TEXT LINE SEGMENTATION FOR MEDIEVAL MANUSCRIPTS

A project report submitted to

Rajiv Gandhi University of Knowledge Technologies

SRIKAKULAM

In partial fulfilment of the requirements for the Award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

3rd year B. Tech 2nd semester

B. Manasa	**S180122**
I. Praveen Kumar	**S180221**
J. Bhuvana Sree	**S180073**

Under the Esteemed Guidance of

Mr. G. Shiva Rama Sastry, (M. Tech, PhD)

BONAFIED CERTIFICATE

This is to certify that the mini project report titled **"TEXTLINE SEGMENTATION FOR MEDIEVAL MANUSCRIPTS"** was successfully completed by **B. MANASA (S180122), I. PRAVEEN KUMAR (S180221), J. BHUVANA SREE (S180073)** under the guidance of **Mr. G. Shiva Rama Sastry, (M. Tech, PhD)** In partial fulfillment of the requirements for the Mini Project in Computer Science and Engineering of **Rajiv Gandhi University of Knowledge Technologies** under my guidance and output of the work carried out is satisfactory.

Mr. G. Shiva Rama Sastry, (M. Tech, PhD)
Assistant Professor
PROJECT GUIDE

Mr. N. Sesha Kumar, (M.Tech)
Head of the department
PROJECT CO-ORDINATOR

DECLARATION

We declared that this thesis work titled **"TEXT LINE SEGMENTATION FOR MEDIEVAL MANUSCRIPTS"** is carried out by us during the year 2022-2023 in partial fulfillment of the requirements for the Mini Project in **Computer Science and Engineering.** We further declare that this dissertation has not been submitted elsewhere for any Degree. The matter embodied in this dissertation report has not been submitted elsewhere for any other degree. Furthermore, the technical details furnished in various chapters of this thesis are purely relevant to the above project and there is no deviation from the theoretical point of view for design, development and implementation.

With Regards

Bojja Manasa	**S180122**
Ippili Praveen Kumar	**S180221**
Jami Bhuvana Sree	**S180073**

ACKNOWLEDGEMENT

We would like to articulate my profound gratitude and indebtedness to our project guide **Mr. G. Shiva Rama Sastry**, who has always been a constant motivation and guiding factor throughout the project time. It has been a great pleasure for us to get an opportunity to work under his guidance and complete the thesis work successfully.

We wish to extend our sincere thanks to **Mr. N. Shesha Kumar** Head of the Computer Science and Engineering Department, for her constant encouragement throughout the project. We are also grateful to other members of the department without their support our work would have been carried out so successfully.

I thank one and all who have rendered help to me directly or indirectly in the completion of my thesis work.

Project Associate:

Bojja Manasa	**S180122**
Ippili Praveen Kumar	**S180221**
Jami Bhuvana Sree	**S180073**

ABSTRACT

Text line segmentation is an essential step in the digitization of medieval manuscripts. These manuscripts are often written in scripts that vary in size, style, and spacing. This process involves identifying individual lines of text within an image of a manuscript page. It is a challenging task due to noise, degradation, and variations in the manuscript layout and format. In recent years, text line segmentation techniques have advanced, including traditional image processing-based methods. This paper provides an overview of these techniques and their performance on medieval manuscript datasets, highlighting the advantages and limitations of each approach. Additionally, the paper identifies open research challenges and future directions for text line segmentation for medieval manuscripts.

Keywords: Text line segmentation, medieval manuscripts, Computer Vision, Image processing, Seam Carving Algorithm.

TABLE OF CONTENTS

CHAPTER-1
INTRODUCTION

1.1 Introduction

Text Line Segmentation is an essential step in the digitalization of medieval manuscripts which uses computer vision and traditional image-processing techniques. These manuscripts, rich in the historical and cultural significance, contain hand written text arranges in lines, often with decorations and irregular layouts. Traditional segmentation algorithms designed for modern printed text often struggle to handle the complexities variability present in the medieval manuscripts. Therefore the development of specialized and techniques tailored to the specific requirements of medieval manuscripts is necessary. This project is to identify and separate individual lines of text, providing a foundation for subsequent analysis and processing.

1.2 Statement of the problem

The problem at hand is the precise segmentation of text lines in mediaeval manuscripts using a seam carving method. Due to variable line spacing, decorative embellishments, fading or damaged text, and different handwriting styles, mediaeval manuscripts provide special difficulties for text line segmentation. The algorithm must effectively divide text into lines while supporting irregular line spacing, correctly identifying and handling decorative components, and adjusting to variances in handwriting styles frequently seen in these manuscripts.

1.3 Objectives

The major goal of our effort is to use the following steps to split the text lines of mediaeval manuscripts for further analysis.

1) Image loading
2) Preprocessing
3) Generate an energy map
4) Seam carving
5) Binning
6) Polygon manager
7) Text line segmentation

1.4 Goal

The goal of this research is to accurately segment text into individual lines in mediaeval manuscripts using a seam carving algorithm. To handle the special qualities of mediaeval manuscripts, such as inconsistent line spacing, artistic embellishments, faded or damaged text, and a variety of handwriting styles, the chosen algorithm will be customized and fine-tuned. In order to effectively digitize and analyze mediaeval manuscripts for historical study

and preservation purposes, it is important to obtain high accuracy and reliability in text line segmentation.

1.5 Scope

The scope of The Text Line Segmentation for medieval manuscripts includes:

- The seam carving method is better suited to tackle the particular difficulties of mediaeval manuscripts, such as irregular line spacing and changes in handwriting styles.
- Our project uses seam carving algorithm for text line segmentation of mediaeval manuscripts.
- For our project, we selected one dataset of mediaeval manuscripts and processed them digitally.
- Our project does some preprocessing and postprocessing procedures on the input image in order to implement the modified algorithm.
- To show the algorithm's value for historical study and preservation efforts, utilize actual samples of mediaeval manuscripts.

1.6 Applications

Text line segmentation used in various application areas some of those are

1. Document Layout Analysis: Text line segmentation is an essential step in document layout analysis, where the goal is to understand the structure and organization of a document. The segmented text lines can be used to identify paragraphs, headings, captions, and other structural elements within a document.

2. OCR Systems: Optical character recognition (OCR) systems aim to convert scanned or printed documents into machine-readable text. Text line segmentation is a crucial component of OCR systems as it helps isolate individual lines of text, enabling accurate character recognition and text extraction.

3. Text Recognition in Images: The segmented text lines can be used for text recognition in images, such as street signs, vehicle license plates, or product labels. By segmenting the text lines, the system can focus on recognizing and understanding the text content more accurately.

1.7 Limitations

Currently, our project only supports one chosen dataset we need to add additional functionalities for this project to work any other type of Manuscripts.

CHAPTER-2
LITERATURE SURVEY

2.1 Collect Information

Information for the Text Line segmentation for Medieval Manuscripts system was collected from various sources, including research papers, existing manuscript databases, online resources and academic publications.

2.2 Study

TEXT LINE SEGEMENTATION FOR MEDIEVAL MANUSCRIPTS

Image and Text Segmentation pipeline for the paper **"Labeling, Cutting, Grouping: an Efficient Text Line Segmentation Method for Medieval Manuscripts",** published at the 15th IAPR International Conference on Document Analysis and Recognition (ICDAR) in 2019.

2.3 Benefits

- Text line segmentation
- Optical character recognition
- Text recognition in medieval manuscripts

2.4 summary

Text line segmentation for medieval manuscripts involves a series of steps to accurately detect and separate lines of text within the historical documents which are in various languages.

CHAPTER -3
ANALYSIS

3.1 Existing system

Text line segmentation use modern text documents as their dataset which aims to improve the accuracy of OCR systems or assist in document analysis tasks.

3.2 Disadvantages

Text line segmentation for modern text document is not that much complex. But when coming to the historical manuscripts its often challenging task because these papers suffer from degradation, noisy, contain ornaments and decorations, contain varying font sizes and scripts.

3.3 Proposed system

Text line segmentation for medieval manuscripts use images of historical manuscripts which based on seam carving algorithm and focus on preserving and digitizing historical documents.

