CAPTCHA Solver Documentation

Overview

This project implements a custom CAPTCHA solver for the demo page https://2captcha.com/demo/normal using **only open-source libraries** and no third-party CAPTCHA solving services. The solution automates browser interactions to fetch the CAPTCHA image, processes it with OCR to extract the text, and submits the result on the page.

Two implementations are provided:

- **Node.js** version using Puppeteer and Tesseract.js.
- **Python** version using Selenium and EasyOCR.

Approach

1. **Browser Automation**
    - Used Puppeteer (Node.js) / Selenium (Python) to launch a browser, navigate to the CAPTCHA demo page, and locate the CAPTCHA image element.
    - Extracted the CAPTCHA image URL from the page.
2. **Image Capture & Preprocessing**
    - Downloaded the CAPTCHA image.
    - Applied preprocessing to improve OCR accuracy:
        - Resizing the image (Node.js).
        - Converted to grayscale (both).
        - Normalization and thresholding (Node.js).
    - In Python, used PIL to convert the image to grayscale before OCR.

3. **OCR Text Extraction**
    - Node.js: Used Tesseract.js to recognize characters from the preprocessed image, with a whitelist of expected characters (alphanumeric uppercase).
    - Python: Used EasyOCR to perform OCR on the grayscale image.
    - The extracted text is cleaned and trimmed before use.
4. **Form Submission & Validation**
    - Programmatically entered the OCR text into the CAPTCHA input field.
    - Submitted the form.
    - Captured the page response message to determine if the CAPTCHA was solved successfully or if an error occurred.

Challenges Faced

- **OCR Accuracy:** The CAPTCHA image is noisy and distorted to prevent easy recognition. Preprocessing techniques like grayscale conversion and thresholding were essential to improve OCR results.
- **Character Segmentation:** The CAPTCHA contains uppercase alphanumeric characters, so limiting OCR character sets helped reduce misrecognition.
- **Dynamic Page Elements:** Handling the success or error message elements required conditional checks because the page layout changes depending on the result.
- **Timing:** Adding appropriate delays was necessary to allow the page to update after form submission, especially for the Python Selenium script.

Assumptions

- The page elements’ CSS selectors remain consistent.
- OCR libraries work well enough on preprocessed CAPTCHA images for this demo.

Enhancements & Image Preprocessing

- **Node.js version** applies image preprocessing using the Sharp library:
  - Resizing the image to standardize size.
  - Converting to grayscale to reduce color noise.
  - Normalization to adjust contrast.
  - Thresholding to convert the image to black-and-white for clearer character edges.
- These steps significantly improved OCR accuracy compared to raw images.
- **Python version** uses grayscale conversion with PIL, which also helps OCR performance by simplifying image details.

Output

- The extracted CAPTCHA text is printed to the console.
- The script automatically submits the extracted text on the CAPTCHA page.
- The page response is logged, indicating success or failure of the CAPTCHA submission.

How to Run

Node.js

1. Install dependencies:

```
npm install puppeteer tesseract.js sharp axios
```

2. Run the script:

```
node captcha_solver.js
```

Python

1. Install dependencies:

```
pip install selenium easyocr pillow requests numpy
```

2. Download ChromeDriver and ensure it’s in your PATH.
3. Run the script:

```
python captcha_solver.py
```