Journal of Informatics
Electrical and Electronics Engineering, 2023,
Vol. 00, Iss. 00, S. No. 000, pp. 0-00
ISSN (Online): 2582-7006

Optical Character Recognition Development Using Python

Prakhar Sisodia[1], Syed Wajahat Abbas Rizvi[2]

[1,2]Department of Computer Science and Engineering, Amity School of Engineering and Technology Lucknow, Amity University Uttar Pradesh, India
[1]prakhar.sisodia22@gmail.com, [2]swarizvi@lko.amity.edu

How to cite this paper: P. Sisodia and S. W. A. Rizvi, "Optical Character Recognition Development Using Python," *Journal of Informatics Electrical and Electronics Engineering* (*JIEEE*).

https://doi.org/10.54060/jieee.2023.75

Received: 01/04/2023
Accepted: 01/06/2023
Online First: 08/08/2023

Abstract

Optical Character Recognition (OCR) is a technology used to convert scanned or digital images into editable text. OCR has become an increasingly important tool in the fields of data extraction and information retrieval, allowing for quick and efficient conversion of scanned documents and digital images into text. In this paper, we explore the use of the Python programming language to implement OCR algorithms and systems. We provide a comprehensive overview of existing Python libraries and packages used for OCR, including Tesseract and pytesseract, along with their strengths and limitations. We also examine the different OCR approaches and techniques, including template matching, feature extraction, and encrypting/decrypting the OCR parsed files and discuss their implementation in Python. Finally, we present a case study of a simple OCR system built using Python and evaluate its performance on a sample dataset. The results of our study highlight the potential of Python for OCR implementation and demonstrate its feasibility for real-world applications.

Keywords

pytesseract, flask, opencv-python, pypdf, pdfplumber, pymupdf, scikit-learn scipy matplotlib, youtbe-dl and shutil

1. Introduction

Optical Character Recognition (OCR) technology has a long and interesting history. The first OCR systems were developed in the early 1900s, but they were limited in their ability to recognize text accurately. In the 1960s, OCR technology began to evolve rapidly with the advent of computers, leading to the development of more advanced OCR systems. During the 1970s, OCR systems became more sophisticated, incorporating features such as document layout analysis and font recognition. In

the 1980s, OCR technology continued to advance, with the development of more advanced algorithms for character recognition, as well as the introduction of OCR software for personal computers. The 1990s saw the rise of digital imaging and the widespread adoption of OCR technology in a variety of industries, including government, finance, and healthcare [1-2].

In recent years, OCR technology has advanced significantly with the advent of deep learning and machine learning algorithms. Today, OCR systems can recognize text in a variety of languages and scripts and are used in a wide range of ap- plications, from document scanning and archiving to data extraction and information retrieval. The development development of OCR technology is expected to further improve its accuracy and efficiency, making it an increasingly important tool for businesses and individuals alike [2-6].

Some recent developments in the field of OCR includes Improved Text Recognition in Complex Scenes, Handwritten Text Recognition, Integration with Augmented Reality, Integration with Internet of Things (IoT). The section of this paper is organized as follows, and part 2 contains related works on optical character recognition his- tory and its presence in python programming language. In section 3, the methodology adopted with driving code for the OCR process, section 4 reviews the mechanism of OCR engine along with future scope of OCR development using py- thon. Section 5 concludes the paper with future research.

2. Problem Analysis

OCR (Optical Character Recognition) is a technology that allows the recognition of text within digital images or scanned documents, and the conversion of that text into machine-readable characters. While OCR technology has made significant advancements in recent years, there are still several common problems associated with its use, including: Quality of the Source Image: The quality of the source image is critical for accurate OCR. Poor quality images with low resolution, blurred text, or skewing can make it difficult for OCR software to recognize characters correctly. Character Recognition Errors: OCR technology can sometimes misinterpret characters, leading to errors in the output. For example, similar looking characters such as 'l' and '1' or 'O' and '0' can be easily misinterpreted [1, 8-10].

Formatting Issues: OCR software can struggle with formatting issues, such as columns, tables, and font styles, which can make it difficult to accurately recognize and convert text language and Character Set Support: OCR software may not support all languages and character sets, which can result in inaccurate recognition of characters or an inability to recognize them at all.

Handwriting Recognition: OCR technology struggles with handwritten text recognition. Even with the best OCR software, it's challenging to recognize handwriting with high accuracy. Noise and Distortion: Noise and distortion in the source im- age, such as smudges, stains, and creases, can cause OCR technology to misinterpret characters or fail to recognize them at all. Overall, OCR technology has made significant advancements in recent years, but there are still several challenges that need to be addressed to improve its accuracy and reliability [4].

