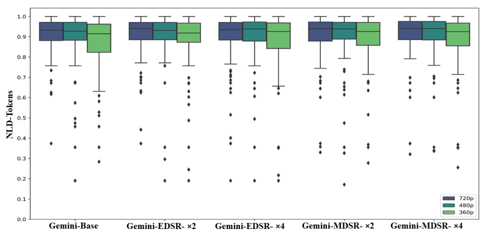
(**a**) Tesseract



(**b**) Google Vision

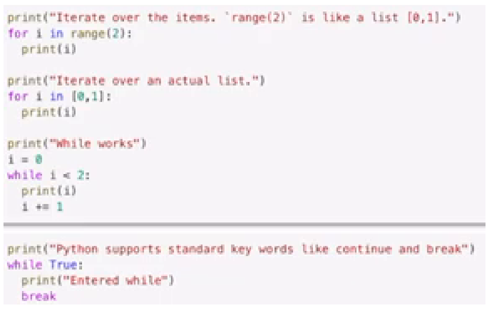


(**c**) GPT-4V



(**d**) Gemini

Figure 5. Boxplots showing how well OCRs and LLMs worked on different image qualities, measured by NLD-Token scores on pre-processed images using super-resolution.



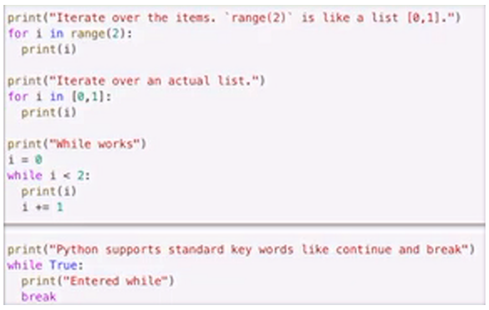
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**a**) Input image without processing (360p)



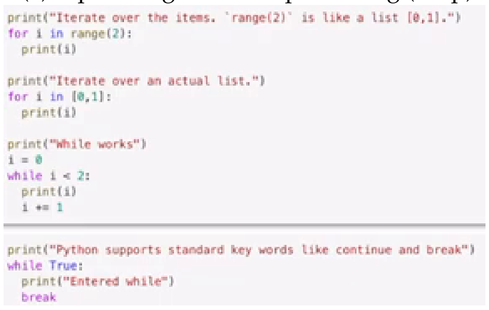
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**b**) Processed image EDSR-× 2 (360p)



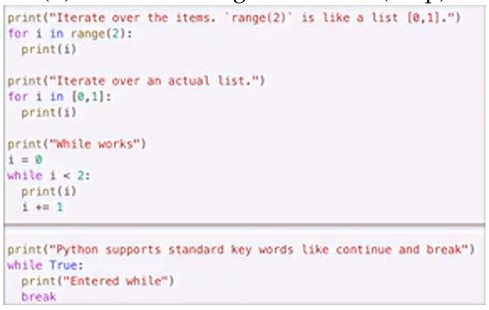
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**c**) Input image without processing (360p)



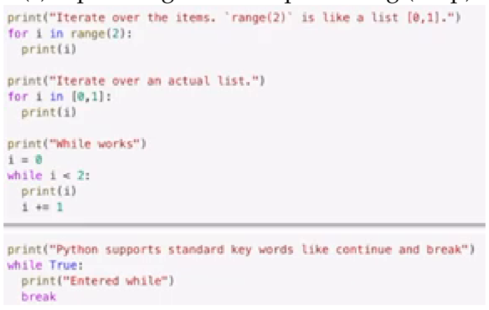
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**d**) Processed image EDSR-×4 (360p)



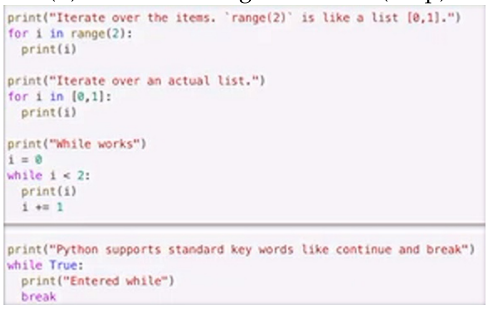
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**e**) Input image without processing (360p)