3.4 Advantages

1. Improved Document Analysis
2. Precise text recognition
3. Efficient Document Processing
4. Research Contribution

3.5 system requirements

3.5.1 Software requirements:

- Google Colab
- Windows 10

3.5.2 Hardware requirements:

- RAM: 4GB above
- Hard disk: 5 GB above

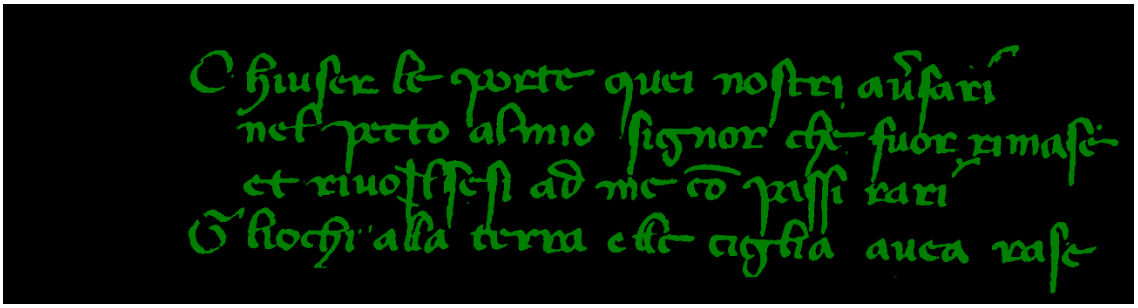
CHAPTER -4

SYSTEM IMPLEMENTATION

4.1 TEXT LINE SEGMENTATION FOR MEDIEVAL MANUSCRIPTS

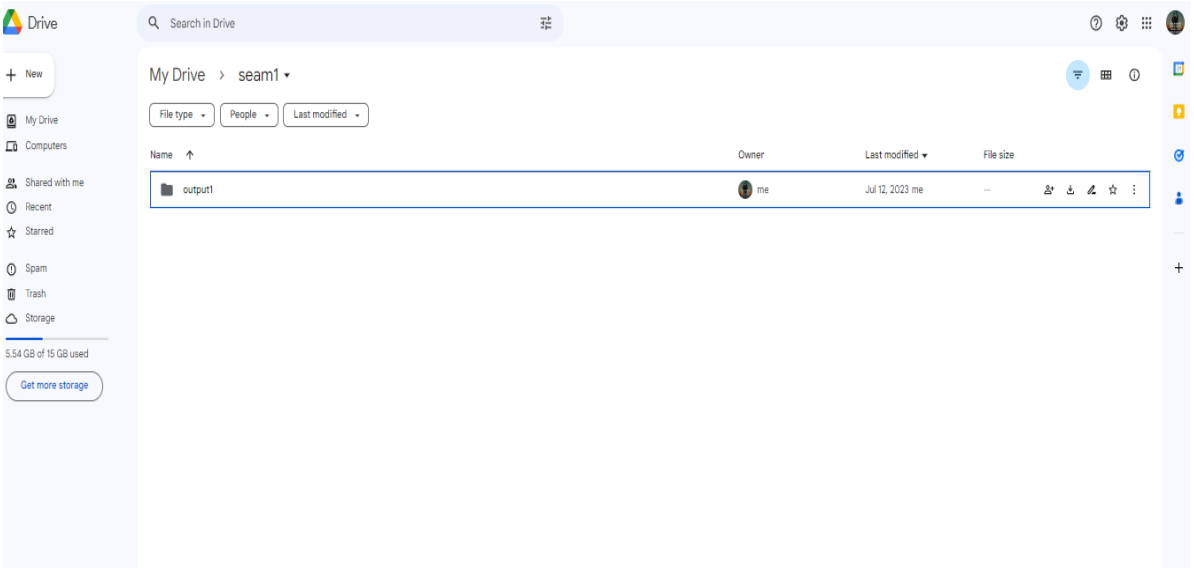
This project is done by using python language based on computer vision and traditional image processing techniques .Mainly we are using Open CV module in this project .In this project we load, preprocess the input image and calculates the energy map of the preprocessed image and then we apply seam carving algorithm for generating seams after that we applied binning and polygon manager for efficient output.

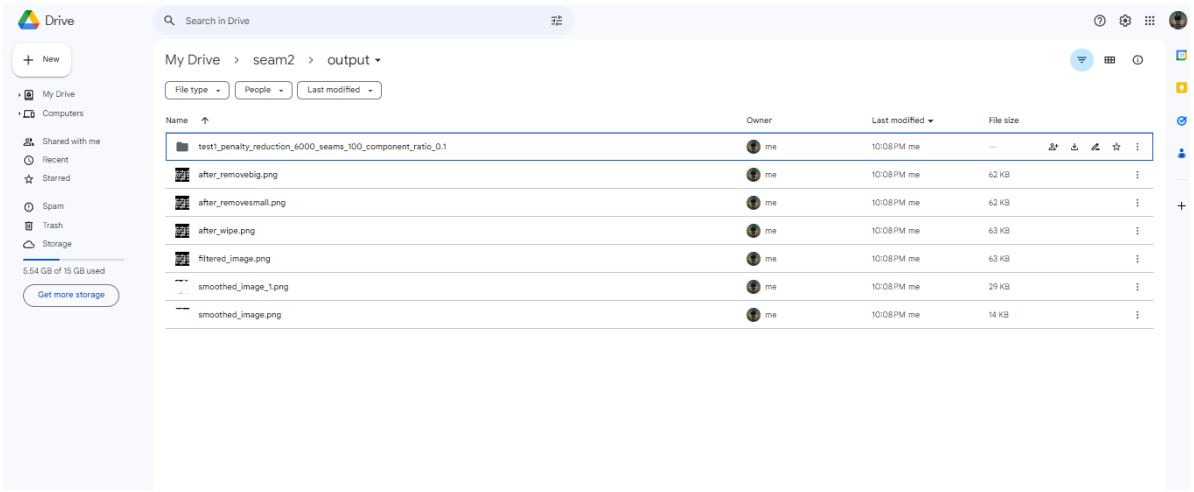
Input Image:

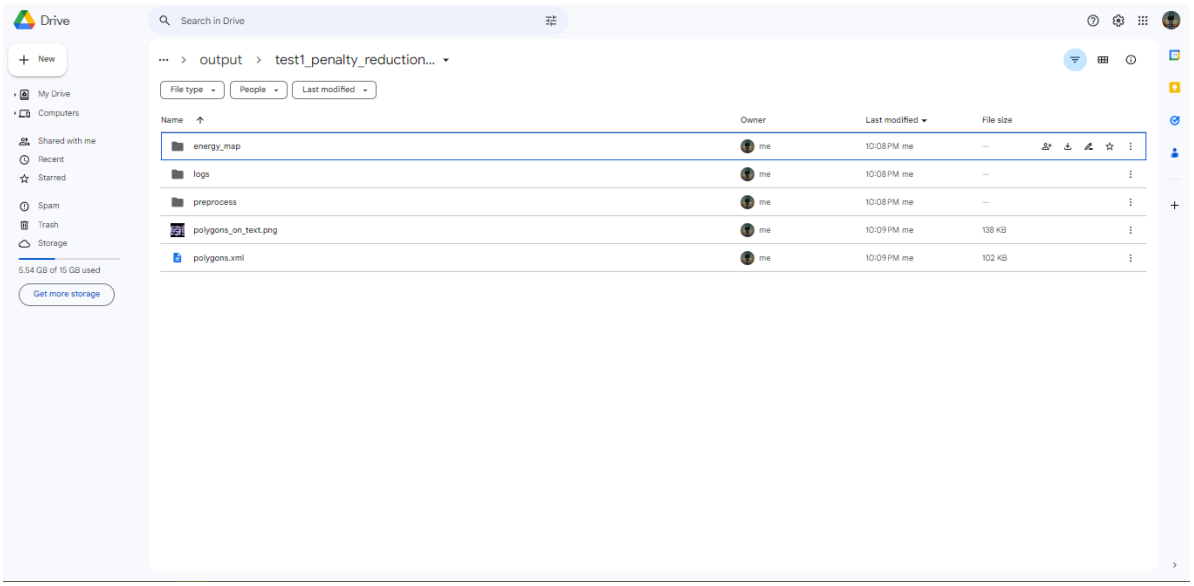


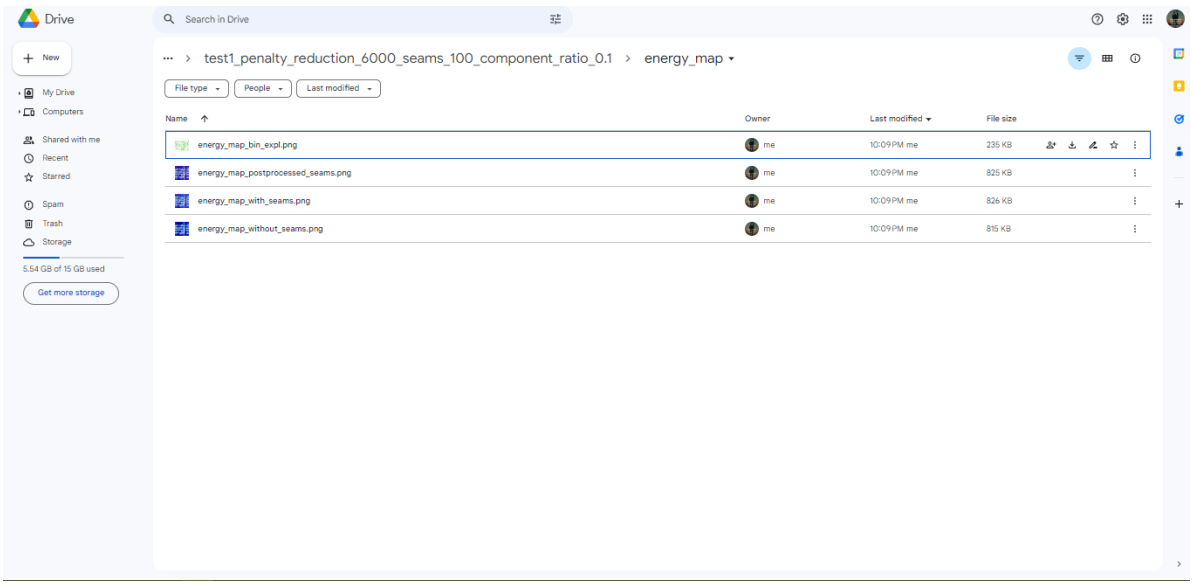
The input image is given to the system it produce output in a folder through the path specified in the code

Output folder:



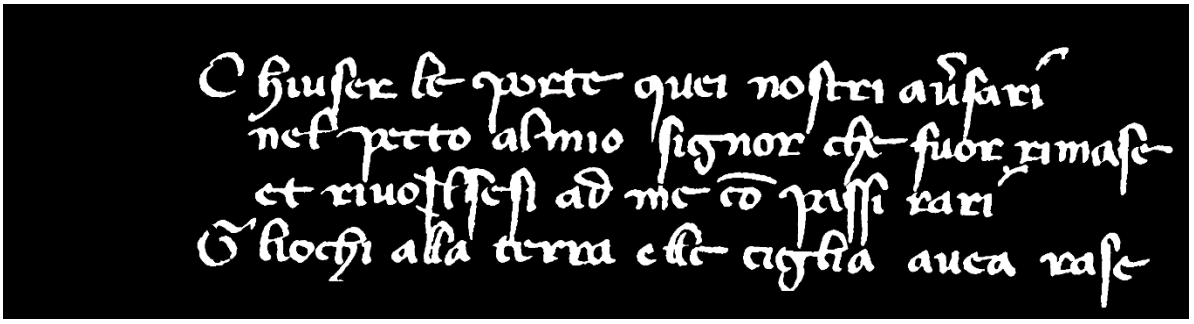

Drive
Search in Drive
New
My Drive
Computers
Shared with me
Recent
Starred
Spam
Trash
Storage
5.54 GB of 15 GB used
Get more storage
My Drive › seam2 › output
File type
People
Last modified
Name
Owner
Last modified
File size
test1_penalty_reduction_6000_seams_100_component_ratio_0.1 me 10:08 PM me
after_removebig.png me 10:08 PM me 62 KB
after_removesmall.png me 10:08 PM me 62 KB
after_wipe.png me 10:08 PM me 63 KB
filtered_image.png me 10:08 PM me 63 KB
smoothed_image_1.png me 10:08 PM me 29 KB
smoothed_image.png me 10:08 PM me 14 KB


Drive
Search in Drive
New
My Drive
Computers
Shared with me
Recent
Starred
Spam
Trash
Storage
5.54 GB of 15 GB used
Get more storage
output › test1_penalty_reduction...
File type
People
Last modified
Name
Owner
Last modified
File size
energy_map me 10:08 PM me
logs me 10:08 PM me
preprocess me 10:08 PM me
polygons_on_text.png me 10:09 PM me 138 KB
polygons.xml me 10:09 PM me 102 KB


Drive
Search in Drive
New
My Drive
Computers
Shared with me
Recent
Starred
Spam
Trash
Storage
5.54 GB of 15 GB used
Get more storage
test1_penalty_reduction_6000_seams_100_component_ratio_0.1 › energy_map
File type
People
Last modified
Name
Owner
Last modified
File size
energy_map_bin_expl.png me 10:09 PM me 235 KB
energy_map_postprocessed_seams.png me 10:09 PM me 825 KB
energy_map_with_seams.png me 10:09 PM me 826 KB
energy_map_without_seams.png me 10:09 PM me 815 KB

Output

After preprocessing

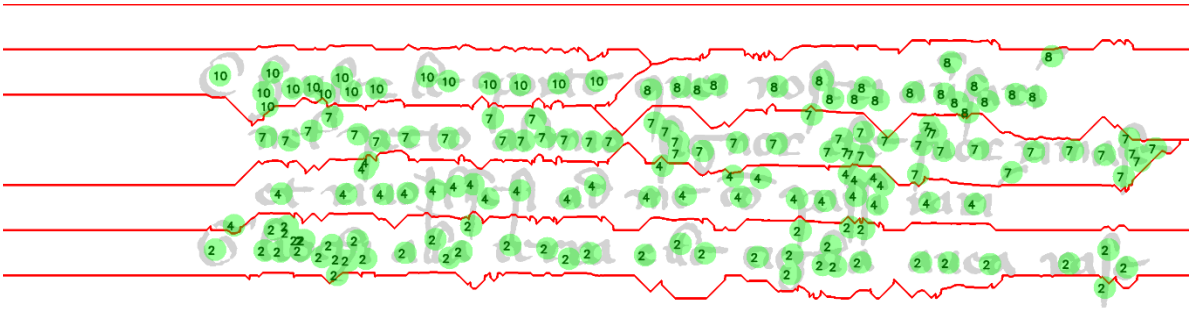


Energy map

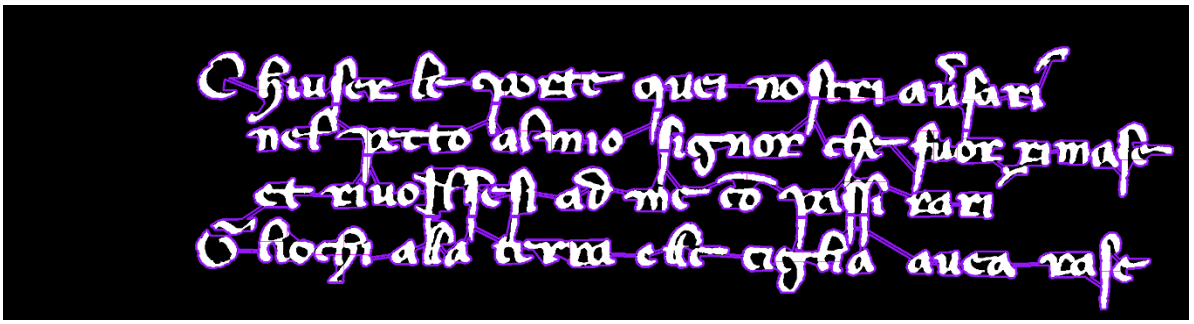
Energy map with seams

Energy map with post preprocessed seams

After binning:



Polygons on text:



CHAPTER-5
SOURCE CODE

5.1 CODE

Drive Mount

```
from google.colab import drive
drive.mount("/content/drive", force_remount=True)
import sys
sys.path.insert(0, '/content/drive/MyDrive/seam1')
%cd /content/drive/MyDrive/seam1/
```