3. Literature Review

Optical Character Recognition (OCR) using Python provides an overview of the various Python libraries and packages available for OCR, as well as the current state of the art in OCR using Python. One of the most widely used OCR libraries in Python is Tesseract, which is an open-source OCR engine developed by Google. Tesseract provides a high level of accuracy and supports a variety of languages and scripts, making it a popular choice for OCR applications. The Python binding for Tesseract, pytesseract, provides a simple interface for integrating Tesseract into Python applications. Another popular OCR library in Python is OpenCV, which is an open-source computer vision library. OpenCV provides a range of image processing and computer vision algorithms, including object detection and segmentation, which can be used to improve the accuracy of OCR.

The integration of OpenCV with Tesseract or pytesseract provides a powerful tool for OCR applications. Other OCR libraries in Python include OCRopus, a Python-based OCR engine developed by Google, and pyOCR, a Python wrapper for the Tesseract OCR engine [9-14]. These libraries provide alternative options for OCR implementation in Python and offer different levels of functionality and accuracy. In recent years, there has been growing interest in the use of deep learning algorithms for OCR. Python provides a number of deep learning libraries, such as TensorFlow and PyTorch, which can be used to build OCR systems. These libraries allow for the training of deep learning models for OCR and can be used to improve the accuracy and efficiency of OCR systems. Overall, the literature survey highlights the versatility of Python for OCR implementation, with a range of libraries and packages available for OCR, including Tesseract and OpenCV, as well as deep learning libraries such as TensorFlow and PyTorch. Python provides a simple and flexible platform for OCR implementation, making it an attractive option for OCR applications in a variety of domains.

4. Aims and Objectives

Optical Character Recognition (OCR) using Python provides an overview of the various Python libraries and packages available for OCR, as well as the current state of the art in OCR using Python. One of the most widely used OCR libraries in Python is Tesseract, which is an open-source OCR engine developed by Google. Tesseract provides a high level of ac- curacy and supports a variety of languages and scripts, making it a popular choice for OCR applications. The Python binding for Tesseract, pytesseract, provides a simple interface for integrating Tesseract into Python applications. Another popular OCR library in Python is OpenCV, which is an open-source computer vision library. OpenCV provides a range of image processing and computer vision algorithms, including object detection and segmentation, which can be used to improve the accuracy of OCR. The integration of OpenCV with Tesseract or pytesseract provides a powerful tool for OCR applications. Other OCR libraries in Python include OCRopus, a Python-based OCR engine developed by Google, and pyOCR, a Python wrapper for the Tesseract OCR engine. These libraries provide alternative options for OCR implementation in Python and offer different levels of functionality and accuracy. In recent years, there has been growing interest in the use of deep learning algorithms for OCR.

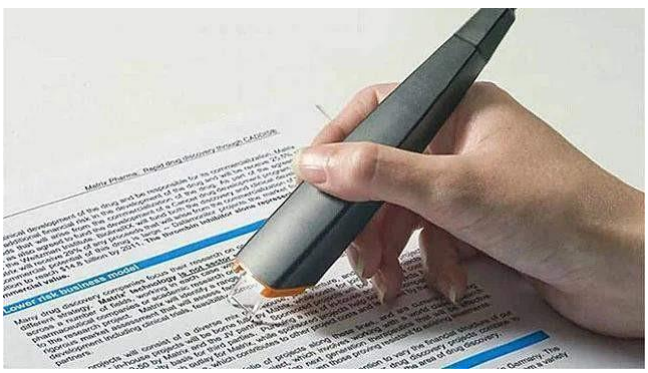


Figure 1. Optical Character Recognition

Python provides a few deep learning libraries, such as TensorFlow and PyTorch, which can be used to build OCR systems. These libraries allow for the training of deep learning models for OCR and can be used to improve the accuracy and efficiency of OCR systems. Overall, the literature survey highlights the versatility of Python for OCR implementation, with a range of libraries and packages available for OCR, including Tesseract and OpenCV, as well as deep learning libraries such as TensorFlow and PyTorch. Python provides a simple and flexible platform for OCR implementation, making it an attractive option for OCR applications in a variety of domains.

Overall, the main aim and objectives of OCR technology are to automate the recognition and conversion of text-based data, improving the efficiency, accuracy, and accessibility of text-based information. OCR technology has made significant advancements in recent years, but there are still several challenges that need to be addressed to improve its accuracy and reliability.