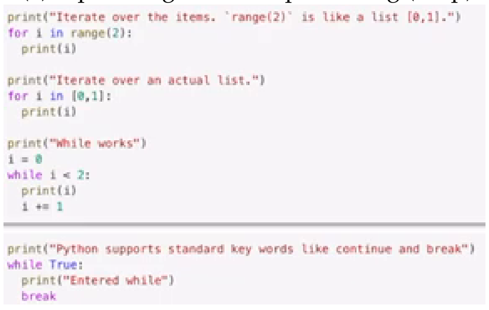
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**f**) Processed image MDSR-×2 (360p)



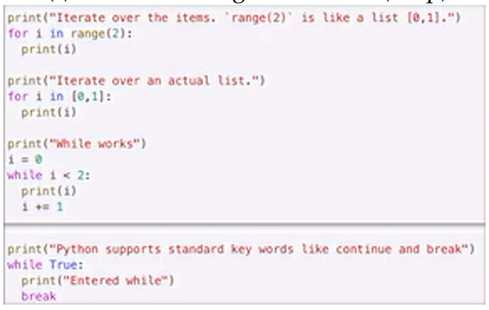
```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**g**) Input image without processing (360p)



```
print("Iterate over the items. `range(2)` is like a list [0,1].")
for i in range(2):
  print(i)

print("Iterate over an actual list.")
for i in [0,1]:
  print(i)

