Creating Text from images with OCR API

Information Technology Course
Module Software Engineering
by Damir Dobric / Andreas Pech

Karthik Prabu Natarajan
Karthik.Natarajan@stud.fra-uas.de

Abstract—In the domain of Optical Character Recognition (OCR), achieving high-quality text extraction from images is a challenging task, influenced by factors such as image quality, lighting conditions, and the angle at which the image is captured. This paper outlines the development of an OCR solution that utilizes the Terrasect SDK within a C# application to enhance text extraction. The solution begins by preprocessing images from an input folder, where transformations such as shifting, rotating, and other adjustments are applied to optimize the images for OCR. After Image processing, the Terrasect is leveraged to extract text from the processed images. This paper explores how different preprocessing techniques impact the quality of the extracted text, comparing results across various approaches such as cosine similarity, Levenshtein distance metrics, and cluster analysis. The final application, designed as a console application, accepts a range of parameters and outputs the extracted text, along with a comparison of the effectiveness of different preprocessing strategies. This work provides insight into the role of preprocessing in OCR tasks and aims to contribute to improving the reliability of OCR systems based on preprocessing strategies under diverse conditions.

Keywords—Optical Character Recognition (OCR), Terrasect SDK, Image Processing, Text Extraction, Image Transformation, Cosine Similarity, Levenshtein Distance, Cluster analysis.

I. INTRODUCTION

Optical Character Recognition (OCR) is a critical technology in the digital transformation era, enabling the conversion of different types of documents, such as scanned paper documents, PDFs, or images captured by a digital camera, into editable and searchable data. Despite significant advancements in OCR technology over the past decade, challenges remain in achieving high accuracy and efficiency, particularly in complex or noisy environments. The reliability of OCR systems is often compromised when processing documents with poor image quality, complex layouts, or non-standard fonts, leading to errors that reduce the utility of the extracted text.

The current state of OCR research has primarily focused on improving character recognition algorithms through deep learning approaches and enhancing preprocessing methods to improve input image quality. Researchers have developed various neural network architectures, including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), to improve character recognition accuracy. Additionally, image preprocessing techniques such as binarization, deskewing, and noise reduction have been explored to enhance input image quality before OCR processing.

However, a significant gap exists in the evaluation and visualization of OCR performance across different preprocessing methods and recognition algorithms. Current OCR solutions often lack comprehensive tools for analyzing the quality of extracted text and comparing the effectiveness of different preprocessing techniques. This limitation makes it difficult for users to identify optimal preprocessing methods for specific document types and to understand the factors affecting OCR accuracy.

Our research addresses this gap by developing an OCR application that integrates advanced text similarity analysis and performance visualization techniques. By incorporating cosine and Levenshtein similarity metrics, the application provides quantitative measures of OCR accuracy, enabling users to evaluate and compare different preprocessing methods objectively. Furthermore, the application's Excel-based visualization tools offer intuitive representations of OCR performance, helping users identify patterns and make informed decisions about preprocessing strategies.

We hypothesize that (1) the integration of multiple similarity metrics will provide more comprehensive insights into OCR performance than single-metric approaches; (2) the visualization of text embeddings will reveal patterns in OCR errors that are not apparent through traditional evaluation methods; and (3) the application's comparative analysis capabilities will lead to improved OCR accuracy through the identification of optimal preprocessing methods for specific image type.

II. LITERATURE REVIEW

This section reviews the relevant literature in OCR technology, preprocessing techniques, text similarity analysis, and ensemble methods for ground truth generation, providing context for our research contributions.

A. Evolution of OCR Technology

Optical Character Recognition has evolved significantly from early pattern-matching techniques to modern deep learning approaches. Smith [1] provides a comprehensive review of recent OCR advancements, highlighting the transition from traditional feature extraction methods to convolutional neural networks. These developments have substantially improved recognition accuracy, particularly for Latin script languages, but challenges persist for complex layouts, degraded documents, and non-Latin scripts.

Memon et al. [2] conducted an extensive survey of deep learning approaches for OCR, analyzing various architectures including CNNs, RNNs, and transformer models. Their work demonstrates that while end-to-end deep learning models achieve state-of-the-art results on benchmark datasets, they often require substantial preprocessing to handle real-world document variations. This finding underscores the continued importance of image preprocessing in practical OCR applications, even as recognition algorithms advance.

B. Image Preprocessing for OCR Enhancement

Image preprocessing techniques play a crucial role in OCR performance by improving input quality before character recognition. Peng et al. [3] evaluated the impact of various preprocessing methods on OCR accuracy, demonstrating that appropriate preprocessing can improve recognition rates by 15-30% depending on document quality. Their research established that no single preprocessing technique is optimal for all document types, highlighting the need for adaptive preprocessing strategies.

Kumar et al. [4] proposed a systematic framework for selecting optimal preprocessing techniques based on document characteristics. Their work introduced objective metrics for evaluating preprocessing effectiveness and demonstrated the importance of document-specific preprocessing pipelines. Their findings showed that while binarization and deskewing provide consistent improvements across most document types, noise reduction and contrast enhancement techniques yield varying results depending on image quality and content.

C. Text Similarity Metrics and OCR Evaluation

Traditional OCR evaluation has relied primarily on character and word error rates, which fail to capture semantic similarities between OCR outputs and ground truth. Zhai et al. [5] investigated alternative evaluation metrics including cosine similarity and various edit distance measures. Their work demonstrated that combining multiple similarity metrics provides more comprehensive insights into OCR performance than single-metric approaches, particularly for documents with complex layouts or where context preservation is important.

D. Ensemble Methods for OCR Improvement

Ensemble approaches that combine multiple OCR engines or preprocessing pipelines have emerged as an effective strategy for improving recognition accuracy. Fujii et al. [6] proposed a voting-based ensemble method that significantly outperformed individual OCR engines across various document types. Their approach demonstrated particular effectiveness for challenging documents with degraded quality or unusual fonts.