Load Image

```
import errno
import cv2
import logging
import os
import time
import numpy as np
from PIL import Image


def load_image(path):
    if not os.path.exists(path):
        raise FileNotFoundError(errno.ENOENT, os.strerror(errno.ENOENT), path)
    if os.path.splitext(path)[1] == '.tif':
        img = np.asarray(Image.open(path), dtype=int)
        img[np.where(img == 0)] = 8
        img[np.where(img == 1)] = 0
        img = np.stack((img,)*3, axis=-1)
        img[:, :, 1] = 0
        img[:, :, 2] = 0
```

```
    else:
        img = cv2.imread(path)

    if img is None:
        raise Exception("Image is empty or corrupted", path)

    return img


def prepare_image(img, testing, cropping=True, vertical=False):

    start = time.time()
    if testing:
        img[:, :, 0] = 0
        img[:, :, 2] = 0
        locations = np.where(img == 127)
        img[:, :, 1] = 0
        img[locations[0], locations[1]] = 255
        if cropping:
            locs = np.array(np.where(img == 255))[0:2, ]
            img = img[np.min(locs[0, :]):np.max(locs[0, :]), np.min(locs[1, :]):np.max(locs[1, :])]

    else:
        assert len(np.unique(img[:, :, 1])) == 1
        assert np.unique(img[:, :, 1])[0] == 0
        assert len(np.unique(img[:, :, 2])) <= 2
        assert np.unique(img[:, :, 2])[0] == 0
        if len(np.unique(img[:, :, 2])) > 1:
            assert np.unique(img[:, :, 2])[1] == 128
        locations = np.where(img == 128)
        img[locations[0], locations[1]] = 0
        locations_text = np.where(img == 8)
```

```
        locations_text_decoration = np.where(img == 12)
        img[:, :, :] = 0
        img[locations_text[0], locations_text[1]] = 255
        img[locations_text_decoration[0], locations_text_decoration[1]] = 255


    if vertical:
        img = cv2.rotate(img, cv2.ROTATE_90_COUNTERCLOCKWISE)
    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return img
```

Preprocessing Image

```
import logging
import os
import sys
import time
import cv2
from skimage import measure
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

def preprocess(image, small_component_ratio):
  start = time.time()
  image = wipe_outside_textarea(image)
  save_img(image, path=os.path.join('./output', 'after_wipe.png'), show=False)
  image = remove_small_components(image, small_component_ratio)
  save_img(image, path=os.path.join('./output', 'after_removesmall.png'), show=False)
```

```
    image = remove_big_components(image)
    save_img(image, path=os.path.join('./output', 'after_removebig.png'), show=False)
    image[image > 255] = 255


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))

    return image


def wipe_outside_textarea(image):

    ORIGINAL = image


    image = image[:, :, 1]
    filter_size_H = 64
    filter_size_V = 192
    kernel = np.ones((filter_size_V, filter_size_H)) / filter_size_H
    image = cv2.filter2D(image, -1, kernel)
    image[5:-5, int(image.shape[1] / 2) - 5:int(image.shape[1] / 2) + 5] = 255
    save_img(image, path=os.path.join('./output', 'smoothed_image_1.png'), show=False)
    tmp = np.ones((image.shape[0], image.shape[1], 3), dtype=np.uint8)//
    cc = measure.find_contours(image, 200, fully_connected='high')[0]
    cc[:, 0], cc[:, 1] = cc[:, 1], cc[:, 0].copy()
    cc = [cc.astype(np.int32, copy=False)]
    cv2.fillPoly(tmp, cc, (255, 255, 255))
    save_img(tmp, path=os.path.join('./output', 'smoothed_image.png'), show=False)
    tmp = tmp - ORIGINAL
    image = np.stack((image,) * 3, axis=-1)
    image[np.where(tmp != 0)] = 0
```

```
    save_img(image, path=os.path.join('./output', 'filtered_image.png'), show=False)



    image[:, 0:left] = 0
    image[:, right:] = 0

    plt.figure()
    plt.plot(ver)
    plt.axhline(y=np.mean(ver), color='r', linestyle='-')
    plt.axvline(x=left, color='r', linestyle='-')
    plt.axvline(x=right, color='r', linestyle='-')
    plt.savefig('./output/ver.png')



    hor = np.sum(SMOOTH_IMAGE, axis=1)
    hor_indexes = np.where(hor > np.mean(hor))
    top = np.min(hor_indexes)
    bottom = np.max(hor_indexes)
    image[0:top, :] = 0
    image[bottom:, :] = 0

    plt.figure()
    plt.plot(hor)
    plt.axhline(y=np.mean(hor), color='r', linestyle='-')
    plt.axvline(x=top, color='r', linestyle='-')
    plt.axvline(x=bottom, color='r', linestyle='-')
    plt.savefig('./output/hor.png')
    return image

def remove_small_components(image, small_component_ratio):
```

```
    cc_properties = measure.regionprops(measure.label(image[:, :, 1], background=0),
    cache=True)
    avg_area = np.mean([item.area for item in cc_properties])
    for cc in cc_properties:
        if cc.area < small_component_ratio * avg_area:
          image[(cc.coords[:, 0], cc.coords[:, 1])] = 0
     return image


def remove_big_components(image):
    cc_properties = measure.regionprops(measure.label(image[:, :, 1], background=0),
   cache=True)
    avg_area = np.mean([item.area for item in cc_properties])
    for cc in cc_properties:
        if cc.area > 10 * avg_area:
            image[(cc.coords[:, 0], cc.coords[:, 1])] = 0
    return image




import logging
import time
import cv2
import numpy as np


def blow_up_image(image, seams):
    start = time.time()
    new_image = []
    ori_height, _, _ = image.shape
    height = ori_height + len(seams)
```

```
    seams = np.array(seams)

    for i in range(0, image.shape[1]):
        col = np.copy(image[:, i])
        y_cords_seams = seams[:, i, 1]

        seam_nb = 0
        for y_seam in y_cords_seams:
            col = np.insert(col, y_seam + seam_nb, [0, 0, 0], axis=0)
            seam_nb += 1

        new_image.append(col)


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return np.swapaxes(np.asarray(new_image), 0, 1), (((100 / ori_height) * height) - 100) / 100


def blur_image(img, save_name="blur_image.png", save=False, show=False, filter_size=1000,
horizontal=True):
    kernel_motion_blur = np.zeros((filter_size, filter_size))
    if horizontal:
        kernel_motion_blur[int((filter_size - 1) / 2), :] = np.ones(filter_size)
        kernel_motion_blur[:, int((filter_size - 1) / 2)] = np.ones(filter_size)
    kernel_motion_blur = kernel_motion_blur / filter_size
    output = cv2.filter2D(img, -1, kernel_motion_blur)

    if save:
        cv2.imwrite(save_name, output)
```

```
    if show:
        cv2.imshow('image', output)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    return output
```