5. Methodology Adopted

The following (packages/modules) would need to be installed in order to build the OCR Application successfully and achieve the desired results from each of the added functionalities. These include Apache Tika, Requests, Warnings, Pytesseract, PIL, OS, along with I/O, flask, opencvpython, pypdf, pdfplumber, pymupdf, scikit-learn scipy matplotlib, youtube_dl and shutil too. Where necessary, we will import them and use them appropriately. Apache Tika is a library that is used to identify report types and extract content from various document designs. Tika uses various record parsers that are already in use as well as archive-type information discovery techniques to locate and segregate content inside. With the help of Tika, one may develop a general-purpose kind finder and content extractor to partially remove organized text and metadata from a variety of records, including calculation sheets, text reports, photos, PDFs, and even setups for visual and aural information. Tika provides a single traditional programming interface for parsing various record designs. For each archive format, specific parser libraries already exist. This vast array of parser libraries is encapsulated by a single point of contact known as the Parser interface.

Before starting for approach, we will use warnings module of python again, Warning - is not the same as mistake in a program. If a mistake occurs, the Python application terminates immediately. It isn't deadly to Caution then again. Despite showing a specified message, the programme continues. Warnings are issued to caution the patron of a particular circumstances which aren't precisely exemptions. Regularly A warning appears on the off chance that A warning appears utilization of programming component such as /capability/class and so forth is found. is an OCR component of python. In other words, it will recognize and "read" the text included in the images. The Tesseract-OCR engine from Google is covered by Python-tesseract. It can read all image formats supported by the Cushion and Leptonic imaging libraries, including jpeg, png, gif, bmp, spat, and others, making it useful as a standalone conjuring con- tent to tesseract. Additionally, if used as content, Python-tesseract will print the perceived text rather than writing it to a document. The PIL library is an open-source library for handling and transforming image files in Python. The OS library allows us to interact with the system's operating system files to get the job done without OS having to interfere with the processes of the working code to give the desired results. The Python's io module helps to deal with document related information and result activities. Pypdf is a python open-source pdf library which allows every operation associated to a pdf done by using its methods/functions present inside the library.

5.1. Working

```
from tika import parser as p
import requests
import tika
import warnings
```

Figure 2. Importing of required components

This code imports the Tika parser module, the requests library, and the Tika library itself. The Tika library provides a toolkit

for detecting and extracting metadata and text content from various types of files, including PDFs, Microsoft Office documents, and other popular formats. The parser module within Tika provides a simple interface for using Tika to parse files and extract their content. The requests library is a popular Python library for making HTTP requests, including downloading files from the internet. While it is not strictly necessary for using Tika, it can be useful for downloading files to parse with Tika. Overall, this code sets up the necessary libraries to use Tika for parsing files, and it also provides a way to download files from the internet if needed.

Access to Files

Different Ways to access the file:

```
# 1. From internet
url = "the url to your file"
response = requests.get(url)
results = p.from_buffer(response.content)

# 2. From the local destination
file_path = "path to your file"
results = p.from_file(file_path)
```

Figure 3. Ways to access filetypes.

The first code snippet shows how to download a file from the internet using the requests library, and then pass its contents to the Tika parser to extract text. You will need to replace "the url to your file" with the actual URL of the file you want to download. The second code snippet shows how to extract text from a local file on your computer using the Tika parser. You will need to replace "path to your file" with the actual path to the file you want to parse. In both cases, the result of the parsing operation will be stored in the "results" variable as a dictionary containing various metadata about the file, as well as the extracted text content.

```
def get_data_from_web(url):
    response = requests.get(url)
    results = p.from_buffer(response.content)
    return results

def get_data_from_given_path(file_path):
    results = p.from_file(file_path)
    return results
```

Figure 4. Definition of helping functions

These are two functions that use the Tika parser module to extract text from files, either from a web URL or from a local file path. The "get_data_from_web" function takes a URL as input, downloads the file contents using the requests library, passes them to the Tika parser using the "from_buffer" method, and returns the results as a dictionary. The "get_data_from_given_path" function takes a file path as input, uses the Tika parser to extract text from the file using the

"from_file" method, and returns the results as a dictionary. Both functions return the results as a dictionary containing various metadata about the file, as well as the extracted text content.

5.2. PDF document

5.2.1. From the web

```
pdf_url = "https://www.bl.uk/learning/resources/pdf/makeanimpact/sw-transcripts.pdf"
results = get_data_from_web(pdf_url)
print("File Content: \n{}".format(results["content"].strip()))
```

Figure 5. Pseudo code for the web PDF

Here we have defined the "get_data_from_web" function as shown in the previous message, this code downloads the PDF file located at "pdf_url", extracts its text content using the Tika parser, and then prints the text content to the console.
The "results" variable should contain a dictionary with metadata about the file and the extracted text content. You can access the extracted text content using the "content" key of the dictionary, and then strip any leading or trailing whitespace using the "strip" method before printing it to the console.The Tika parser may take some time to process the PDF file, especially if it is large or has complex formatting. The resulting text may also contain errors or formatting issues depending on the quality of the original PDF file.