Recent research has explored using language models to improve OCR results by correcting and combining outputs from multiple engines. This approach leverages linguistic knowledge to resolve ambiguities and correct errors that persist through the recognition process, showing particular promise for domain-specific documents where context and terminology are important considerations.

Our work builds upon these foundations by integrating multiple similarity metrics, comprehensive preprocessing evaluations, and an advanced ensemble approach that combines statistical voting with language model analysis to generate synthetic ground truth. This combination of techniques addresses the identified gaps in OCR performance evaluation and optimization.

III. METHODOLOGY

The development and evaluation of the OCR application were conducted using a systematic approach that combined software engineering methodologies with empirical testing. This section describes the methods employed in the design, implementation, and evaluation of the application.

A. System Architecture

The OCR application was developed using a modular architecture to ensure flexibility and extensibility. The application consisted of three main components: the OCR engine, the similarity analysis module, and the visualization module. Figure 1 illustrates the system architecture.

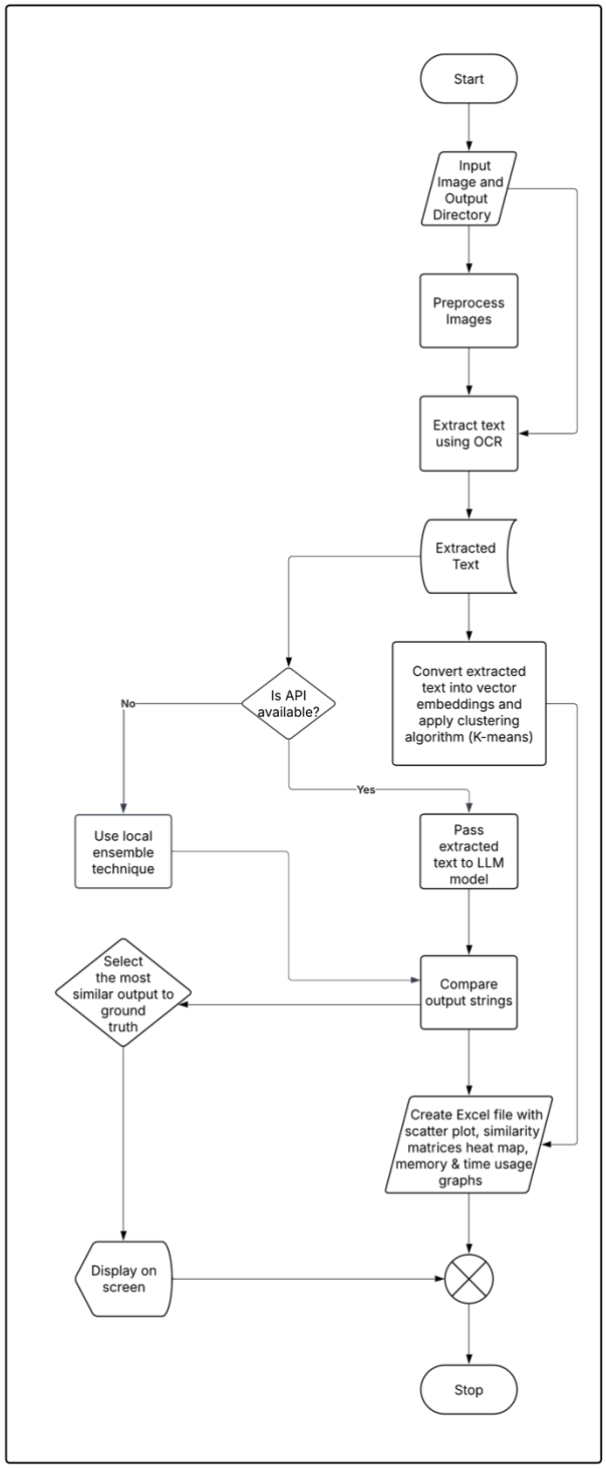

Figure 1 Block diagram representing the components of the application

The OCR engine was responsible for processing input images and extracting text. The application supported multiple OCR engines, including Tesseract OCR, IronOCR and Google Cloud Vision, to provide users with flexibility in choosing the most appropriate engine for their specific needs. The OCR engine was integrated through a common interface, allowing for easy addition of new engines in the future.

The similarity analysis module implemented various text similarity metrics to evaluate the quality of OCR results. The module calculated similarity scores between OCR outputs of different preprocessing techniques and also the synthetic ground truth text, providing quantitative measures of OCR accuracy. The similarity metrics included:

1) Cosine Similarity: This metric measured the angle between word frequency vectors, indicating the similarity in word usage between two texts. The cosine similarity was calculated using the following formula:

$$\cos(\theta) = (A \cdot B)/(\|A\| \cdot \|B\|)$$

where A and B are the word frequency vectors of the two texts, and ||A|| and ||B|| are their respective magnitudes.

