Justyce Countryman

May 5, 2024

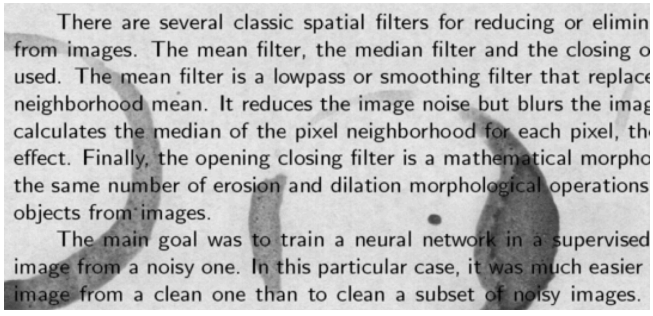
CSC 474

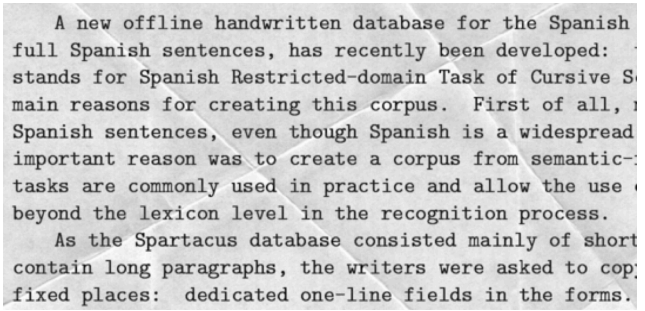
Professor Coman

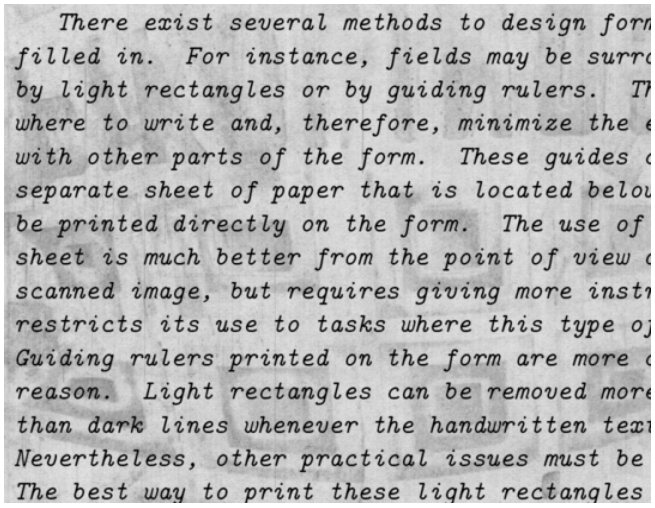
Text Document Enhancer

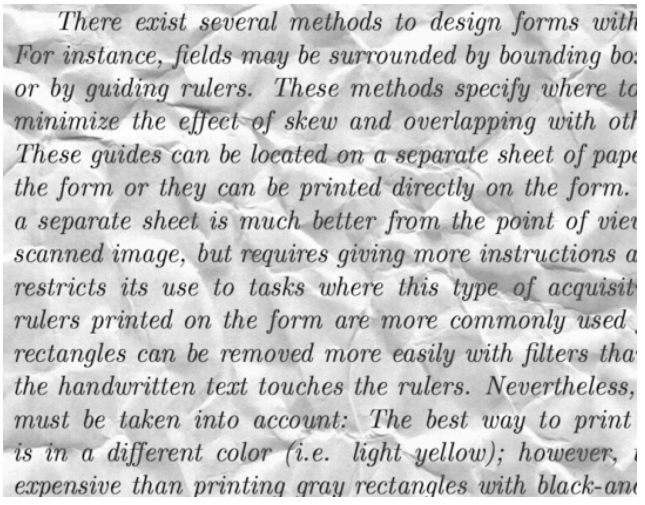
Section 1: Introduction, Team Members, and Project Vision