Energy map

```
import logging
import os
import sys
import time
import cv2
import numpy as np
from scipy.spatial import distance
from skimage import measure

def create_heat_map_visualization(ori_energy_map):
    start = time.time()
    heatmap = ((np.copy(ori_energy_map) / np.max(ori_energy_map)))
    heatmap = (np.stack((heatmap,) * 3, axis=-1)) * 255
    heatmap = np.array(heatmap, dtype=np.uint8)
    heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return heatmap
```

```
def prepare_energy(ori_map, left_column, right_column, y):
    y_value_left, y_value_right = left_column[y], right_column[y]
    ori_map[:, 0] = sys.maxsize / 2
    ori_map[:, -1] = sys.maxsize / 2
    ori_map[y][0], ori_map[y][-1] = y_value_left, y_value_right

    return ori_map

def create_distance_matrix(img_shape, centroids, asymmetric=False, side_length=1000):
    start = time.time()
    template = np.zeros((side_length, side_length))
    center_template = np.array([[np.ceil(side_length / 2), np.ceil(side_length / 2)]])
    pixel_coordinates = np.asarray([[x, y] for x in range(template.shape[0]) for y in
range(template.shape[1])])

    if asymmetric:
        template = np.array([calculate_asymmetric_distance(center_template, pxl, 1, 10)
 for pxl in pixel_coordinates]) \
            .flatten().reshape((side_length, side_length))
    else:
        template = distance.cdist(center_template, pixel_coordinates).flatten().reshape(
(side_length, side_length))

    distance_matrix = np.ones(img_shape) * np.max(template)
    centroids = np.rint(centroids).astype(int)

    for centroid in centroids:
```

```
        pos_v, pos_h = (centroid - np.ceil(side_length / 2)).astype(int)  # offset
        v_range1 = slice(max(0, pos_v), max(min(pos_v + template.shape[0],
distance_matrix.shape[0]), 0))
        h_range1 = slice(max(0, pos_h), max(min(pos_h + template.shape[1],
distance_matrix.shape[1]), 0))


        v_range2 = slice(max(0, -pos_v), min(-pos_v + distance_matrix.shape[0],
         template.shape[0]))
        h_range2 = slice(max(0, -pos_h), min(-pos_h + distance_matrix.shape[1],
       template.shape[1]))
        distance_matrix[v_range1, h_range1] = np.minimum(template[v_range2,
h_range2],distance_matrix[v_range1, h_range1])



    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))
     return distance_matrix.flatten()


def create_energy_map(img, blurring=True, projection=True, asymmetric=False):


    start = time.time()
    cc, centroids, areas = find_cc_centroids_areas(img)
    centroids = np.asarray([[point[0], point[1]] for point in centroids])
    areas = (areas - np.min(areas)) / (np.max(areas) - np.min(areas))
    areas = areas - np.mean(areas)
    areas = - np.abs(areas)
    areas *= 500
    distance_matrix = create_distance_matrix(img.shape[0:2], centroids,
    asymmetric=asymmetric)
    distance_matrix /= 30
    distance_matrix += 1
```

```
    energy_background = ((np.ones(img.shape[0] * img.shape[1]) * 100) /
distance_matrix).transpose()
    locs = np.array(np.where(img[:, :, 0].reshape(-1) == 0))[0:2, :]
    energy_text = energy_background
    energy_text[locs] = 0
    energy_map = energy_background + energy_text
    energy_map = energy_map.reshape(img.shape[0:2])

    if blurring:
        blurred_energy_map = blur_image(img=energy_map, filter_size=300)
        energy_map = blurred_energy_map

    if projection:
        projection_profile = create_projection_profile(energy_map)
        projection_profile *= np.max(energy_map) / 2
        projection_matrix = np.zeros(img.shape[0:2])
        projection_matrix = (projection_matrix.transpose() + projection_profile).transpose()
        projection_matrix = blur_image(projection_matrix, filter_size=1000)
        energy_map = energy_map + projection_matrix

    if True:

        filter_size_H = img.shape[0]
        filter_size_V = img.shape[1]
        kernel = np.zeros((filter_size_V, filter_size_H))
        kernel[int(filter_size_V/2), :] = 1
        kernel[:, int(filter_size_H/2)] = 1
        smoothed = cv2.filter2D(energy_map, -1, kernel)
        filter_size_H = 32
        filter_size_V = 32
        kernel = np.ones((filter_size_V, filter_size_H)) / (filter_size_V*filter_size_H)
        smoothed = cv2.filter2D(smoothed, -1, kernel)
```

```
        smoothed -= np.mean(smoothed)
        smoothed[smoothed < 0] = 0
        smoothed = ((smoothed - np.min(smoothed)) * np.max(energy_map)) /
(np.max(smoothed) - np.min(smoothed))
        energy_map = energy_map + smoothed


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return energy_map, cc


def create_projection_profile(energy_map):

    pp = np.sum(energy_map, axis=1)
    WINDOW_SIZE = 100
    pp = smooth(pp, WINDOW_SIZE)[int(WINDOW_SIZE/2):-int(WINDOW_SIZE/2-1)]
    pp -= np.mean(pp)
    pp[pp < 0] = 0
    pp = (pp - np.min(pp)) / (np.max(pp) - np.min(pp))

    return pp


def smooth(x, window_len=11, window='hanning'):
    if x.ndim != 1:
        raise ValueError("smooth only accepts 1 dimension arrays.")

    if x.size < window_len:
```

```
        raise ValueError("Input vector needs to be bigger than window size.")

    if window_len < 3:
        return x

    if not window in ['flat', 'hanning', 'hamming', 'bartlett', 'blackman']:
        raise ValueError("Window is on of 'flat', 'hanning', 'hamming', 'bartlett', 'blackman'")

    s = np.r_[x[window_len - 1:0:-1], x, x[-2:-window_len - 1:-1]]

    if window == 'flat':
        w = np.ones(window_len, 'd')
    else:
        w = eval('np.' + window + '(window_len)')

    y = np.convolve(w / w.sum(), s, mode='valid')
    return y


def find_cc_centroids_areas(img):

    start = time.time()
    cc_labels, cc_properties = get_connected_components(img)
    amount_of_properties = 0
    avg_area = np.mean([item.area for item in cc_properties])
    std_area = np.std([item.area for item in cc_properties])
    avg_height = np.mean([item.bbox[2] - item.bbox[0] for item in cc_properties])
    avg_width = np.mean([item.bbox[3] - item.bbox[1] for item in cc_properties])

    while amount_of_properties != len(cc_properties):
        amount_of_properties = len(cc_properties)
        image = img[:, :, 1]
```

```
    coef = 1.5
    for item in cc_properties:
        if item.area > coef * avg_area \
                or item.bbox[2] - item.bbox[0] > coef * avg_height \
                or item.bbox[3] - item.bbox[1] > coef * avg_width:
            v_size = abs(item.bbox[0] - item.bbox[2])
            h_size = abs(item.bbox[1] - item.bbox[3])
            y1, x1, y2, x2 = item.bbox

            if float(h_size) / v_size > 1.5:
                image[y1:y2, np.round((x1 + x2) / 2).astype(int)] = 0
            elif float(v_size) / h_size > 1.5:
                image[np.round((y1 + y2) / 2).astype(int), x1:x2] = 0
            else:

                image[y1:y2, np.round((x1 + x2) / 2).astype(int)] = 0
                image[np.round((y1 + y2) / 2).astype(int), x1:x2] = 0

    img[:, :, 1] = image


    cc_labels, cc_properties = get_connected_components(img)


all_centroids = np.asarray([cc.centroid[0:2] for cc in cc_properties])


all_areas = np.asarray([cc.area for cc in cc_properties])


no_outliers = detect_outliers(all_areas, avg_area, std_area)
centroids = all_centroids[no_outliers, :]
```

```
    filtered_area = all_areas[no_outliers]
    all_areas = filtered_area[np.argsort(centroids[:, 0])]
    all_centroids = centroids[np.argsort(centroids[:, 0]), :]


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return (cc_labels, cc_properties), all_centroids, all_areas



def get_connected_components(img):
    cc_labels = measure.label(img[:, :, 1], background=0)
    cc_properties = measure.regionprops(cc_labels, cache=True)
    return cc_labels, cc_properties


def detect_outliers(area, mean, std):

    start = time.time()

    if mean is not None:
        mean = np.mean(area)
    if std is not None:
        std = np.std(area)


    no_outliers = area - 0.25*mean > 0
```

```
    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))



    return no_outliers
```