5.2.2. From the local destination

```
pdf_file_path = "./data/PASSENGER DISCLOSURE AND ATTESTATION.pdf"
results = get_data_from_given_path(pdf_file_path)
print(results["content"].strip())
```

Figure 6. Pseudo code for the local PDF

For instance, if we have a dedicated file path in our system then "get_data_from_given_path" function as shown in the previous message, this code extracts the text content from the PDF file located at "pdf_file_path" using the Tika parser, and then prints the text content to the console. The "results" variable should contain a dictionary with metadata about the file and the extracted text content. You can access the extracted text content using the "content" key of the dictionary, and then strip any leading or trailing whitespace using the "strip" method before printing it to the console. The Tika parser may take some time to process the PDF file, especially if it is large or has complex formatting. The resulting text may also contain errors or formatting issues depending on the quality of the original PDF file.

5.3. DOCX document

```
docx_file_path = "./data/covid-19-ets2-sample-employee-choice-vaccination-policy.docx"
results = get_data_from_given_path(docx_file_path)
print(results["content"].strip())
```

Figure 7. Pseudo code for docx document

Assuming that we have docx files then "get_data_from_given_path" function as shown in a previous message, this code ex-

tracts the text content from the Microsoft Word document located at "docx_file_path" using the Tika parser, and then prints the text content to the console. The "results" variable should contain a dictionary with metadata about the file and the extracted text content. You can access the extracted text content using the "content" key of the dictionary, and then strip any leading or trailing whitespace using the "strip" method before printing it to the console. The Tika parser may take some time to process the Word document, especially if it is large or has complex formatting. The resulting text may also contain errors or formatting issues depending on the quality of the original Word document.

5.4. Image document

5.4.1. From the local destination

```
from pytesseract import pytesseract
from PIL import Image
import os

class OCR:
    def __init__(self):
        self.path = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

    def extract(self,filename):
        try:
            pytesseract.tesseract_cmd = self.path

            text = pytesseract.image_to_string(filename)
            return text
        except Exception as e:
            print(e)
            return "Error"

ocr = OCR()
text = ocr.extract("ivory_coast.png")
print(text)
```

Figure 8. Pseudo code for the local image

Here is a Python class called "OCR" that uses the pytesseract library to perform OCR (optical character recognition) on images. The class has an "extract" method that takes a filename as input, sets the path to the Tesseract OCR executable using the "tesseract_cmd" variable, and then uses the "image_to_string" method from the pytesseract library to extract text from the image. The "image_to_string" method performs OCR on the image and returns the extracted text as a string. This text is then returned by the "extract" method. The class also has an "init" method that sets the path to the Tesseract OCR executable. Note that this path may need to be modified depending on the location of the Tesseract OCR executable on your system. In the last two lines of the code, an instance of the OCR class is created, and the "extract" method is called on an image file named "ivory_coast.png". The extracted text is then printed to the console. The accuracy of the OCR process will depend on the quality of the input image, the resolution of the image, and the complexity of the text being recognized. The pytesseract library may also require additional configuration or training to recognize certain types of text or fonts.

5.4.2. From the Web

```
import io
import requests
from PIL import Image

img_url = "https://i.stack.imgur.com/t3qWG.png"
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/103.0.0.0 Safari/537.36'}
r = requests.get(img_url, headers=headers)
img = Image.open(io.BytesIO(r.content))
text = pytesseract.image_to_string(img)
print(text)
```

Figure 9. Pseudo code for the web image

This code performs OCR on an image that is located at "img_url" on the internet using the pytesseract library. The "requests"

library is used to download the image from the internet, and the "PIL" (Python Imaging Library) is used to open the downloaded image. The "image_to_string" method from the pytesseract library is used to extract text from the opened image, and the resulting text is stored in the "text" variable. Finally, the extracted text is printed to the console. The accuracy of the OCR process will depend on the quality of the input image, the resolution of the image, and the complexity of the text being recognized. The pytesseract library may also require additional configuration or training to recognize certain types of text or fonts.