print("While works")
i = 0
while i < 2:
  print(i)
  i += 1
```

```
print("Python supports standard key words like continue and break")
while True:
  print("Entered while")
  break
```

(**h**) Processed image MDSR-×4 (360p)

Figure 6. A sample of Python code images with a 360p resolution processed using EDSR-×2 and EDSR-×4 as part of our super-resolution techniques.

Specifically, at 360p resolution, SR significantly boosts OCR models' performance. For instance, the Tesseract-Base showed a 200% improvement in median and 122.5% in average scores at 360p. Conversely, the improvement in large language models (LLMs) was less striking, with the highest enhancement being a 3% median and 17% average increase in the Gemini model. This lesser degree of improvement in LLMs can be attributed to their inherent robustness, resulting in minimal performance variation across different image qualities. In contrast, Tesseract proved to be more sensitive to image resolution, thus benefiting more from the SR image enchantment.

Moreover, our observations indicate that the application of the SR technique yields the greatest improvement when implemented at a 2× scale, while lesser effects are noted at higher scales, such as 4×.

Notably, GPT-V4 consistently outperformed other models across all image resolutions and in all its variants. The top-performing models were GPT-EDSR-×2 and GPT-MDSR-×2, further underscoring the advanced capabilities of GPT-V4 in varied resolution scenarios.

5. Threats to Validity

Our study encountered a few primary challenges that could impact the reliability of our findings. We break these challenges into internal, construct, and external validity concerns and explain them below.

5.1. Internal Validity

Firstly, we had to manually determine the area covering the source code in each video frame, ensuring it encompassed all the code while excluding any extraneous elements like line numbers. To address this, each frame was annotated by one researcher and then cross-checked by another. Secondly, the transcription of the source code needed to be error-free. To ensure accuracy, three individuals transcribed the code, after which one researcher reviewed and corrected it, followed by a final verification by a second researcher.

5.2. Construct Validity

As for the construct validity, it pertains to the accuracy metric used in evaluating the two OCR engines and the other two LLMs models with the two super-resolution techniques. We addressed this potential issue by using established edit-based metrics, commonly utilized in various research fields for assessing OCR performance. Given that OCR errors often occur at the character level and could be at the token level as well, we employed both the character and token level in reporting the accuracy of our comprehensive experiments.

5.3. External Validity

Regarding external validity, our OCR evaluations and super-resolution techniques may not cover all possible scenarios in terms of programming languages, topics, IDE themes, numbers of lines of code, and image resolutions. To mitigate these threats, we (i) selected five different programming languages, (ii) formulated various queries for searching videos, (iii) selected a frame with at least ten lines of code, and (iv) selected videos with black and white background IDE that could be downloaded in four resolutions.

6. Conclusions

In the evolving landscape of software engineering, video programming tutorials have become a crucial educational resource, offering step-by-step visual instructions that demonstrate programming concepts and practices. However, the code embedded within these video frames is often inaccessible for direct reuse, presenting a significant challenge in the field. This study addresses the critical task of extracting source code from such video tutorials, focusing on the impact of image quality on optical character recognition (OCR) and the efficacy of large language models (LLMs) in this context. We present a comprehensive empirical analysis across various video resolutions, evaluating the performance of traditional OCR engines and advanced LLMs, and examining the enhancement potential of image super-resolution (SR) techniques.

Our findings reveal that vision-based large language models (LLMs) like GPT-4V significantly outperform traditional OCR engines such as Tesseract in code extraction tasks. We also found that the syntax and structure of different programming languages considerably impact OCR efficiency. Furthermore, our study highlights the pivotal role of super-resolution (SR) techniques in enhancing OCR accuracy, particularly in lower-quality videos, with performance gains up to 200% observed at 360p resolution. These findings suggest a shift towards integrating more advanced technologies like LLMs and SR in processing educational video content, paving the way for more effective and accessible programming learning resources. This research contributes to the software engineering domain by providing a benchmark for code extraction from video tutorials and showcasing the significant role of SR and LLMs in improving code readability and extraction accuracy.