Seam Carving

```
import itertools
import sys
import cv2
import numba
import numpy as np

@numba.jit()
def horizontal_seam(energies, penalty_reduction, bidirectional=False):
    height, width = energies.shape[:2]
    ori_y = 0
    previous = 0
     seam_forward = []
     seam_backward = []



   for i in range(0, width, 1):
      col = energies[:, i]
      if i == 0:
         ori_y = previous = np.argmin(col)
      else:
         top = col[previous - 1] if previous - 1 >= 0 else sys.maxsize
         middle = col[previous]
         bottom = col[previous + 1] if previous + 1 < height else sys.maxsize
```

```
        if penalty_reduction > 0:
            top += ((ori_y - (previous - 1)) ** 2) / penalty_reduction
            middle += ((ori_y - previous) ** 2) / penalty_reduction
            bottom += + ((ori_y - (previous + 1)) ** 2) / penalty_reduction

        previous = previous + np.argmin([top, middle, bottom]) - 1

    seam_forward.append([i, previous])


if  bidirectional:
    for i in range(width-1, -1, -1):
        col = energies[:, i]
        if i == width-1:
            ori_y = previous = np.argmin(col)
        else:
            top = col[previous - 1] if previous - 1 >= 0 else sys.maxsize
            middle = col[previous]
            bottom = col[previous + 1] if previous + 1 < height else sys.maxsize

            if penalty_reduction > 0:
                top += ((ori_y - (previous - 1)) ** 2) / penalty_reduction
                middle += ((ori_y - previous) ** 2) / penalty_reduction
                bottom += + ((ori_y - (previous + 1)) ** 2) / penalty_reduction

            previous = previous + np.argmin([top, middle, bottom]) - 1

        seam_backward.append([i, previous])

return [seam_forward, seam_backward[::-1]]
```

```
def draw_seams(img, seams, bidirectional=True):

    x_axis = np.expand_dims(np.array(range(0, len(seams[0]))), -1)
    seams = [np.concatenate((x, np.expand_dims(seam, -1)), axis=1) for seam, x in zip(seams, itertools.repeat(x_axis))]

    for i, seam in enumerate(seams):

        if bidirectional and i % 2 == 0:
            cv2.polylines(img, np.int32([seam]), False, (0, 0, 0), 3)
        else:
            cv2.polylines(img, np.int32([seam]), False, (255, 255, 255), 3)


def draw_seams_red(img, seams, bidirectional=True):

    x_axis = np.expand_dims(np.array(range(0, len(seams[0]))), -1)
    seams = [np.concatenate((x, np.expand_dims(seam, -1)), axis=1) for seam, x in zip(seams, itertools.repeat(x_axis))]
    for i, seam in enumerate(seams):
      cv2.polylines(img, np.int32([seam]), False, (0, 0, 255), 3)


def get_seams(ori_energy_map, penalty_reduction, seam_every_x_pxl):

    seams = []

    left_column_energy_map = np.copy(ori_energy_map[:, 0])

    right_column_energy_map = np.copy(ori_energy_map[:, -1])
```

```
    for seam_at in range(0, ori_energy_map.shape[0], seam_every_x_pxl):
        energy_map = prepare_energy(ori_energy_map,left_column_energy_map,
right_column_energy_map, seam_at)

        seams.extend(horizontal_seam(energy_map, penalty_reduction=penalty_reduction,
bidirectional=True))

    seams = np.array([np.array(s)[:, 1] for s in seams])

    return seams


def post_process_seams(energy_map, seams):

    assert energy_map.shape[1] == len(seams[0])

    SAFETY_STOP = 100
    iteration = 0
    repeat = True
    while repeat:

        iteration += 1
        if iteration >= SAFETY_STOP:
            break

        repeat = False
        for index, seam_A in enumerate(seams):
            for seam_B in seams[index:]:

                overlap = seam_A - seam_B
```

```
    overlap[abs(overlap) < 10] = 0


    seam_A[overlap == 0] = seam_B[overlap == 0]


    sequences = non_zero_runs(overlap)

    if len(sequences) > 0:
        for i, sequence in enumerate(sequences):

            target = sequence[1] - sequence[0]

            left = sequence[0] - sequences[i - 1, 1] if i > 0 else sequence[0]
            right = sequences[i + 1, 0] - sequence[1] if i < len(sequences)-1
else
energy_map.shape[1] - sequence[1]

            if target > left and target > right:
                continue

            repeat = True


            sequence = range(*sequence)

            energy_A = measure_energy(energy_map, seam_A, sequence)
            energy_B = measure_energy(energy_map, seam_B, sequence)


            if energy_A > energy_B:
```

```
                seam_A[sequence] = seam_B[sequence]
            else:
                seam_B[sequence] = seam_A[sequence]

    return seams


def non_zero_runs(a):

    iszero = np.concatenate(([1], np.equal(a, 0).view(np.int8), [1]))
    absdiff = np.abs(np.diff(iszero))
    ranges = np.where(absdiff == 1)[0].reshape(-1, 2)
    return ranges


def measure_energy(energy_map, seam, sequence):
return energy_map[seam[sequence], sequence].sum()
```

Binning algorithm

```
import itertools
import logging
import os
import time
import cv2
import numpy as np


def majority_voting(connected_components, seams):
     start = time.time()
    centroids = np.asarray([cc.centroid[0:2] for cc in connected_components[1]])
    centroids = centroids[np.argsort(centroids[:, 0]), :]
```

```
    values = count_seams_below(centroids, seams)
    small_bins = [42]
    while len(small_bins) > 0:
        bin_index, bin_size, unique_bins = split_into_bins_and_index(values)


        if small_bins[0] == 42:
            avg = np.mean(bin_size[bin_size>1])*0.25


        small_bins = unique_bins[np.where(bin_size < avg)
        merge_small_bins(bin_index, centroids, small_bins, values)


    lines = []
    for bin in unique_bins:
        lines.append(list(centroids[np.where(bin_index == bin)]))


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))

    return lines, centroids, values


def count_seams_below(centroids, seams):
    values = np.zeros([len(centroids)])
    for i, centroid in enumerate(centroids):
        cx = int(centroid[1])
        cy = int(centroid[0])
```

```
        for seam in seams:

            if seam[cx] > cy:
                values[i] = values[i] + 1
    return values


def merge_small_bins(bin_index, centroids, small_bins, values):
    for bin in small_bins:

        for loc in np.where(bin_index == bin)[0]:

            loc_p = loc + 1 if loc + 1 < len(values) else loc
            while bin_index[loc_p] == bin_index[loc]:
                if loc_p + 1 < len(values):
                    loc_p += 1
                else:
                    break


            loc_m = loc - 1 if loc > 0 else loc
            while bin_index[loc_m] == bin_index[loc]:
                if loc_m > 0:
                    loc_m -= 1
                else:
                    break


            XA = np.expand_dims(centroids[loc], axis=0)

            upper = np.array([calculate_asymmetric_distance(XA, c, 1, 5) for c in
                              centroids[np.where(bin_index == bin_index[loc_p])]]).min()
```

```
        lower = np.array([calculate_asymmetric_distance(XA, c, 1, 5) for c in
                          centroids[np.where(bin_index == bin_index[loc_m])]]).min()

        values[loc] = values[loc_m] if (upper == 0 or upper > lower) and lower != 0 else
values[loc_p]



def split_into_bins_and_index(values):

    bin_index = np.digitize(values, np.unique(values))

    unique_bins, bin_size = np.unique(bin_index, return_counts=True)
    return bin_index, bin_size, unique_bins



def compute_avg_pairwise_distance(centroids):

    centroids = centroids[np.argsort(centroids[:, 1]), :]
    dist = []
    for c1, c2 in pairwise(centroids):

        dist.append(c2[1] - c1[1])
    return dist



def check_for_anomaly(centroids, threshold):

    centroids = centroids[np.argsort(centroids[:, 1]), :]
    for c1, c2 in pairwise(centroids):
        if c2[1] - c1[1] > threshold:
            return True
    return False
```

```
def pairwise(iterable):

    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)


def draw_bins(img, centroids, root_output_path, seams, bins):

    binning_img = np.zeros(img.shape[0:2], dtype=np.uint8)
    binning_img.fill(255)

    locs = np.array(np.where(img[:, :, 0].reshape(-1) != 0))[0:2, :]
    binning_img = binning_img.flatten()
    binning_img[locs] = 211
    binning_img = binning_img.reshape(img.shape[0:2])
    binning_img = np.stack((binning_img,) * 3, axis=-1)

    draw_seams_red(binning_img, seams)
    overlay_img = binning_img.copy()

    for centroid, value in zip(centroids, bins):
        cv2.circle(overlay_img, (int(centroid[1]), int(centroid[0])), 25, (0, 255, 0), -1)
        cv2.putText(binning_img, str(int(value)), (int(centroid[1]) - 16, int(centroid[0]) + 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 0), 2, cv2.LINE_AA)
    cv2.addWeighted(overlay_img, 0.4, binning_img, 0.6, 0, binning_img)
    save_img(binning_img, path=os.path.join(root_output_path, 'energy_map',
'energy_map_bin_expl.png'))
```