5.5. Extraction of Contents from PDF

5.5.1. Extracting text from image in PDF

```
#%pip install pypdf
from pypdf import PdfReader
reader = PdfReader("D:\\NTCC\\Major Project\\JIEEE Paper Template-OTH.pdf")
#print(len(reader.pages))
page = reader.pages[0]
#print(page.extract_text())

for i in range(len(reader.pages)):
    page = reader.pages[i]
    print(page.extract_text())

for i in page.images:
    with open(i.name, "wb") as f:
        f.write(i.data)
```

Figure 10. Pseudo code for extracting text from image

Here, the code is attempting to use the "pypdf" library to read a PDF file named "JIEEE Paper Template-OTH.pdf" and extract text and images from the file. The "PdfReader" method from the "pypdf" library is used to read the PDF file, and the "pages" attribute is used to access the individual pages of the PDF. The code then loops through all the pages of the PDF and prints the extracted text for each page to the console using the "extract_text()" method of the "page" object. Next, the code attempts to extract images from the PDF using the "images" attribute of the "page" object. For each image in the page, the code writes the image data to a file with the same name as the image using a "with" statement and the "write" method of the file object. However, there is one thing to note because the "pypdf" library has been deprecated and is no longer maintained. Instead, we have used the PyPDF2 or PyMuPDF library to read and manipulate PDF files in Python.

5.5.2. Extracting tables from PDF

```
#%pip install pdfplumber
import pdfplumber
with pdfplumber.open("D:\\NTCC\\Major Project\\JIEEE Paper Template-OTH.pdf") as f:
    for i in f.pages:
        print(i.extract_tables())
```

Figure 11. Pseudo code for extracting tables from PDF

This code uses the "pdfplumber" library to open and read a PDF file named "JIEEE Paper Template-OTH.pdf". The "pdfplumber.open()" method is used to open the PDF file, and the "with" statement is used to ensure that the PDF file is properly closed

after the file has been processed. The code then loops through all the pages of the PDF using the "pages" attribute of the PDF object. For each page, the "extract_tables()" method is used to extract any tables present on the page. The resulting tables are printed to the console using the "print()" statement. Here, the "extract_tables()" method can only extract tables that are present in the PDF file as separate table objects. If a table is embedded in an image or is part of a larger block of text, it may not be possible to extract the table using this method.

5.5.3. Extracting links from PDF

```
import fitz
doc = fitz.open("D:\\NTCC\\Major Project\\JIEEE Paper Template-OTH.pdf")
#print("Total pages: ", doc.page_count)
for i in range(doc.page_count):
    page = doc.load_page(i)
    pix = page.get_pixmap()
    pix.save("page-%i.png" % page.number)
for i in range(doc.page_count):
    page = doc.load_page(i)
    link = page.get_links()
    print(link)
```

Figure 12. Pseudo code for extracting links from PDF

This code here is using the "fitz" library to open and read a PDF file named "JIEEE Paper Template-OTH.pdf".
The "fitz.open()" method is used to open the PDF file, and the resulting object is assigned to the variable "doc". The code then loops through all the pages of the PDF using the "range()" function and the "doc.page_count" attribute. For each page, the "load_page()" method is used to load the page into memory, and the "get_pixmap()" method is used to convert the page into a pixmap image. The resulting image is saved as a PNG file using the "save()" method. The code then loops through all the pages of the PDF again and uses the "get_links()" method to extract any hyperlinks present on the page. The resulting hyperlinks are printed to the console using the "print()" statement. One thing to note here is this code only extracts hyperlinks that are explicitly defined as such in the PDF file. If a hyperlink is embedded in an image or is part of a larger block of text, it may not be possible to extract the hyperlink using this method.

5.6. Extraction of Transcripts from videos

```
from youtube_transcript_api import YouTubeTranscriptApi
outls = []
tx = YouTubeTranscriptApi.get_transcript('ZIqria9jU9A', languages=['en'])
for i in tx:
    outtxt = (i['text'])
    outls.append(i['text'])
    with open("op.txt", "a") as opf:
        opf.write(outtxt + "\n")

from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(outls)
print("Vocabulary: ", vectorizer.vocabulary_)
```

Figure 13. Pseudo code for extracting transcripts

The code reads the transcript of a YouTube video using the YouTubeTranscriptApi library and stores the text in a list outls.

Then, it uses the CountVectorizer class from the sklearn.feature_extraction.text module to create a document-term matrix of the transcript text. The CountVectorizer class is used to preprocess the text data, by tokenizing the text and converting it into a matrix of word frequencies. The fit() method of the vectorizer object is called on the outls list to learn the vocabulary and create the document-term matrix. Finally, the vocabulary of the document-term matrix is printed. The CountVectorizer class provides many options for preprocessing text data, such as stop word removal, stemming, and n-gram generation. These options can be specified using the constructor arguments or by calling the appropriate methods on the vectorizer object.