While our study makes significant strides in understanding code extraction from video tutorials, it does have limitations. For instance, the focus on a select number of programming languages and video qualities might not fully represent the diverse scenarios encountered in software engineering education. Additionally, the manual transcription process, though thorough, introduces human subjectivity. Future work should look into broadening the scope to include a wider array of programming languages and video qualities. Exploring automated methods for transcription and labeling could also enhance the objectivity and scalability of the dataset creation. Moreover, there is an opportunity to integrate and test emerging OCR and LLM technologies, continually refining the tools and techniques for more effective code extraction from educational video content. This ongoing evolution will further contribute to the advancement of learning resources in software engineering.

Author Contributions: Conceptualization, M.A.; Methodology, M.D.A.; Software, M.A.; Formal analysis, M.A.; Data curation, M.A.; Writing—original draft, M.D.A.; Writing—review & editing, M.D.A. and M.A.; Supervision, M.D.A.; Funding acquisition, M.D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the University of Jeddah, Jeddah, Saudi Arabia, under grant No. (UJ-23-RSP-2). The authors, therefore, thank the University of Jeddah for its technical and financial support.

Data Availability Statement: The data presented in this study are openly available in (https://zenodo.org/records/10823097).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Brandt, J.; Guo, P.J.; Lewenstein, J.; Dontcheva, M.; Klemmer, S.R. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, New York, NY, USA, 4–9 April 2009; CHI '09, pp. 1589–1598. [CrossRef]
2. Grzywaczewski, A.; Iqbal, R. Task-specific information retrieval systems for software engineers. *J. Comput. Syst. Sci.* **2012**, *78*, 1204–1218. [CrossRef]
3. Storey, M.A.; Singer, L.; Cleary, B.; Figueira Filho, F.; Zagalsky, A. The (R) evolution of social media in software engineering. In *Future of Software Engineering*; FOSE: New York, NY, USA, 2014; pp. 100–116. [CrossRef]
4. MacLeod, L.; Bergen, A.; Storey, M.A. Documenting and sharing software knowledge using screencasts. *Empir. Softw. Eng.* **2017**, *22*, 1478–1507. [CrossRef]
5. Lin, Y.T.; Yeh, M.K.C.; Tan, S.R. Teaching Programming by Revealing Thinking Process: Watching Experts' Live Coding Videos With Reflection Annotations. *IEEE Trans. Educ.* **2022**, *65*, 617–627. [CrossRef]
6. Pongnumkul, S.; Dontcheva, M.; Li, W.; Wang, J.; Bourdev, L.; Avidan, S.; Cohen, M.F. Pause-and-play: Automatically linking screencast video tutorials with applications. In Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, Santa Barbara, CA, USA, 16–19 October 2011; ACM: New York, NY, USA, 2011; pp. 135–144.
7. Khandwala, K.; Guo, P.J. 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, London, UK, 26–28 June 2018; pp. 1–10. [CrossRef]
8. Ponzanelli, L.; Bavota, G.; Mocci, A.; Di Penta, M.; Oliveto, R.; Russo, B.; Haiduc, S.; Lanza, M. CodeTube: Extracting relevant fragments from software development video tutorials. In Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE'16), Austin, TX, USA, 14–22 May 2016; pp. 645–648.
9. Ponzanelli, L.; Bavota, G.; Mocci, A.; Oliveto, R.; Di Penta, M.; Haiduc, S.C.; Russo, B.; Lanza, M. Automatic identification and classification of software development video tutorial fragments. *IEEE Trans. Softw. Eng.* **2017**, *45*, 464–488. [CrossRef]
10. Bao, L.; Xing, Z.; Xia, X.; Lo, D.; Wu, M.; Yang, X. psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2020**, *29*, 1–38. [CrossRef]
11. Alahmadi, M.; Khormi, A.; Parajuli, B.; Hassel, J.; Haiduc, S.; Kumar, P. Code Localization in Programming Screencasts. *Empir. Softw. Eng.* **2020**, *25*, 1536–1572. [CrossRef]
12. Khormi, A.; Alahmadi, M.; Haiduc, S. A Study on the Accuracy of OCR Engines for Source Code Transcription from Programming Screencasts. In Proceedings of the 17th IEEE/ACM Working Conference on Mining Software Repositories, Seoul, Republic of Korea, 25–26 May 2020; pp. 376–386.

13. Malkadi, A.; Tayeb, A.; Haiduc, S. Improving code extraction from coding screencasts using a code-aware encoder-decoder model. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Kirchberg, Luxembourg, 11–15 September 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1492–1504.
14. Alahmadi, M.D. VID2XML: Automatic Extraction of a Complete XML Data From Mobile Programming Screencasts. *IEEE Trans. Softw. Eng.* **2022**, *49*, 1726–1740. [CrossRef]
15. Alahmadi, M.D. VID2META: Complementing Android Programming Screencasts with Code Elements and GUIs. *Mathematics* **2022**, *10*, 3175. [CrossRef]
16. Yadid, S.; Yahav, E. 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), Amsterdam, The Netherlands, 2–4 November 2016; pp. 98–111.
17. Perianez-Pascual, J.; Rodriguez-Echeverria, R.; Burgueño, L.; Cabot, J. Towards the optical character recognition of DSLs. In Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, 16–17 November 2020; pp. 126–132.
18. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* **2021**, arXiv:2109.00859.
19. Shetty, A.; Sharma, S. Ensemble deep learning model for optical character recognition. *Multimed. Tools Appl.* **2024**, *83*, 11411–11431. [CrossRef]
20. Elanwar, R.; Qin, W.; Betke, M. Making scanned Arabic documents machine accessible using an ensemble of SVM classifiers. *Int. J. Doc. Anal. Recognit. (IJDAR)* **2018**, *21*, 59–75. [CrossRef]
21. Semkovych, V.; Shymanskyi, V. Combining OCR Methods to Improve Handwritten Text Recognition with Low System Technical Requirements. In Proceedings of the The International Symposium on Computer Science, Digital Economy and Intelligent Systems, Wuhan, China, 16–17 November 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 693–702.