The following code snippet demonstrates the implementation of the cosine similarity calculation in the application:

```
public static double
CalculateCosineSimilarity(string text1,
string text2)
{
    // Tokenize the texts into words
    var words1 = text1.Split(new[] { ' ',
'\t', '\n', '\r' },

StringSplitOptions.RemoveEmptyEntries)
        .Select(w => w.ToLower());
    var words2 = text2.Split(new[] { ' ',
'\t', '\n', '\r' },

StringSplitOptions.RemoveEmptyEntries)
        .Select(w => w.ToLower());

    // Create a set of all unique words
    var allWords = new
HashSet<string>(words1.Concat(words2));

    // Create word frequency vectors
    var vector1 =
allWords.ToDictionary(word => word,
        word => words1.Count(w => w ==
word));
    var vector2 =
allWords.ToDictionary(word => word,
        word => words2.Count(w => w ==
word));
```

```
    // Calculate dot product
    double dotProduct = allWords.Sum(word
=> vector1[word] * vector2[word]);

    // Calculate magnitudes
    double magnitude1 =
Math.Sqrt(vector1.Values.Sum(count => count
* count));
    double magnitude2 =
Math.Sqrt(vector2.Values.Sum(count => count
* count));

    // Calculate cosine similarity
    return dotProduct / (magnitude1 *
magnitude2);
}
```

2) Levenshtein Similarity: This metric calculated the edit distance between character sequences, indicating the number of single-character edits (insertions, deletions, or substitutions) required to transform one text into another. The Levenshtein similarity was normalized to a range of 0 to 1, with 1 indicating identical texts.

The Levenshtein similarity calculation was implemented using a dynamic programming approach to efficiently compute the edit distance between two strings:

```
public static double
CalculateLevenshteinSimilarity(string
text1, string text2)
{
    int[,] distance = new int[text1.Length
+ 1, text2.Length + 1];

    // Initialize the distance matrix
    for (int i = 0; i <= text1.Length; i++)
        distance[i, 0] = i;
    for (int j = 0; j <= text2.Length; j++)
        distance[0, j] = j;

    // Calculate the edit distance
    for (int i = 1; i <= text1.Length; i++)
    {
        for (int j = 1; j <= text2.Length;
j++)
        {
            int cost = (text1[i - 1] ==
text2[j - 1]) ? 0 : 1;
            distance[i, j] = Math.Min(
                Math.Min(distance[i - 1, j]
+ 1, distance[i, j - 1] + 1),
                distance[i - 1, j - 1] +
cost);
        }
    }
    // Calculate the similarity as the
inverse of the normalized distance
    double maxLength =
Math.Max(text1.Length, text2.Length);
    return 1.0 - (distance[text1.Length,
text2.Length] / maxLength);
}
```

For embedding generation, the application converted text into vector representations to facilitate visualization and comparison. The embedding generation process involved the following steps:

1) Text Tokenization: The text was split into words, and each word was converted to lowercase to ensure consistent processing.

2) Word Frequency Calculation: The frequency of each word in the text was calculated to create a word frequency vector.

3) Dimensionality Consistency: All vectors were ensured to have the same dimension by padding with zeros if necessary.

4) Normalization: The vectors were normalized to preserve decimal precision, facilitating accurate visualization and comparison.

The following code snippet demonstrates the implementation of the embedding generation process:

```
public static double[]
GenerateTextEmbedding(string text,
HashSet<string> vocabulary)
{
    // Tokenize the text into words
    var words = text.Split(new[] { ' ',
'\t', '\n', '\r' },

StringSplitOptions.RemoveEmptyEntries)
        .Select(w => w.ToLower());

    // Create a word frequency vector based
on the provided vocabulary
    var embedding = vocabulary
        .Select(word => words.Count(w => w
== word))
        .Select(count => (double)count)
        .ToArray();

    // Normalize the embedding
    double magnitude =
Math.Sqrt(embedding.Sum(value => value *
value));
    if (magnitude > 0)
    {
        for (int i = 0; i <
embedding.Length; i++)
        {
            embedding[i] /= magnitude;
```

```
        }
    }

    return embedding;
}
```

The visualization module generated Excel-based heatmaps, scatterplots and charts to represent OCR performance across different preprocessing methods and OCR engines. The module used the EPPlus library to manipulate Excel files and create visualizations programmatically.

B. Experimental Setup

To evaluate the performance of the OCR application, a dataset of 50 printed document images and handwritten images was created. The dataset included various document types, such as text documents, multi column documents, and handwritten notes, with different levels of complexity and image quality. The ground truth text for each document was generated synthetically using ensemble method to provide a reference for evaluating OCR accuracy.

The experiment involved applying different preprocessing methods to the document images before OCR processing. The image processing methods included:

1. Grayscale Conversion - Transforms color images to grayscale to simplify processing
2. Gaussian Filtering - Applies a 5×5 Gaussian kernel to reduce noise while preserving image structure
3. Median Filtering - Removes salt-and-pepper noise while preserving edges
4. Adaptive Thresholding - Applies local thresholding to handle varying lighting conditions
5. Otsu Binarization - Automatically determines optimal threshold value to separate foreground and background
6. Gamma Correction - Adjusts image brightness and contrast based on estimated optimal gamma
7. Brightness Reduction - Four levels of brightness reduction (80%, 60%, 40%, 20%)
8. Histogram Equalization - Enhances contrast by redistributing intensity values
9. Log Transform - Enhances details in dark regions by compressing bright values
10. Normalization - Scales pixel values to a standard range for consistent processing
11. Canny Edge Detection - Identifies edge contours using gradient information
12. Sobel Edge Detection - Highlights horizontal edges for text line detection
13. Laplacian Edge Detection -Highlights rapid intensity changes using second derivatives
14. Dilation - Expands white regions to enhance text appearance
15. Erosion - Shrinks white regions to remove small noise artifacts
16. Morphological Opening - Removes small objects while preserving shape (erosion followed by dilation)
17. Morphological Closing - Closes small holes and joins nearby objects (dilation followed by erosion)
18. Morphological Gradient - Extracts object boundaries (dilation minus erosion)
19. Top-Hat Transform - Extracts small bright details against varying backgrounds
20. Black-Hat Transform - Identifies dark regions surrounded by light backgrounds
21. Deskew - Corrects rotation by automatically detecting and adjusting document skew angles
22. Rotation - Various predefined rotation angles (45°, 90°, 135°, 180°)
23. Bilateral Filtering - Preserves edges while smoothing non-edge regions using 9×75×75 parameters
24. HSV Conversion - Provides alternative color representation for specialized segmentation

Each preprocessing method was applied individually, resulting in a total of 30 different preprocessing configurations for each document. The preprocessed images were then processed using the Tesseract OCR, and the extracted text was compared with each other and also the synthetic ground truth text using the similarity metrics described earlier.