5.7. Extraction of text from videos

D: > NTCC > Major Project > yt-dlx.py

```
1  from __future__ import unicode_literals
2  import yt_dlp as youtube_dl
3  ydl_opts = {'format': 'mp4'}
4  ydl = youtube_dl.YoutubeDL(ydl_opts)
5  ydl.download([url_list=['https://www.youtube.com/watch?v=ktBoZ927Uow']])
```

Figure 14. Pseudo code for extracting video link from its URL

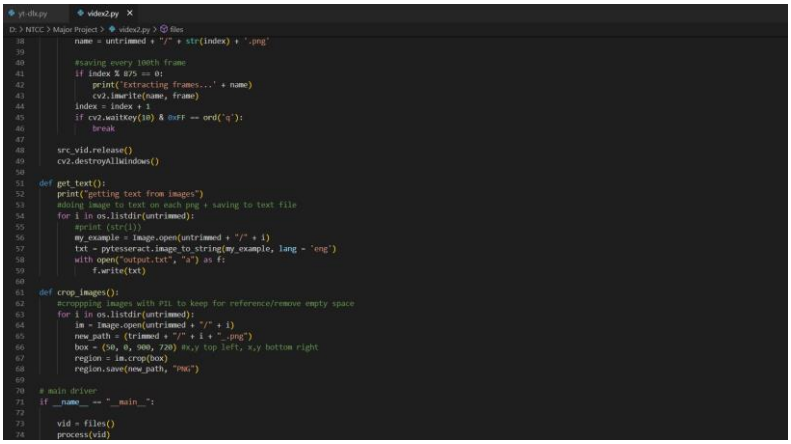
The code here is using the yt-dlp library to download a video from the given YouTube URL in the MP4 format. The options for the yt-dlp downloader are specified in the ydl_opts dictionary, which includes the format key set to "mp4". Then, an instance of the YoutubeDL class is created with the ydl_opts dictionary as an argument. Finally, the download() method of this object is called with a list of URLs to download. In this case, the list contains only one URL, which is the YouTube video with the ID ktBoZ927Uow. When executed, the code will download the video to the current working directory.

yt-dlx.py | videx2.py

D: > NTCC > Major Project > videx2.py > files

```
1  from PIL import Image
2  import pytesseract
3  import os
4  import cv2
5  import shutil
6
7  #path for image files
8  image_frames = 'image_frames'
9  trimmed = 'image_frames/trimmed'
10 untrimmed = 'image_frames/untrimmed'
11
12 op = []
13
14 def files():
15
16     try:
17         shutil.rmtree(image_frames)
18     except OSError:
19             pass
20     if not os.path.exists(image_frames):
21         os.makedirs(image_frames)
22         os.makedirs(trimmed)
23         os.makedirs(untrimmed)
24
25     #specifying the source video path
26     src_vid = cv2.VideoCapture('my_video.mp4')
27     return(src_vid)
28
29 def process(src_vid):
30
31     #using an index to integer-name the files
32     index = 0
33     while src_vid.isOpened():
34         ret, frame = src_vid.read()
35         if not ret:
36             break
37         #naming each frame and saving as png
```

Figure 15(a). Pseudo code for extracting text from videos



```
        name = untrimmed + "/" + str(index) + '.png'

        #saving every 100th frame
        if index % 875 == 0:
            print('Extracting frames...' + name)
            cv2.imwrite(name, frame)
        index = index + 1
        if cv2.waitKey(10) & 0xFF == ord('q'):
            break

    src_vid.release()
    cv2.destroyAllWindows()

def get_text():
    print("getting text from images")
    #doing image to text on each png + saving to text file
    for i in os.listdir(untrimmed):
        #print (str(i))
        my_example = Image.open(untrimmed + "/" + i)
        txt = pytesseract.image_to_string(my_example, lang = 'eng')
        with open("output.txt", "a") as f:
            f.write(txt)

def crop_images():
    #cropping images with PIL to keep for reference/remove empty space
    for i in os.listdir(untrimmed):
        im = Image.open(untrimmed + "/" + i)
        new_path = (trimmed + "/" + i + "_.png")
        box = (50, 0, 900, 720) #x,y top left, x,y bottom right
        region = im.crop(box)
        region.save(new_path, "PNG")

# main driver
if __name__ == "__main__":

    vid = files()
    process(vid)
```