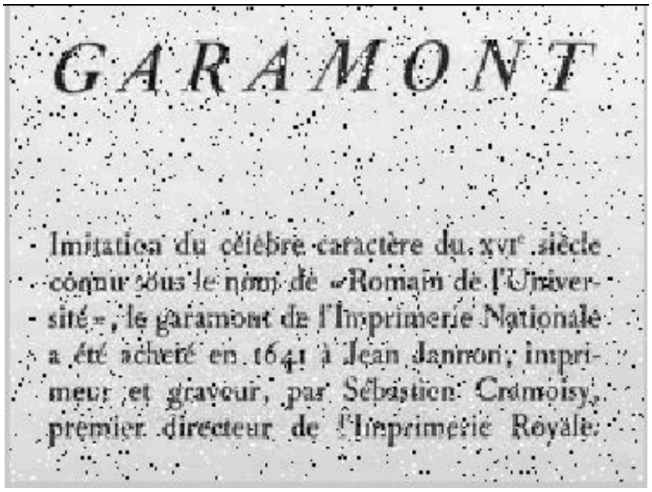
The Text Document Enhancer application was done as a semester-long project at SUNY Oswego for the CSC 474 - Image Processing course. The project was accomplished single-handedly by Justyce Countryman through the utilization of Python research, primarily with image processing libraries and techniques. The heavy passion for this project was due to readers, whether people or computers, oftentimes having trouble comprehending documents that have strong background noise, which hinders text readability confidence. The goal for this project was to reduce as much of the noise as possible while maintaining a high level of text clarity. Some examples of noisy images are shown below:

(1) 

(2) 

(3) 

(4) 

(5) 

(1), (2), (3), and (4): (Espaa-Boquera, Pastor-Pellicer, Castro-Bleda, & Zamora-Martinez, 2015)

(5): (Balamurugan, Sengottuvelan, & Sangeetha, 2013)

Is it clear that there are many types of noise that lower readability confidence. It is also a challenge to consider all types of noise, since there is virtually an infinite number of them. However, there are several image preprocessing techniques that can help reduce common types of background noise found in hard-to-read text documents. For the sake of noise descriptions, this report will refer to (1), (2), (3), (4), and (5) as coffee-stained, folded, eraser-marked, crumpled-up, and salt-and-pepper noises respectively. In addition to increasing text readability, this project can be successful with restoring the clarity of distorted historical documents, and conducting research into which image preprocessing methods work the best under certain noisy conditions. In other words, this project may potentially aid with the development of complex and effective image preprocessing algorithms that handle other variations of background noise.

Section 2: Project Description & Algorithms

Section 2.1: Project Description

Most components in the Text Document Enhancer project rely on image preprocessing methods available in a Python library known as OpenCV. Nevertheless, the project extends these methods into a software application where users may easily insert their noisy text document images, and improve text quality and readability. A primary objective to this project was to find enough image preprocessing algorithms in which all general types of noise are considered. As a result, the project is divided into nine subprograms based on commonly used image preprocessing techniques. The available methods include the Otsu's, binary, and adaptive thresholding techniques, the opening-closing and closing-opening mathematical morphologies, and the median, Gaussian blur, and bilateral filtering strategies. The last subprogram involves

automatically utilizing the image preprocessing methods that gave the most promising results based on five datasets of noisy images and their clean counterparts. Any of these subprograms will also allow the user to retrieve the histogram of the currently displayed image, extract its text through the Tesseract Python library, save the preprocessed image, and provide a clean image so that there is a gold-standard that the noisy image can strive for. If a clean image is provided, the user can get the text from the clean and noisy counterparts, and receive a text extraction accuracy from 0% to 100%, which describes what percentage of the text from the noisy version is similar to the clean version.

Section 2.2: Project Algorithms

There are many algorithms that were needed to complete the Text Document Enhancer project, but it is beneficial to first discuss the algorithm that accomplishes the overall purpose of this project, cleaning the noise out of a text document image.

General Purpose Algorithm:

While the user has not quit the program:
1: The user selects one of the subprograms.
While the user has not quit the subprogram:
2: The user provides an image (preferably a text document).
3: The grayscale version of the image is then calculated.
4: Allow the user to change relevant parameters for the chosen subprogram, if any.
5: When requested by the user, perform the desired preprocessing operation, or a variant of the method if necessary, on the grayscale image with OpenCV, which will then be converted into a binary image to display to the user.
6: Dynamically update the displayed image based on changes to the parameters.

Note: At any time during the subprogram, the user may extract text and/or get the histogram of the noisy image currently being displayed, whether preprocessed or not, as well as provide a clean version of the image.

It is also convenient to mention the text extraction accuracy algorithm, which calculates the cost to go from the text of a noisy image to its clean counterpart.

Text Extraction Accuracy Algorithm:

1: Extract the text from the noisy and clean images.
2: Initialize a matrix with (number of words from the noisy image + 1) rows and (number of words from the clean image + 1) columns.
For each word in the noisy image:
For each word in the clean image:
If on the first row or the first column:

3: Set the base distances that indicate the cost to get from a empty string to the current string.