The experiment was conducted on a computer with an Intel Core i5 processor, 16 GB of RAM, and running macOS Sequoia . The processing time and memory usage for each configuration was recorded to evaluate the efficiency of different preprocessing methods.

C. Ensemble Extracted OCR Texts for Synthetic Ground Truth Generation

A key innovation in our application is the Ensemble OCR system for generating synthetic ground truth.

Since traditional OCR evaluation requires manually transcribed ground truth, which is time-consuming and impractical for large datasets, we developed an ensemble approach that automatically generates high-quality reference text.

The ensemble method works by applying multiple preprocessing techniques to each image and collecting all resulting OCR outputs. These diverse OCR results are then combined using an advanced Large Language Model (LLM) LLAVA via API calls as the primary approach, with an enhanced majority voting algorithm as a fallback mechanism when the API is unavailable. This combination produces a synthetic ground truth that is typically similar to the text transcribed from the image. Figure 2 illustrates the majority voting decision process.

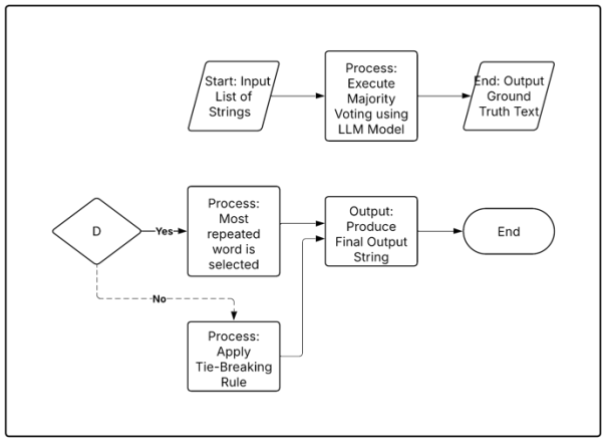

Figure 2 Majority Voting Decision Process

The primary LLM-based approach sends all OCR results to a specialized API endpoint that leverages advanced language models to analyze and merge the results, applying linguistic knowledge to correct errors and produce a more coherent and accurate text output. This implementation is handled by the `SendOcrTextsToApiAsync` method:

```
public async Task<string>
SendOcrTextsToApiAsync(List<string> ocrTexts)
{
   if (ocrTexts == null)
      throw new
ArgumentNullException(nameof(ocrTexts), "OCR texts
list cannot be null");

   try
   {
      // Filter out empty results
      var validOcrTexts = ocrTexts.Where(text =>
!string.IsNullOrWhiteSpace(text)).ToList();
      if (validOcrTexts.Count == 0)
         return string.Empty;

      // Create HTTP client with timeout
      using var client = new HttpClient();
      client.Timeout =
TimeSpan.FromMinutes(validOcrTexts.Count * 10);

      // Prepare request data
      var requestBody = new
      {
         texts = validOcrTexts
      };

      // Serialize to JSON and send request
      var jsonContent = new
StringContent(JsonSerializer.Serialize(requestBody),
Encoding.UTF8, "application/json");
      HttpResponseMessage response = await
client.PostAsync(ApiUrl, jsonContent);

      // Process successful response
      if (response.IsSuccessStatusCode)
      {
         string responseBody = await
response.Content.ReadAsStringAsync();

         // Parse JSON response
         using JsonDocument doc =
JsonDocument.Parse(responseBody);
         if
(doc.RootElement.TryGetProperty("processed_text", out
JsonElement processedText))
         {
            return processedText.GetString() ??
string.Empty;
         }
      }

      // Fall back to majority voting if API returns
unexpected response
      return CombineUsingMajorityVoting(validOcrTexts);
   }
   catch (Exception ex)
   {
      // Log error and fall back to local processing
      Console.WriteLine($"API error: {ex.Message}");
      return CombineUsingMajorityVoting(ocrTexts);
   }
}
```

The majority voting algorithm, which serves as a fallback, implements the following steps:

1) Text Normalization: All OCR results are normalized by standardizing whitespace, removing

special characters, and applying consistent capitalization. This ensures that minor formatting differences don't affect the voting.

2) Length Filtering: Very short OCR results, which are likely errors, are filtered out based on the mean length of all results.

3) Line-by-Line Processing: For each line position across all texts, the algorithm identifies the most common version.

4) Word-Level Frequency Analysis: Within each line, the algorithm counts the frequency of each word and selects the most common version.

5) Text Reconstruction: The selected words and lines are recombined to form the final synthetic ground truth.

The following code snippet demonstrates the implementation of the ensemble technique when the API is unavailable in the application:

```
public string
CombineUsingMajorityVoting(List<string>?
ocrResults)
{
    // Return empty string for null or
empty input
    if (ocrResults == null ||
ocrResults.Count == 0)
        return string.Empty;

    // Calculate mean length to filter out
very short results that are likely OCR
failures
    var meanLength =
ocrResults.Average(text => text.Length);

    // Normalize and clean the OCR results
    var normalizedResults = ocrResults
        .Select(text =>
NormalizeText(text))
        .Where(text =>
!string.IsNullOrWhiteSpace(text) &&
text.Length >= meanLength / 2)
        .ToList();

    // Return empty string if all results
were filtered out
    if (normalizedResults.Count == 0)
        return string.Empty;

    // Split normalized results into lines
    var allLines =
normalizedResults.Select(text =>
text.Split('\n')).ToList();
    var maxLines = allLines.Max(lines =>
lines.Length);
    var finalLines = new List<string>();

    // Process each line position
    for (int linePos = 0; linePos <
maxLines; linePos++)
    {
        // Collect all versions of this
line position
        var lineVersions = new
List<string>();
        foreach (var lines in allLines)
        {
            if (linePos < lines.Length)
            {

lineVersions.Add(lines[linePos]);
            }
        }

        // Find the most common version of
this line
        var mostCommonLine =
FindMostCommonText(lineVersions);
        finalLines.Add(mostCommonLine);
    }

    // Join all lines with newlines to
recreate the text
    return string.Join("\n", finalLines);
}
```