Polygon Manager

```
import logging
import time
import cv2
import networkx as nx
import numpy as np
from skimage import measure



def get_polygons_from_lines(img, lines, connected_components, vertical):


    polygon_coords = []
    for i, line in enumerate(lines):
        cc_coords = []
        graph_nodes = []
        polygon_img = np.zeros(img.shape)
        for c in line:
            cc = find_cc_from_centroid(c, connected_components[1])
            points = cc.coords[::3, 0:2]
            points = np.asarray([[point[1], point[0]] for point in points])
            cc_coords.append(points)
            graph_nodes.append(find_graph_node(cc.coords, cc.centroid))


        overlay_graph = createTINgraph(np.array(list(set(tuple(p) for p in graph_nodes))))


        overlay_graph = nx.minimum_spanning_tree(overlay_graph)
```

```
        polygon_img = print_graph_on_img(polygon_img, [overlay_graph], color=(255, 255, 255),
thickness=1)
        cv2.fillPoly(polygon_img, cc_coords, color=(255, 255, 255))


        if vertical:
            polygon_img = cv2.rotate(polygon_img, cv2.ROTATE_90_CLOCKWISE)


        filter_size_H = 5
        filter_size_V = 5
        kernel = np.ones((filter_size_V, filter_size_H)) / filter_size_H

        polygon_img = cv2.filter2D(polygon_img, -1, kernel)


        polygon_coords.append(measure.find_contours(polygon_img[:, :, 0], 5,
fully_connected='high')[0])

    return polygon_coords


def find_graph_node(coords, centroid):

    centroid = np.asarray(centroid, dtype=int)


    if centroid in coords:
        return centroid
```

```
    return [coords[0][0], coords[0][1]]


def find_cc_from_centroid(c, cc_properties):

    for cc in cc_properties:
        if cc.centroid[0] == c[0] and cc.centroid[1] == c[1]:
            return cc
    print("If this is printed, you might want to uncomment the line swapping the coordinates!")
    return None


def draw_polygons(image, polygons, vertical):
    if vertical:
        image = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)

    for polygon in polygons:
        cv2.polylines(image, np.array([[[np.int(p[1]), np.int(p[0])] for p in polygon]]), 1,
        color=(248, 24, 148), thickness=3)
    return image


def polygon_to_string(polygons):

    start = time.time()

    strings = []
    for polygon in polygons:
        line_string = []
        for i, point in enumerate(polygon):
            if i % 3 != 0:
```

```
                continue
            line_string.append("{},{}".format(int(point[1]), int(point[0])))
        strings.append(' '.join(line_string))


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))

    return strings
```

Utils

Util XML:

```
import logging
import time
import cv2
import networkx as nx
import numpy as np
from skimage import measure


def get_polygons_from_lines(img, lines, connected_components, vertical):
    polygon_coords = []
    for i, line in enumerate(lines):
        cc_coords = []
        graph_nodes = []
        polygon_img = np.zeros(img.shape)
        for c in line:
            cc = find_cc_from_centroid(c, connected_components[1])
            points = cc.coords[::3, 0:2]
            points = np.asarray([[point[1], point[0]] for point in points])
```

```
            cc_coords.append(points)
            graph_nodes.append(find_graph_node(cc.coords, cc.centroid))

        overlay_graph = createTINgraph(np.array(list(set(tuple(p) for p in graph_nodes))))

        overlay_graph = nx.minimum_spanning_tree(overlay_graph)

        polygon_img = print_graph_on_img(polygon_img, [overlay_graph], color=(255, 255, 255),
thickness=1)
        cv2.fillPoly(polygon_img, cc_coords, color=(255, 255, 255))

        if vertical:
            polygon_img = cv2.rotate(polygon_img, cv2.ROTATE_90_CLOCKWISE)

        filter_size_H = 5
        filter_size_V = 5
        kernel = np.ones((filter_size_V, filter_size_H)) / filter_size_H

        polygon_img = cv2.filter2D(polygon_img, -1, kernel)

        polygon_coords.append(measure.find_contours(polygon_img[:, :, 0], 5,
fully_connected='high')[0])

    return polygon_coords
```

```
def find_graph_node(coords, centroid):

    centroid = np.asarray(centroid, dtype=int)


    if centroid in coords:
        return centroid

    return [coords[0][0], coords[0][1]]


def find_cc_from_centroid(c, cc_properties):
    for cc in cc_properties:
        if cc.centroid[0] == c[0] and cc.centroid[1] == c[1]:
            return cc
    print("If this is printed, you might want to uncomment the line swapping the coordinates!")
    return None


def draw_polygons(image, polygons, vertical):
    if vertical:
        image = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)

    for polygon in polygons:
        cv2.polylines(image, np.array([[[np.int(p[1]), np.int(p[0])] for p in polygon]]), 1,
         color=(248, 24, 148), thickness=3)
    return image


def polygon_to_string(polygons):

    start = time.time()
```

```
    strings = []
    for polygon in polygons:
        line_string = []
        for i, point in enumerate(polygon):
            if i % 3 != 0:
                continue
            line_string.append("{},{}".format(int(point[1]), int(point[0])))
        strings.append(' '.join(line_string))


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))

    return strings



import shutil
import cv2
import os
import numpy as np


def create_folder_structure(input_file, output_path, params):


    fileName = os.path.basename(input_file).split('.')[0]


    if not os.path.exists(output_path):
        os.mkdir(output_path)
```

```
    basefolder_path = os.path.join(output_path, fileName +
'_penalty_reduction_{}_seams_{}_component_ratio_{}'.format(*params))
    if not os.path.exists(basefolder_path):
        os.mkdir(basefolder_path)

        os.mkdir(os.path.join(basefolder_path, 'energy_map'))

        os.mkdir(os.path.join(basefolder_path, 'logs'))

        os.mkdir(os.path.join(basefolder_path, 'preprocess'))

    return basefolder_path



def save_img(img, path='experiment.png', show=False):
    if show:
        cv2.imshow('img', img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
        cv2.imwrite(path, img)


def calculate_asymmetric_distance(x, y, h_weight=1, v_weight=5):
    return [np.sqrt((((y[0] - x[0][0]) ** 2) * v_weight + ((y[1] - x[0][1]) ** 2) * h_weight)/
(h_weight+v_weight))]



def dict_to_string(dictionay):
    string = []
    for entry in dictionay.items():
        string.append('_'.join(entry))
    return '_'.join(string)
```

Util-graph:

```
 import bisect
import logging
import os
import time
import networkx as nx
import numpy as np
import cv2

from scipy.spatial import Delaunay
from shapely.geometry import LineString


def createTINgraph(points):

    start = time.time()
     TIN = Delaunay(points)
    edges = set()

    for n in range(TIN.nsimplex):

        edge = sorted([TIN.vertices[n, 0], TIN.vertices[n, 1]])
         edges.add((edge[0], edge[1], asymetric_distance(edge, points)))
        edge = sorted([TIN.vertices[n, 0], TIN.vertices[n, 2]])
        edges.add((edge[0], edge[1], asymetric_distance(edge, points)))
        edge = sorted([TIN.vertices[n, 1], TIN.vertices[n, 2]]
     edges.add((edge[0], edge[1], asymetric_distance(edge, points)))



    graph = nx.Graph()
    graph.add_weighted_edges_from(edges)
```

```
    original_nodes = points
    assert len(original_nodes) == graph.number_of_nodes()


    attributes = {}
    for n in range(len(original_nodes)):
        XY = original_nodes[n]
        attributes[n] = [XY[1], XY[0]]

    nx.set_node_attributes(graph, attributes, 'XY')


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return graph


def asymetric_distance(edge, points):
    return np.linalg.norm(
        np.asarray(points[edge[0]]) * np.array([1, 3]) - np.asarray(points[edge[1]]) *
          np.array([1, 3]))


def print_graph_on_img(img, graphs, color=(0, 255, 0), thickness=3):
    img = img.copy()
    for graph in graphs:
        for edge in graph.edges:
            p1, p2 = get_edge_node_coordinates(edge, graph)
            cv2.line(img, p1, p2, color, thickness=thickness)
```

```
    return img


def get_edge_node_coordinates(edge, graph):
    node_attributes = nx.get_node_attributes(graph, 'XY')
    p1 = np.asarray(node_attributes[edge[0]], dtype=np.uint32)
    p1 = (p1[0], p1[1])
    p2 = np.asarray(node_attributes[edge[1]], dtype=np.uint32)
    p2 = (p2[0], p2[1])
    return p1, p2


def cut_graph_with_seams(graph, seams, too_small_pc):
    start = time.time()
    tic = time.time()
    unique_edges, weights, occurrences = find_intersected_edges(graph, seams)
    logging.info("find_intersected_edges: {}".format(time.time()-tic))
    graph.remove_edges_from(unique_edges)
    if nx.is_connected(graph):
        return list([graph])


    graphs = np.asarray(list(nx.connected_component_subgraphs(graph)))
    small_graphs = detect_small_graphs(graphs, too_small_pc).tolist()


    while small_graphs:
        graph = merge_small_graphs(graph, list(small_graphs), unique_edges, weights)

        graphs = np.asarray(list(nx.connected_component_subgraphs(graph)))
```

```
        small_graphs = detect_small_graphs(graphs, too_small_pc)


    if nx.is_connected(graph):
        return list([graph])


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))


    return graphs


def get_neighbouring_seams_index(seams_max_y, seams_min_y, edge_max_y, edge_min_y):

    return [bisect.bisect_left(seams_max_y, edge_min_y), bisect.bisect_left(seams_min_y,
edge_max_y)]


def chunks(l, n):

    n = np.min([n, len(l)])
     size  = int(np.ceil(len(l)/n))
    for i in range(0, len(l), size):
        yield l[i:i + size]


def find_intersected_edges(graph, seams):

    seams_y = [np.array(s)[:, 1] for s in seams]
    seams_max_y = np.max(seams_y, axis=1)
```

```
    seams_min_y = np.min(seams_y, axis=1)


    node_attributes = nx.get_node_attributes(graph, 'XY')


    seams = [LineString(seam) for seam in seams]


    edges = [e for e in graph.edges]
    edges.sort(key=lambda tup: tup[0])

    edges_to_remove = []

    for chunk in chunks(edges, 250):

        tmp = np.array(chunk)
        p1 = np.array([node_attributes[edge[0]] for edge in tmp])
        p2 = np.array([node_attributes[edge[1]] for edge in tmp])
        edge_max_y = np.max((p1[:, 1], p2[:, 1]))
        edge_min_y = np.min((p1[:, 1], p2[:, 1]))

        index = get_neighbouring_seams_index(seams_max_y, seams_min_y, edge_max_y,
edge_min_y)

        for start, end, edge in zip(p1, p2, chunk):
            line_edge = LineString([start, end])
            for seam in seams[index[0]:index[1]]:
                if line_edge.intersects(seam):
                    edges_to_remove.append(edge)
                    break
```

```
    unique_edges, occurrences = np.unique(np.array(edges_to_remove), return_counts=True,
     axis=0)
    weights = [graph.edges[edge]['weight'] for edge in unique_edges]

    return unique_edges, weights, occurrences


def merge_small_graphs(graph, small_graphs, unique_edges, weights):

    edges_to_add = []
    weights = np.asarray(weights)


    while small_graphs:

        small_graph = small_graphs.pop()
        edge_idxs = np.unique(np.hstack(np.asarray(
            [np.where(unique_edges == node)[0] for node in list(small_graph.nodes)])))

        min_edge_idx = edge_idxs[np.argmin(weights[edge_idxs])]

        edge = unique_edges[min_edge_idx]
        unique_edges = np.delete(unique_edges, min_edge_idx, axis=0)
        weights = np.delete(weights, min_edge_idx, axis=0)
        edges_to_add.append((edge[0], edge[1], weights[min_edge_idx]))

    graph.add_weighted_edges_from(edges_to_add)
    return graph


def graph_to_point_lists(graphs):
```

```
    return [list(nx.get_node_attributes(graph, 'XY').values()) for graph in graphs]


def detect_small_graphs(graphs, too_small_pc):

    start = time.time()

    graph_sizes = np.asarray([len(g.nodes) for g in graphs])

    too_small = graph_sizes < too_small_pc * np.mean(graph_sizes)


    stop = time.time()
    logging.info("finished after: {diff} s".format(diff=stop - start))

    return graphs[too_small]
```