Otherwise, if the current noisy word is the same as the clean word:

4: Keep the distance the same.

Otherwise:

5: Check the distance for previously calculated neighbors and find the smallest distance + 1.

6: Set accuracy for the noisy image as ((max_distance - calculated_distance) / max_distance) * 100), where max_distance is the length of the extracted text string from either the noisy or clean image, whichever is longer, and calculated_distance is the cost calculated by the algorithm, which is found in the last row and column of the matrix.

The last significant algorithm to consider is how the project automatically decides which preprocessing method is superior in cleaning up as much of the noise from the user-provided image as possible without requiring any parameters. In the application, this subprogram is called “Get Best Image.”

“Get Best Image” Algorithm:

While the user has not quit the “Get Best Image” subprogram:

1: The user provides an image (preferably a text document).

2: The grayscale version of the image is then calculated.

3: When the user wants to begin the preprocessing, send the grayscale image through several effective preprocessing methods that are the most likely to reduce background noise from the image.

4: Display the binary image that Tesseract believes is the most recognizable, not essentially the cleanest image.

Note: If a clean image counterpart is provided, the preprocessed image that is displayed will instead be based on the binary image that has the highest text extraction accuracy with respect to the clean image.

Together, these three algorithms become the heart of the entire Text Document Enhancer project, which are put to the test during project implementation and performance.

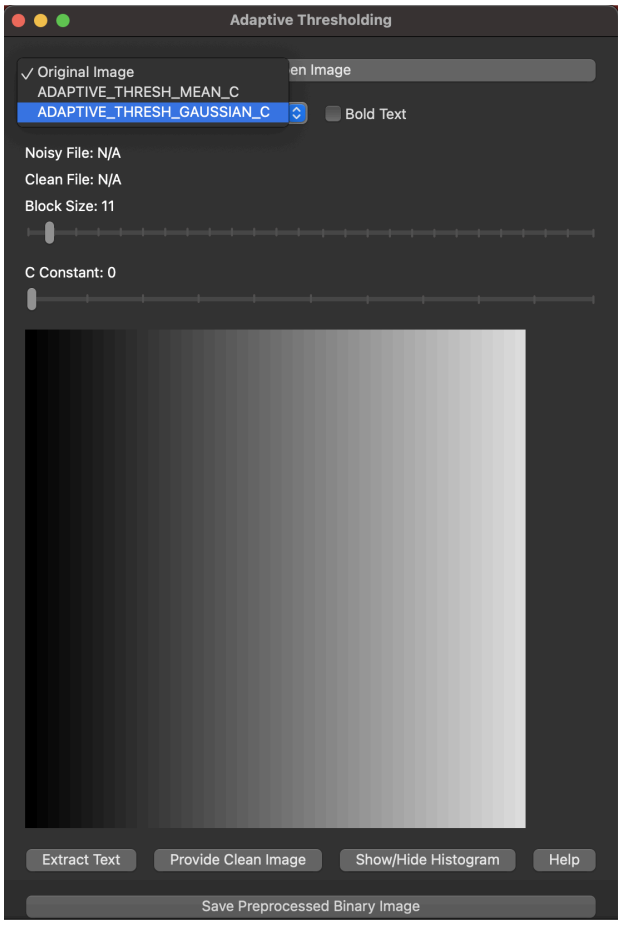
Section 3: Implementation Process

Project code: https://github.com/lljustycell999/Text-Document-Enhancer

Section 3.1: Algorithms

Constructing the algorithms from section 2.2 did require some careful consideration, with one result being the inclusion of constraints to improve the enhancement capabilities of the project for all tested types of noise.

Firstly, the general purpose algorithm was implemented on all of the preprocessing methods with little technical problems. When the user wishes to put in a noisy image, the subprogram will open up a file dialog so the user can do so, and is then automatically converted into a grayscale image. The only significant consideration was what preprocessing parameters the user should have the ability to modify and what their interval limitations should be. For example, the project only allows for odd integer kernel sizes for adaptive thresholding, median filtering and Gaussian blur due to dataset testing and OpenCV limitations. If there are available variations of the preprocessing method, like inverse binary thresholding or choosing between a mean or weighted sum in adaptive thresholding, the user may choose from any of these options from the respective subprogram's combo box, with a tutorial screenshot shown below:



Moreover, the "Get Best Image" algorithm needed the most adjustments out of the three main algorithms. When a user provides a noisy text document and chooses to begin preprocessing, the subprogram only considers the most successful preprocessing methods

that were tested on multiple image samples from each of the five types of noise previously mentioned. Based on calculated text extraction accuracies, here are the best preprocessing methods for each type of noise:

Coffee-stained: Adaptive thresholding
Eraser-marked and folded: Otsu's or binary thresholding
Crumpled-up: Binary thresholding or adaptive thresholding
Salt-and-pepper noise: Median filtering

Originally, salt-and-pepper noise was eliminated the most by bilateral filtering. However, it was later discovered that the dataset for this type of noise consisted of only binary images, which explains why the other preprocessing methods had no luck in removing any noise when dealing with this dataset. More importantly, keeping this method in the subprogram hindered the results of other images that did not have salt-and-pepper noise. This method would almost always be considered the most recognizable according to Tesseract, but it barely removed any of the noise, it just made the text a little clearer to the computer. This information helped with the final decision of which preprocessing methods would be implemented into this subprogram. On top of that, the "Get Best Image" subprogram will also not consider binary images that lost a substantial amount of text in contrast to the original noisy image. The reasoning behind this adaptation was because Tesseract was feeling confident about preprocessed images that had hardly any extractable text. In many cases, the noise was not reduced. To verify this claim, data analysis was conducted between the original noisy images and their preprocessed image counterparts. By observation, the problematic magnitude of Tesseract missing text in the enhanced images was greater than Tesseract adding extra text. As a result, the "Get Best Image" subprogram only considers preprocessed binary images where the length of its text string is at least 90% of that of the original text string. Implementing the final part of the algorithm to check for the binary image that Tesseract believes is the most recognizable was done through a Tesseract method called image_to_data(), which can return confidence percentages for every word it extracts in the image. Tesseract can however return confidence values of -1, meaning there was no confidence at all for the current word, so the project does not consider them during the confidence calculation. The overall text document confidence is simply the sum of all of the word confidence values divided by the number of words in the extracted text. Keep in mind that in order to avoid long wait times, especially for large images, this algorithm utilizes concurrency with eight cores. The multithreading goes into effect once the subprogram begins using the listed preprocessing methods, with the exception of Otsu's thresholding. Each of the other preprocessing techniques will consist of two sets of reasonable parameters based on tested image samples from all five previously stated datasets, along with a decision on

whether to try bolding the text, which would be done with a foreground dilation with a matrix size of 2x2, or not.

Lastly, the text extraction accuracy algorithm went according to plan without the need to make any modifications. Nonetheless, this algorithm was very valuable in creating the project with high quality in mind.

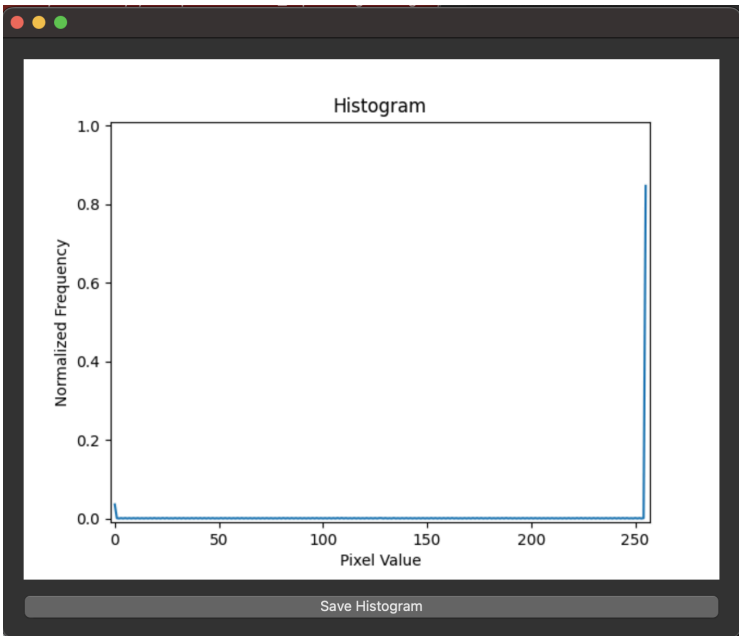
Section 3.2: Additional Features

To allow for plenty of user interaction with the project and image processing as a whole, there are more features available to the user within the GUI that rely on the algorithm implementations discussed in section 3.1.

To begin, any of the preprocessing subprograms provide the additional feature of bolding the text, which is accomplished by using a 2x2 kernel matrix, inverting the given grayscale image, performing dilation with the kernel matrix and the inverted image, and un-inverting the resulting image. At times, the "Get Best Feature" subprogram may recommend bolding the text in order to get the most enhanced binary image.