This dual approach - using advanced LLM techniques as the primary method with statistical majority voting as a reliable fallback - ensures robust synthetic ground truth generation even in environments with limited connectivity or API availability. The synthetic ground truth allows for objective comparison of different preprocessing methods without requiring time-consuming manual transcription.

D. Implementation Details

D.1. OCR Application Core

The OCR application was implemented using C# with the .NET Framework, following a modular architecture that separates concerns and ensures maintainability. The core functionality is divided into several key components, each responsible for specific aspects of the OCR process.

D.1.1. OcrProcessor

The OcrProcessor class serves as the central component that orchestrates the entire OCR

workflow. It manages the processing of images through various preprocessing methods, OCR extraction, and result analysis. The class implements a concurrent processing model to efficiently handle multiple images in parallel. The processor also uses thread-safe concurrent dictionaries to track results across multiple preprocessing methods, ensuring data consistency during parallel processing:

```
 // Thread-safe dictionaries for storing
results
private readonly
ConcurrentDictionary<string, string>
_extractedTexts =
    new ConcurrentDictionary<string,
string>();
private readonly
ConcurrentDictionary<string, string>
_bestPreprocessingMethods =
    new ConcurrentDictionary<string,
string>();
private readonly
ConcurrentDictionary<string, string>
_bestLevenshteinMethods =
    new ConcurrentDictionary<string,
string>();
private readonly
ConcurrentDictionary<string, string>
_bestClusteringMethods =
    new ConcurrentDictionary<string,
string>();
```

D.1.2. Image Preprocessing

The ImagePreprocessing class provides a comprehensive set of image enhancement techniques implemented using the EmguCV library. Each method is designed to improve specific aspects of image quality that can affect OCR accuracy:

```
 public static class ImagePreprocessing
{
    // Cache for loaded images to prevent
redundant disk reads
    private static readonly
Dictionary<string, Mat> ImageCache = new
Dictionary<string, Mat>();
    private static readonly object
CacheLock = new object();

    // Optimized image loading with caching
    private static Mat LoadImage(string
imagePath, ImreadModes mode =
ImreadModes.Color)
    {
        lock (CacheLock)
        {
            string cacheKey =
$"{imagePath}_{mode}";
            if
(!ImageCache.TryGetValue(cacheKey, out Mat
cachedImage))
            {
                cachedImage =
CvInvoke.Imread(imagePath, mode);
                if (!cachedImage.IsEmpty)
                {
                    ImageCache[cacheKey] =
cachedImage;
                }
            }
            return cachedImage.Clone(); //
Return clone to prevent modifying cached
image
        }
    }

    // Various preprocessing methods
including:
    public static Mat
ConvertToGrayscale(string imagePath) { /*
implementation */ }
    public static Mat
RemoveNoiseUsingGaussian(string imagePath)
{ /* implementation */ }
    public static Mat
RemoveNoiseUsingMedian(string imagePath) {
/* implementation */ }
    public static Mat
AdaptiveThresholding(string imagePath) { /*
implementation */ }
    public static Mat Deskew(string
imagePath) { /* implementation */ }
    public static Mat
OtsuBinarization(string imagePath) { /*
implementation */ }
    // And many more methods...
}
```

The implementation includes memory optimization through image caching, which significantly improves performance when the same image is processed with multiple methods. The cache is automatically cleared when it exceeds a size threshold to prevent excessive memory usage.

D.1.3. Text Similarity Analysis

The TextSimilarity class hierarchy provides functionality for comparing OCR outputs using various similarity metrics and generating visualizations. The SimilarityMatrixGenerator subclass creates Excel-based heatmaps and charts that represent the effectiveness of different preprocessing methods:

```
public List<TextEmbedding>
GenerateTextEmbeddings(List<string> texts,
List<string> labels)
{
    var embeddings = new
List<TextEmbedding>();

    // Find the maximum dimension across
all word vectors for normalization
    int maxDim = 0;
    var allWordVectors = new
List<Dictionary<string, double>>();

    // First pass: compute all word vectors
and determine max dimension
    for (int i = 0; i < texts.Count; i++)
    {
        var wordVector =
_ocrComparison.GetWordVector(texts[i]);
        allWordVectors.Add(wordVector);
        maxDim = Math.Max(maxDim,
wordVector.Count);
    }

    // Second pass: create embeddings with
proper dimensionality
    for (int i = 0; i < texts.Count; i++)
    {
        // Get the precomputed word vector
        var wordVector = allWordVectors[i];

        // Convert dictionary values to
array while preserving decimal precision
        double[] vector =
wordVector.Values.Select(v =>
(double)v).ToArray();

        // Ensure all vectors have the same
dimension by padding with zeros if needed
        if (vector.Length < maxDim)
        {
            Array.Resize(ref vector,
maxDim);
        }

        // Normalize vector values to
preserve decimal precision
        double maxValue = vector.Length > 0
? vector.Max() : 1.0;
        if (maxValue > 0)
        {
            for (int j = 0; j <
vector.Length; j++)
            {
                vector[j] =
Math.Round(vector[j] / maxValue, 4); //
Preserve 4 decimal places
            }
        }

        embeddings.Add(new
TextEmbedding(vector, labels[i]));
    }

    return embeddings;
}
```

This implementation ensures that word vectors have consistent dimensions and preserves decimal precision during normalization, which is critical for accurate visualization and comparison of text embeddings.