22. Bao, L.; Pan, P.; Xing, X.; Xia, X.; Lo, D.; Yang, X. Enhancing Developer Interactions with Programming Screencasts through Accurate Code Extraction. In Proceedings of the 28th ACM/SIGSOFT International Symposium on Foundations of Software Engineering (FSE'20), Sacramento, CA, USA, 13–18 November 2020.
23. Moslehi, P.; Adams, B.; Rilling, J. A feature location approach for mapping application features extracted from crowd-based screencasts to source code. *Empir. Softw. Eng.* **2020**, *25*, 4873–4926. [CrossRef]
24. Ott, J.; Atchison, A.; Harnack, P.; Bergh, A.; Linstead, E. 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, Gothenburg Sweden, 29–28 May 2018; pp. 376–386.
25. Ott, J.; Atchison, A.; Harnack, P.; Best, N.; Anderson, H.; Firmani, C.; Linstead, E. Learning Lexical Features of Programming Languages from Imagery Using Convolutional Neural Networks. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 27 May–3 June 2018.
26. Ott, J.; Atchison, A.; Linstead, E.J. Exploring the applicability of low-shot learning in mining software repositories. *J. Big Data* **2019**, *6*, 35. [CrossRef]
27. Zhao, D.; Xing, Z.; Chen, C.; Xia, X.; Li, G.; Tong, S.J. ActionNet: Vision-based workflow action recognition from programming screencasts. In Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19), Montreal, QC, Canada, 27 May 2019.
28. Szegedy, C.; Ioffe, S.; Vanhoucke, V.; Alemi, A. Inception-v4, Inception-ResNet and the impact of residual connections on learning. *arXiv* **2016**, arXiv:1602.07261.
29. Parra, E.; Escobar-Avila, J.; Haiduc, S. Automatic tag recommendation for software development video tutorials. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg Sweden, 28–29 May 2018; ACM: New York, NY, USA, 2018; pp. 222–232.
30. Moslehi, P.; Adams, B.; Rilling, J. Feature location using crowd-based screencasts. In Proceedings of the 15th International Conference on Mining Software Repositories—MSR '18, Gothenburg, Sweden, 28–29 May 2018; pp. 192–202. [CrossRef]
31. Poché, E.; Jha, N.; Williams, G.; Staten, J.; Vesper, M.; Mahmoud, A. Analyzing user comments on YouTube coding tutorial videos. In Proceedings of the 25th International Conference on Program Comprehension, Buenos Aires, Argentina, 22–23 May 2017; IEEE Press: Piscataway, NJ, USA, 2017; pp. 196–206.
32. Ponzanelli, L.; Bavota, G.; Mocci, A.; Di Penta, M.; Oliveto, R.; Hasan, M.; Russo, B.; Haiduc, S.; Lanza, M. CodeTube: Extracting relevant fragments from software development video tutorials. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; ACM Press: New York, NY, USA, 2016; pp. 261–272. [CrossRef]
33. Vahedi, M.; Rahman, M.M.; Khomh, F.; Uddin, G.; Antoniol, G. Summarizing Relevant Parts from Technical Videos. In Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 434–445.
34. Bao, L.; Xing, Z.; Xia, X.; Lo, D. VT-Revolution: Interactive programming video tutorial authoring and watching system. *IEEE Trans. Softw. Eng.* **2018**, *45*, 823–838. [CrossRef]
35. Bao, L.; Xing, Z.; Xia, X.; Lo, D.; Li, S. VT-revolution: Interactive programming tutorials made possible. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 924–927.

36. Xu, F.F.; Alon, U.; Neubig, G.; Hellendoorn, V.J. A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022; pp. 1–10.
37. Chaaben, M.B.; Burgueño, L.; Sahraoui, H. Towards using Few-Shot Prompt Learning for Automating Model Completion. *arXiv* **2022**, arXiv:2212.03404.
38. Kang, S.; Yoon, J.; Yoo, S. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *arXiv* **2022**, arXiv:2209.11515.
39. Sobania, D.; Briesch, M.; Hanna, C.; Petke, J. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. *arXiv* **2023**, arXiv:2301.08653.
40. Akli, A.; Haben, G.; Habchi, S.; Papadakis, M.; Traon, Y.L. Predicting Flaky Tests Categories using Few-Shot Learning. *arXiv* **2022**, arXiv:2208.14799.
41. Lyu, Y.; Le-Cong, T.; Kang, H.J.; Widyasari, R.; Zhao, Z.; Le, X.B.D.; Li, M.; Lo, D. Chronos: Time-aware zero-shot identification of libraries from vulnerability reports. *arXiv* **2023**, arXiv:2301.03944.
42. Le, V.H.; Zhang, H. Log Parsing with Prompt-based Few-shot Learning. *arXiv* **2023**, arXiv:2302.07435.
43. Nashid, N.; Sintaha, M.; Mesbah, A. Retrieval-based prompt selection for code-related few-shot learning. In Proceedings of the 45th International Conference on Software Engineering (ICSE'23), Melbourne, Australia, 14–20 May 2023.
44. Siddiq, M.L.; Samee, A.; Azgor, S.R.; Haider, M.A.; Sawraz, S.I.; Santos, J.C. Zero-shot Prompting for Code Complexity Prediction Using GitHub Copilot. In Proceedings of the 2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE), Melbourne, Australia, 20 May 2023.
45. Lim, B.; Son, S.; Kim, H.; Nah, S.; Lee, K.M. Enhanced Deep Residual Networks for Single Image Super-Resolution. In Proceedings of the CVPR Workshops, Honolulu, HI, USA, 21–26 July 2017; IEEE Computer Society: Piscataway, NJ, USA; pp. 1132–1140.
46. Dong, C.; Loy, C.C.; Tang, X. Accelerating the Super-Resolution Convolutional Neural Network. In *Lecture Notes in Computer Science* ; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9906, pp. 391–407.
47. Kim, J.; Lee, J.K.; Lee, K.M. Accurate Image Super-Resolution Using Very Deep Convolutional Networks. In Proceedings of the CVPR, San Francisco, CA, USA, 18–20 June 2016; IEEE Computer Society: Piscataway, NJ, USA, 2016; pp. 1646–1654.
48. Kim, J.; Lee, J.K.; Lee, K.M. Deeply-Recursive Convolutional Network for Image Super-Resolution. In Proceedings of the CVPR, Las Vegas, NV, USA, 27–30 June 2016; IEEE Computer Society: Piscataway, NJ, USA, 2016; pp. 1637–1645.
49. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.