In all subprograms, the user may also extract text through the Tesseract image_to_string() method, provide a clean image when prompted by a file dialog, show the histogram of the currently displayed noisy image and save it to a user-selectable directory, save a preprocessed binary image, and display help that should let the user learn about the purpose and functionality of the subprograms and the entire project. Also, when a clean image is provided, pressing the "Extract Text" button will display the text Tesseract was able to obtain from both the noisy and clean images, conduct the text extraction accuracy algorithm, and display the accuracy to the user.

As a brief note, the histogram of the noisy image is retrieved from an OpenCV method called calc_hist(), in which two of the parameters include the size and range of the histogram. For clarity, the histogram will always have a size of 256 and a range from 0 to 255, indicating the pixel values for the x-axis. The y-axis will give a normalized frequency value between 0.0 and 1.0 for each pixel value. A sample histogram implemented into the GUI is given below:



Apart from this, the Otsu's thresholding subprogram exclusively contains buttons that permit the user to provide a clean and noisy directory of images. Once both are provided by the user, Otsu's thresholding is automatically performed on all images in the noisy directory, and the resulting preprocessed images are compared with their ideal clean versions. This feature is advantageous for running Otsu's thresholding on a large quantity of images as well as testing the performance of the preprocessing method under various types of background noise. Be advised that the directories will be checked alphabetically, meaning the project will assume the first image in both directories alphabetically are the noisy and clean counterparts of the same image, followed by the next image in both directories alphabetically, and so on. At the end of this feature, a bulk text extraction accuracy will be calculated and displayed with respect to the number of image comparisons completed.

Section 3.3: Error Handling & Quality Assurance

In addition to testing the noisy datasets to ensure there is always a preprocessing method available that should reduce background noise from virtually any text document, the software application itself also has to be maintained so that no error-prone data is processed. The most noticeable error handling in this project is that opening and saving images is only permitted with specific file extensions, specifically .png, .tiff, .jpeg, .jpg, and .bmp. The reason for these choices is to deny non-image file extensions like .txt as well as image file extensions that OpenCV does not support, like .pdf. The subprogram will warn the user if he or she closes the file dialog without supplying an image or if the subprogram has a problem with retrieving the desired image, and thus not update anything in the GUI. Continuing on, the user is able to adjust the parameters of each preprocessing method when available. The possible parameter values must be limited to what OpenCV can handle, like the previously discussed situation in section 3.1 where adaptive thresholding, median filtering, and Gaussian blur require odd kernel sizes. Lastly, before starting any preprocessing in the "Get Best Image" subprogram, the application first takes the initial noisy image and checks if there is any extractable text. If Tesseract cannot obtain any text, the user is given an error that says an ideal image cannot be calculated. The issue with having no text in the noisy image involves an almost guaranteed divide by zero error when trying to calculate the most recognizable binary image. All other error handling cases involved checking that all of the GUI features are always based on the displayed images. In other words, the GUI is dynamically updated whenever a new noisy or clean image is provided.

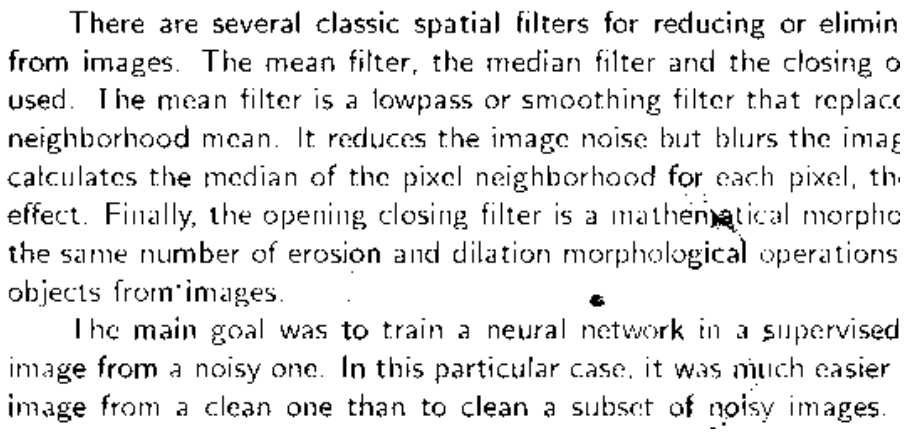
Moreover, the project itself includes convenient implementation decisions that improve the completeness and quality of the project. Firstly, the application will tell the user what files are being used as the noisy and clean image counterparts at all times, which contributes to helping users take advantage of the entire project with ease. The user-changeable parameters being used for the relevant preprocessing subprograms are