Figure 15(b). Pseudo code for extracting text from videos

The code mentioned above is a script that extracts frames from a video and saves them as PNG images, crops the images to remove empty spaces and saves them in a new folder, and then uses pytesseract to extract text from each image and save it to a text file. Here's a step-by-step breakdown of what the script is doing:

i. Importing necessary modules: The script first imports several Python modules including PIL (Python Imaging Library), pytesseract, os, cv2 (OpenCV), and shutil.

ii. Setting up file paths: The script sets up three file paths: one for the source video (src_vid), one for the extracted PNG frames (image_frames), and two subfolders within the extracted frames folder (trimmed and untrimmed).

iii. Defining functions: The script defines several functions to carry out specific tasks, including:

- files(): This function sets up the image_frames folder and subfolders and returns the source video as a VideoCapture object.
- process(): This function extracts frames from the source video using the VideoCapture object, saves every 875th frame as a PNG in the untrimmed folder, and releases the VideoCapture object.
- get_text(): This function uses pytesseract to extract text from each image in the untrimmed folder and saves it to a text file called output.txt.
- crop_images(): This function opens each image in the untrimmed folder, crops out any empty space using PIL, and saves the cropped image in the trimmed folder with the same name as the original image.

iv. Main driver code: The script calls the files() function to set up the image_frames folder and get the source video, then calls the process() function to extract frames from the video and save them as PNGs, the get_text() function to extract text from the images and save it to a text file, and the crop_images() function to crop the images and save them in the trimmed folder.

The main mechanism behind the proposed OCR Application involves the following steps:

- Pre-processing: The input image is cleaned and enhanced to improve the recognition process. This may involve removing noise, correcting perspective, and adjusting the brightness and contrast of the image.
- Segmentation: The image is divided into smaller segments, such as lines, words, or individual characters. This is done to make the recognition process more manageable.
- Feature Extraction: Features are extracted from each segmented image. These features are unique patterns and characteristics of the characters that help to distinguish one character from another. This is done by computing things like the shape, size, and orientation of the characters.

Classification: The extracted features are then compared to a database of known characters. Based on the similarity between the features of the segmented characters and the reference characters, the system assigns a label to each character.

Post-processing: The recognized characters are then assembled into words and lines, and the final output is produced. This may involve checking for spelling errors, correcting errors, and formatting the output.

6. Conclusion

Optical Character Recognition has been around for many years and has become increasingly important as the amount of digital information has grown. The future of OCR development using Python looks very promising as Python is a popular and widely used programming language for various applications, including OCR. Here are a few areas where OCR development using Python is expected to grow in the future: Improved Accuracy: With advancements in deep learning and computer vision, OCR algorithms will continue to improve their accuracy in recognizing text in images and PDFs, leading to even better performance. Real-Time OCR: As the demand for real-time processing of images and videos increases, OCR systems will need to adapt to real-time processing capabilities. Python's efficient programming and ability to handle real-time data processing makes it a perfect choice for developing real-time OCR systems. Multilingual OCR: As the world becomes more connected and globalized, the demand for OCR systems that can handle multiple languages will continue to grow. Python has strong support for processing multiple languages, and this makes it an ideal platform for multilingual OCR development. Handwriting Recognition: With the increasing use of digital devices for notetaking, the demand for OCR systems that can recognize handwritten text will continue to grow.

Python's ability to integrate with machine learning libraries like TensorFlow and PyTorch makes it a great choice for developing handwriting recognition systems. Integration with Other Technologies: OCR technology will continue to be integrated with other technologies like augmented reality, virtual reality, and the Internet of Things (IoT) to create new and innovative applications. Python's ability to integrate with various technologies and its popularity as a programming language make it an ideal choice for developing these kinds of applications.

The topic of OCR (Optical Character Recognition) technology is one that is fast developing, and numerous ongoing research projects are working to increase the accuracy, speed, and adaptability of OCR. Here are some of the most recent OCR research trends: Deep learning-based OCR: Systems that can accurately recognize characters and words are being developed using deep learning algorithms like CNNs and RNNs. To enhance OCR performance, researchers are experimenting with new deep learning architectures, training methodologies, and data augmentation strategies. Multimodal OCR: To increase OCR accuracy and make input methods more adaptable, multimodal OCR systems combine image recognition with speech recognition or natural language processing. To improve OCR performance, researchers are investigating new multimodal OCR designs, such as attention-based models. OCR systems that can recognize and translate text from a variety of languages are becoming more and more crucial in today's globalized society. Using methods like language modelling, cross-lingual transfer learning, and neural machine translation, researchers are creating multilingual OCR systems. OCR for low-resource languages: OCR systems for low-resource languages encounter a number of difficulties, including a lack of standardization and a lack of training data. Researchers are investigating techniques, such as transfer learning and unsupervised learning, to adapt current OCR systems to low-resource languages. OCR systems for historical documents confront a number of difficulties, including deterioration, noise, and differences in writing styles. The accuracy of OCR on historical documents is being improved by researchers using techniques like picture enhancement, character identification based on context, and crowdsourcing. In general, ongoing OCR research strives to increase OCR speed, accuracy, and adaptability as well as make OCR available for a wider variety of applications and languages. OCR technology is expected to become more crucial as tasks related to digitalization, automation, and data analysis progress.