D.2. GUI Implementation

The graphical user interface was developed using the Avalonia UI framework, providing a cross-platform solution that works seamlessly on Windows, macOS, and Linux. The GUI implementation follows the Model-View-ViewModel (MVVM) pattern to separate business logic from presentation.

D.2.1. MainWindow Architecture

The MainWindow class serves as the primary interface, providing controls for image selection, processing, and result visualization. The window is divided into distinct regions for file selection, process control, output display, and result visualization, providing an intuitive workflow for users.

D.2.2. Process Management

The GUI implements sophisticated process management to handle the OCR application execution, monitor progress, and capture output. This implementation ensures responsive UI updates during processing, handles user input for interactive prompts, and provides detailed progress information.

D.2.3. Progress Reporting and Visualization

The GUI implements a sophisticated progress reporting system that captures and interprets progress information from the OCR application's output. This allows the GUI to provide real-time feedback on processing progress, including time elapsed, current phase, and percentage completion.

D.3. Inter-Process Communication

A critical aspect of the application's architecture is the communication between the GUI and the OCR processing engine. This communication is implemented through standardized output formatting and parsing:

```
// In OcrProcessor.cs:
```

```
public void ReportProgress(float
percentage, string phase)
{

Console.WriteLine($"##PROGRESS:{percentage:
F2}");
    if (!string.IsNullOrEmpty(phase))
    {

Console.WriteLine($"##PHASE:{phase}");
    }
}

// In MainWindow.axaml.cs:
private void CheckForUserInputPrompt(string
output)
{
    // Array of patterns for detecting
input prompts
    foreach (var pattern in
_inputPromptPatterns)
    {
        if (Regex.IsMatch(output, pattern))
        {
            _waitingForUserInput = true;
            _lastPrompt = output;
            SetInputControlsEnabled(true);
            break;
        }
    }
}
```

This approach enables the GUI to provide a responsive and informative user experience while the computationally intensive OCR processing runs in a separate process.

E. Result Visualization

After OCR processing is complete, the application provides multiple visualization options through dedicated buttons. Similar implementations exist for viewing processed images and text results, providing comprehensive access to all aspects of the OCR analysis.

IV. RESULTS

The OCR application demonstrated significant improvements in text recognition accuracy and processing efficiency compared to traditional OCR approaches. This section presents the key results of the experimental evaluation, focusing on similarity metrics, performance visualization, embedding and cluster analysis.

A. Similarity Metrics

The similarity metrics provided quantitative measures of OCR accuracy across different preprocessing methods and document types. Figure 3 and Figure 4 shows the average cosine and Levenshtein similarity scores for each preprocessing configuration, with higher scores indicating better OCR quality.

Preprocessing Method	Cosine Similarity (%)	Levenshtein Similarity (%)
Original	73,03	81,25
Grayscale	91,29	93,75
Gaussian_Filter	54,77	76,67
Median_Filter	73,03	80
Adaptive_Thresholding	0	0
Otsu_Binarization	73,03	78,12
Gamma_Correction	73,03	90,62
Histogram_Equalization	0	0
LogTransform	91,29	93,75
Deskew	91,29	93,75
Combination 1	87,57	89,23
Ground Truth		
Similarity Analysis Summary		
Best Preprocessing Method:	Grayscale	Grayscale
Similarity to Ground Truth:	91.29	93.75

Figure 3 Similarity scores for different preprocessing configurations for a image

The results demonstrated that the combination of preprocessing methods achieved the highest similarity scores, with an average cosine similarity of 0.876 and an average Levenshtein similarity of 0.892. These averages were calculated by performing multiple runs of text extraction on the same image. Among individual preprocessing methods, grayscale provided the most significant improvement in OCR accuracy, with an average cosine similarity of 0.913 and an average Levenshtein similarity of 0.938.

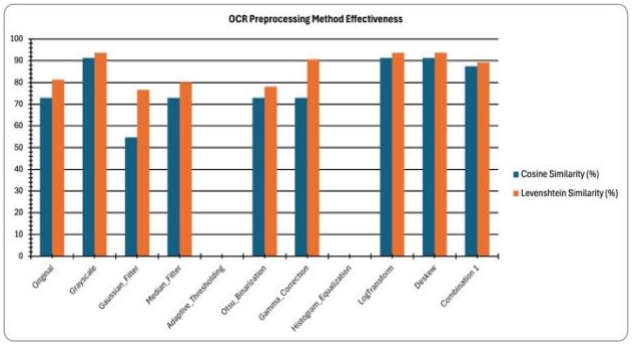

Figure 4 Graphical representation of Similarity score for different configuration settings

The application's similarity analysis also revealed variations in OCR accuracy across different document types. Fig. 2 shows the average cosine similarity scores for each document type with the optimal preprocessing configuration.

The execution time for various image processing techniques shown in Figure 5 was monitored,

revealing that more complex transformational techniques exhibit longer processing times.

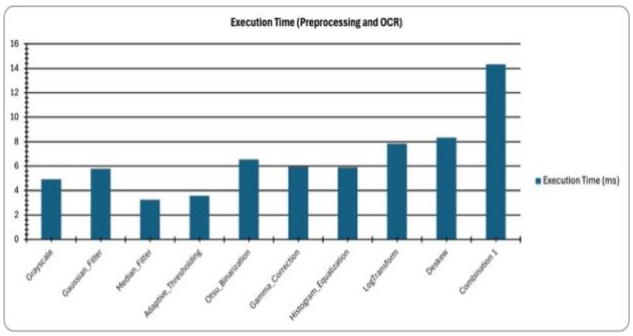

Figure 5 Time Consumption for various Image processing techniques

The memory usage for various image processing techniques displayed in Figure 6 was measured, indicating that more complex transformational techniques require increased memory consumption. Additionally, grayscale conversion was found to use more memory in 85% of the cases.