util-graph_logger:

```
import cv2
import os
import numpy as np
class GraphLogger:
    IMG_SHAPE = ()
    ROOT_OUTPUT_PATH = ''
```

```
@classmethod
def draw_graphs(cls, img, graphs, color=(0, 255, 0), thickness=3, name='graph.png'):
    if not list(img):
        img = np.zeros(cls.IMG_SHAPE)
    else:
        img = img.copy()

    for graph in graphs:
        img = cls.draw_graph(img, graph, color, thickness, False)

    save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name),
    show=False)

    return img

@classmethod
def draw_graph(cls, img, graph, color=(0, 255, 0), thickness=3, save=False,
name='graph.png'):
    if not list(img):
        img = np.zeros(cls.IMG_SHAPE)
    else:
        img = img.copy()

    cls.draw_edges(img, graph.edges, graph, color, thickness, save=False)

    if save:
        save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name),
         show=False)

    return img
```

```
    @classmethod
    def draw_edges(cls, img, edges, graph, color, thickness, save=False, name='graph.png'):
        for edge in edges:
            p1, p2 = get_edge_node_coordinates(edge, graph)
            cv2.line(img, p1, p2, color, thickness=thickness)

        if save:
            save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name),
            show=False)
```

Line Segmentation

```
import cv2
import os
import numpy as np



class GraphLogger:
    IMG_SHAPE = ()
    ROOT_OUTPUT_PATH = ''

    @classmethod
    def draw_graphs(cls, img, graphs, color=(0, 255, 0), thickness=3, name='graph.png'):
        if not list(img):
            img = np.zeros(cls.IMG_SHAPE)
        else:
            img = img.copy()

        for graph in graphs:
            img = cls.draw_graph(img, graph, color, thickness, False)
```

```
        save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name)
        , show=False)

        return img

    @classmethod
    def draw_graph(cls, img, graph, color=(0, 255, 0), thickness=3, save=False,
    name='graph.png'):
        if not list(img):
            img = np.zeros(cls.IMG_SHAPE)
        else:
            img = img.copy()

        cls.draw_edges(img, graph.edges, graph, color, thickness, save=False)

        if save:
            save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name),
            show=False)

        return img

    @classmethod
    def draw_edges(cls, img, edges, graph, color, thickness, save=False, name='graph.png'):
        for edge in edges:
            p1, p2 = get_edge_node_coordinates(edge, graph)
            cv2.line(img, p1, p2, color, thickness=thickness)

        if save:
            save_img(img, path=os.path.join(cls.ROOT_OUTPUT_PATH, 'graph', name),
             show=False)
```

```
 extract_textline(input_path='test1.png', output_path= './output', penalty_reduction = 6000,
sseam_every_x_pxl= 100,testing=True, vertical=False, console_log= False, small_component_

ratio= 0.1)

logging.info('Terminated')
```

CHAPTER-6
SYSTEM TESTING

6.1 INTRODUCTION

The cause of testing is to detect mistakes. Making an attempt out is the technique of looking for to realize each viable fault or weakness in a piece product. It presents a method to determine the performance of add-ons, sub-assemblies, assemblies and/or a completed product. It is the method of ex excising g program with the intent of constructing certain that the application procedure meets its necessities and client expectations and does no longer fail in an unacceptable process. There are rather plenty of forms of scan. Each experiment sort addresses a special trying out requirement.

6.2 TYPES OF TESTS:

Unit testing:

Unit testing is undertaken when a module has been created and successfully reviewed. In order to test a single module we need to provide a complete environment i.e. besides the module we would require

- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that module accesses.
- A procedure to call the functions of the module under test with appropriate parameters

In this testing individual components of the project status detecting, sound alerting, automatic mail sub functions are verified successfully.

Integration testing:

The documents created using the main code should be sent by mailing module correctly. If not reports of the documents won't reach the agency correctly.

Functional testing:

Testing on the functionality of the driver monitoring system for travel agencies include whether system correctly detecting the state of the driver correctly or not and mail is sent to correct destination or not and sound alert testing. Functionality testing done successfully.

System Testing:

Testing of whole system is done after the completion all the required sub functions like automatic mail sending and sound alert. System testing is completed successfully.

White Box Testing:

This testing is a trying out wherein where the application tester has competencies of the interior workings, constitution and software language, or at least its cause. It's rationale. It's used to test areas that can't be reached from a black box stage.

Black Box Testing:

This is testing the software with none advantage of the inside workings, establishment or words of the unit life form veteran.

Acceptance Testing:

User Acceptance testing trying out is a crucial section of any mission and requires enormous participation by the tip user. It additionally ensures that the procedure meets the functional specifications.

Test Results:

The entire test cases recounted above passed effectually. No defects Encountered.

Conclusion

This project “TEXT LINE SEGMENTATION FOR MEDIEVAL MANUSCRIPTS” has been successfully executed by our team to produce segmented lines of handwritten text for our dataset and it is found that our proposed system provides efficient and effective result when compared to existing system. Our project provides text with polygons which specifies the segmentation of handwritten text.

In conclusion, text line segmentation for medieval manuscripts is a critical task that plays a vital role in the analysis, digitization, and preservation of these valuable historical documents.

Appendices

References:

- Image and Text Segmentation pipeline for the paper **"Labeling, Cutting, Grouping : an Efficient Text Line Segmentation Method for Medieval Manuscripts",** published at the 15th IAPR International Conference on Document Analysis and Recognition (ICDAR) in 2019.

- Research Gate – Text line segmentation for handwritten documents.

- IEEE Papers for Text Line Segmentation.