With this work, we have cultivated the application development project (OCR) utilizing python. We utilized the famous libraries that are used to extract text data from images, docs, website’s URLs etc. We utilized python libraries like: Apache

tika, requests, warnings, pytesseract, PIL, os, io, pypdf, pdfplumber, flask, open-cv, pymupdf, scikit-learn scipy matplotlib, youtube-dl and shutil too. In this paper, we introduced Python as a practical language for instruction and practical programming. We also observed the Python-introduced characteristics, features, and types of programming assistance. In agreement with these qualities, we discovered Python to be a quick, amazing, versatile, basic, open-source language that maintains numerous advancements. Then, various Python projects of different types were bought. The report has similarly examined how a significant section of Python is being used by various associations. According to facts gathered from well-known and reliable journals and locations, the paper has discussed the reasons why Python is the fastest-creating programming language.

Acknowledgments

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of people whose ceaseless cooperation made it possible, whose constant guidance and encouragement crown all efforts with success. I would like to thank Prof. (Dr.) Deepak Arora, Head of Department-CSE & IT, and Amity University for giving me the opportunity to undertake this work. I would also like to thank my faculty guide Dr. Syed Wajahat Abbas Rizvi who is the biggest driving force behind my successful completion of the work. He has always been there to solve any query of mine and guided me in the right direction regarding the work. Without his help and inspiration, I would not have been able to complete the work. Also, I would like to thank my batch mates who guided me, helped me and gave me ideas and motivation at each and every step.

References

[1]. A. L. Reibman and M. Veeraraghawan, “Reliability Modeling: an overview for system design,” IEEE Computer Society, vol.24, no.4, pp.49-57, 1991.

[2]. J. L. Lions, “ARIANE 5 Flight - 501 Failures Report,” 2010.

[3]. M. R. Lyu, “Handbook of Software Reliability Engineering,” IEEE Computer Society Press Los Alamitos California, ISBN: 978-0070394001, pp.1-850, 1996.

[4]. S. R. Dalal, M. R. Lyu and C.L. Mallows, “Software Reliability,” John Wiley & Sons, 2014.

[5]. R. A. Khan, K. Mustafa and S. I. Ahson, “Operation Profile-a key Factor for Reliability Estimation,” University Press Gautam Das and V. P. Gulati (Eds) CIT, vol.204, pp.347-354, 2004.

[6]. E. E. Ogheneovo, “Software Dysfunction: Why Do Software Fail?” Journal of Computer and Communications, vol.2, pp.25-35, 2014.

[7]. S. W. A. Rizvi, V. K. Singh and R. A. Khan, “Revisiting Software Reliability Engineering with Fuzzy Techniques, Proc. of the 3rd IEEE Int. Conf. on Computing for Sustainable Global Development. Published by IEEE Xplore, New Delhi, India, 2016.

[8]. H. B. Yadav and D. K. Yadav, “Early Software Reliability Analysis using Reliability Relevant Software Metrics,” International Journal of System Assurance Engineering and Management, pp.1-12, 2014.

[9]. S. W. A. Rizvi and R. A. Khan, “Maintainability Estimation Model for Object-Oriented Software in Design Phase (MEMOOD), Journal of Computing, vol.2, no.4, pp.26-32, 2010.

[10]. S. W. A. Rizvi and R. A. Khan, “A Critical Review on Software Maintainability Models,” Proceedings of the Conference on Cutting Edge Computer and Electronics Technologies, pp.144-148, 2009.

[11]. H. Pham, “System Software Reliability,” London: Reliability Engineering Series Springer, 2006.

[12]. A. K. Pandey and N. K. Goyal, “Early Software Reliability Prediction,” Springer India, 2013.

[13]. A. L. Goel, “Software Reliability Models: Assumptions Limitations and Applicability,” IEEE Transaction on Software Engineering, vol.11, no.12, pp.1411-1423, 1985.

[14]. H. B. Yadav and D. K. Yadav, “Early Software Reliability Analysis using Reliability Relevant Software Metrics,” International Journal of System Assurance Engineering and Manage, vol.8, no.4, pp.2097-2108, 2014.