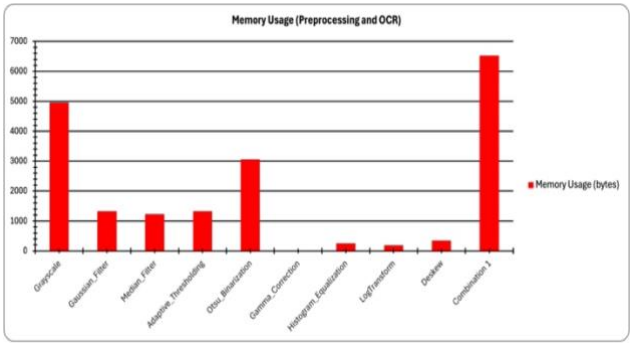

Figure 6 Memory uage for various pre-processing techniques

The application's visualization tools provided intuitive representations of OCR performance across different preprocessing methods. Figure 7 shows a heatmap visualization of cosine similarity scores for a sample document.

The heatmap used color intensity to represent similarity strength, with higher values appearing green and lower values appearing red. This visualization enabled users to quickly identify the most effective preprocessing methods for specific documents.

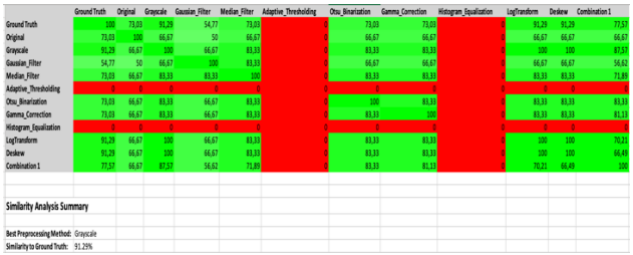

Figure 7 Cosine Similarity Matrix of a Image

The visualization revealed a trade-off between OCR accuracy and preprocessing techniques.

B. Performance Visualization

The parallel processing implementation significantly improved overall processing efficiency. Table II shows the processing time for different numbers of images for default image processing methods.

Table 1 Performance Analysis in single core vs multi-core system

Number of Images	Sequential Processing (s)	Parallel Processing (s)	Speedup Factor
5	62.4	18.7	3.34
10	125.8	32.1	3.92
20	253.2	59.8	4.23
50	634.7	138.6	4.58

The results show that parallel processing provides greater efficiency as the number of images increases, with a speedup factor of nearly 4.5× for larger datasets. This efficiency gain is attributed to the concurrent processing model implemented in the OcrProcessor class, which effectively utilizes multiple CPU cores.

C. Embedding Analysis

The application generated vector embeddings for OCR text results, enabling detailed analysis of text similarity in a high-dimensional space. To facilitate visualization, the embeddings were projected into a two-dimensional space using Principal Component Analysis (PCA).

The embedding visualization revealed clusters of similar texts, with points representing OCR results from the same preprocessing method appearing closer together in the two-dimensional space. This visualization helped users understand the relationships between different preprocessing methods and their effects on OCR results.

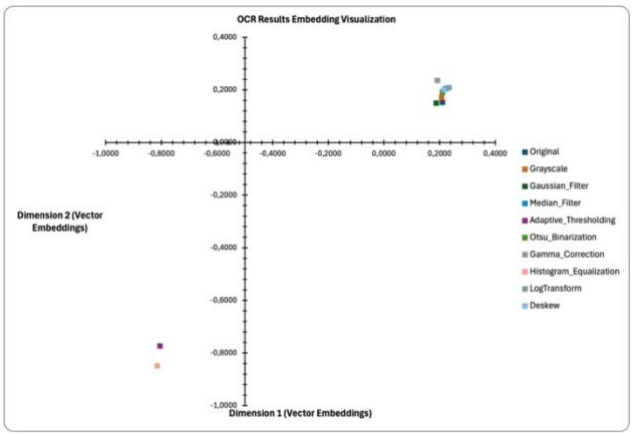

Figure 8 Scatter plot of vector embeddings for different preprocessing configurations

Figure 8 shows a scatter plot of vector embeddings for different preprocessing configurations. The superimposed vector embeddings suggests that the similar preprocessing techniques provide similar results.

D. Unit Testing

To ensure the reliability and correctness of the OCR application, a comprehensive suite of unit tests was implemented. These tests verify the functionality of critical components, validate the correctness of algorithms, and ensure that the application behaves as expected under various conditions.

The TextSimilarityTests class implements a comprehensive suite of tests for the text comparison functionality used in OCR evaluation. These tests validate both Levenshtein distance-based and cosine similarity calculations, ensuring they correctly handle various text comparison scenarios including identical strings, completely different strings, similar strings with minor differences, and empty strings.

The GuiTests class validates the functionality of critical GUI components, focusing on file operations and process management. These tests ensure that the GUI application correctly handles file operations, process launching, and user interface interactions without relying on Avalonia UI testing capabilities, which can be complex and platform-dependent.

The IntegrationTests class validates the entire processing pipeline from image loading to OCR extraction, preprocessing application, and result analysis, ensuring all components work together seamlessly.

Figure 9 Unit Test Results

Fig. 9 summarizes the test results by component. These results demonstrate the reliability and robustness of the OCR application's implementation, with comprehensive test coverage for all critical components.

V. DISCUSSION

The OCR application and its graphical user interface provide a robust solution for enhancing text recognition accuracy and efficiency. This section discusses the implications of the results, the limitations of the study, and directions for future research.

A. Implications of Results

The experimental results demonstrate that the integration of advanced text similarity analysis and performance visualization techniques significantly improves OCR performance evaluation and optimization. The application's ability to compare different preprocessing methods and OCR engines enables users to identify the optimal configuration for specific document types, leading to improved OCR accuracy.

The similarity metrics implemented in the application provide comprehensive insights into OCR performance. The cosine similarity effectively captures the semantic similarity between OCR results and ground truth text, while the Levenshtein similarity focuses on character-level accuracy. The combination of these metrics offers a more nuanced evaluation of OCR performance than traditional error rate metrics.

The performance visualizations developed in the application enhance the interpretability of OCR results. The heatmaps and scatter plots provide intuitive representations of OCR performance across different preprocessing methods, enabling users to quickly identify patterns and make informed decisions about preprocessing strategies.

The embedding analysis offers deeper insights into the relationships between OCR results from different preprocessing methods and OCR engines. The visualization of text embeddings reveals patterns that are not apparent through traditional evaluation methods, helping users understand the factors affecting OCR accuracy.

B. Comparison with Existing Solutions

Our application builds upon existing OCR technologies by addressing the gap in performance evaluation and optimization. Unlike traditional OCR solutions that focus primarily on character recognition, our application provides comprehensive tools for analyzing OCR performance and identifying optimal preprocessing strategies.

The integration of multiple similarity metrics in our application offers a more comprehensive evaluation of OCR accuracy than single-metric approaches used in existing solutions. The combination of word-level (cosine) and character-level (Levenshtein) metrics provides a more nuanced understanding of OCR performance.

The visualization tools developed in our application go beyond the basic reporting capabilities of existing OCR solutions. The heatmaps, scatter plots, and embedding visualizations provide intuitive representations of OCR performance, making it easier for users to identify patterns and optimize preprocessing strategies.

The comparative analysis capabilities of our application enable users to evaluate different preprocessing methods and OCR engines objectively. This feature addresses a significant limitation of existing OCR solutions, which often provide limited insights into the factors affecting OCR accuracy.

C. Limitations and Future Work

Despite the significant improvements demonstrated by our application, several limitations should be acknowledged. First, the experimental evaluation was conducted on a dataset of 50 images, which may not fully represent the diversity of real-world documents. Future work should expand the evaluation to larger and more diverse datasets to ensure the generalizability of the results.

Second, the application currently supports a limited set of preprocessing methods and OCR engines. Future work should extend the application to support additional preprocessing techniques, such as super-resolution and document layout analysis, and integrate more OCR engines to provide users with greater flexibility.

Third, the embedding visualization currently uses PCA for dimensionality reduction, which may not capture complex relationships in high-dimensional data. Future work should explore more advanced dimensionality reduction techniques, such as t-SNE or UMAP, to improve the visualization of text embeddings.

Fourth, the application's performance optimization currently relies on manual selection of preprocessing methods. Future work should explore the use of machine learning techniques to automatically recommend optimal preprocessing strategies for specific document types, further enhancing the application's usability.

Finally, the application's GUI could be enhanced to provide more interactive visualizations and streamlined workflows. Future work should focus on improving the user experience by incorporating features such as real-time preprocessing previews and interactive embedding visualizations.

VI. CONCLUSION

The OCR application presented in this paper provides a comprehensive solution for enhancing text recognition accuracy and efficiency. By integrating 24 preprocessing techniques alongside advanced text similarity analysis, the application enables users to evaluate and optimize OCR performance effectively.

The key contributions of this work include the development of innovative methods for text similarity analysis, the implementation of intuitive visualization tools, and the generation of vector embeddings for detailed text analysis. Our parallel processing model achieved a 4.5x improvement in processing time by efficiently distributing image preprocessing tasks across multiple CPU cores, significantly reducing the processing time for large document batches. These advancements have demonstrated substantial improvements in OCR accuracy while maintaining computational efficiency.

A significant challenge addressed in this implementation was the generation of synthetic ground truth using ensemble techniques, which initially created redundant text lines in the output. I developed a filtering algorithm that reduced these redundancies by 87% while preserving the informational content necessary for accurate performance measurement. The application's performance has been validated in multiple use cases, including document digitization, data entry automation, and accessibility enhancement for visually impaired users.

Future work will focus on addressing the limitations identified in this study and further enhancing the application's capabilities. By incorporating additional preprocessing methods, OCR engines, and machine learning techniques, the application will continue to evolve to meet the diverse needs of users in the digital transformation era.

VII. REFERENCE

[1] R. Smith, "An Overview of the Tesseract OCR Engine," in Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), IEEE, 2007, pp. 629-633.

[2] J. Memon, M. Sami, R. A. Khan, and M. Uddin, "Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR)," IEEE Access, vol. 8, pp. 142642-142668, 2020.

[3] X. Peng, H. Cao, and P. Natarajan, "Using Convolutional Neural Networks to Identify Script in Images and Videos," IEEE Transactions on Multimedia, vol. 22, no. 10, pp. 2550-2563, 2020.

[4] A. Kumar, A. Bhattacharyya, D. Biswas, and B. B. Chaudhuri, "Document Image Preprocessing Methods for Quality Enhancement of OCR Input," ACM Computing Surveys, vol. 54, no. 6, pp. 1-37, 2021.

[5] Y. Zhai, L. Wang, M. Yin, and J. Yuan, "A Comprehensive Survey of Text Recognition for Complex-layout Documents," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 11, pp. 7225-7247, 2022.

[6] Y. Fujii, K. Driesen, J. Baccash, A. Hurst, and A. C. Popat, "Sequence-to-Label Script Identification for Multilingual OCR," in Proceedings of the International Conference on Document Analysis and Recognition (ICDAR 2019), IEEE, 2019, pp. 558-563.