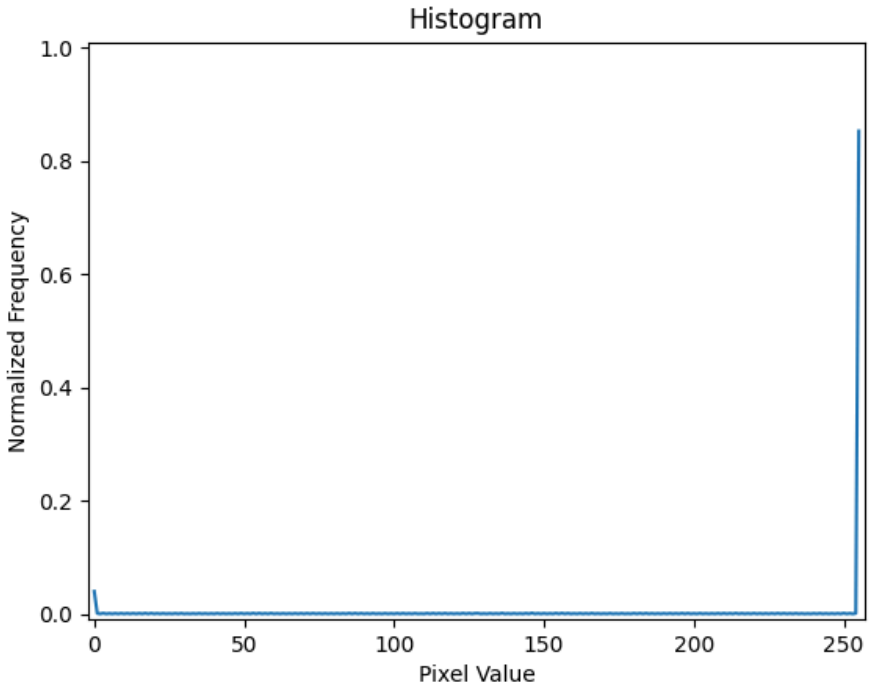
also dynamically updated. In the case of the “Get Best Image” subprogram, once an ideal preprocessed binary image is calculated, the GUI states which image preprocessing method was used, along with its parameters and whether or not bolding was necessary to achieve the produced image. Finally, it is vital to consider the size of the image being inserted into the GUI. Testing was assured to verify that all of the GUI features are reachable on a 13-inch computer display. If a provided image is too large to be displayed, the image will be compressed to an acceptable size that will still allow for access to the entire GUI. On top of that, all preprocessing, saving, text extraction, accuracy, and histogram methods will always be conducted on the original dimensions of all given images. This quality testing guarantees that the application will always support usability and functionality, regardless of image size.

Section 4: Project Results

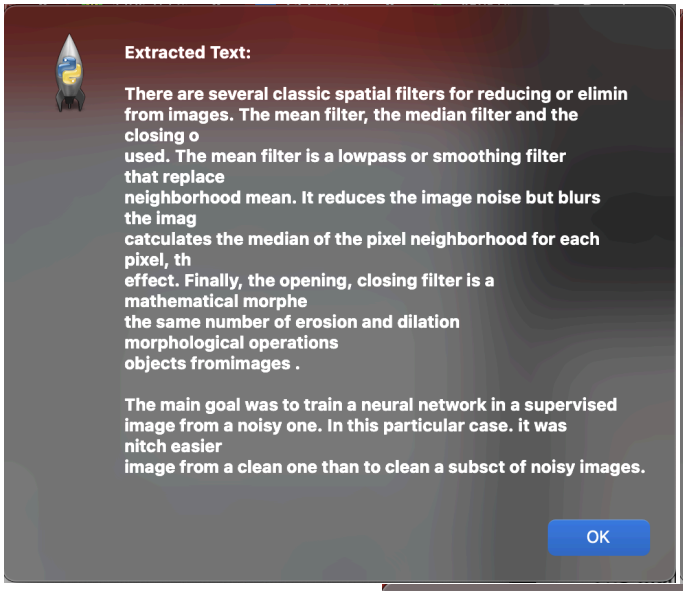
With the implementation process completed, it is now possible to test out noisy text documents to see how much noise is removed and how close the resulting preprocessed binary images are to their clean version counterparts. For the sake of discussion, this section will use the sample images from section 1 since each general type of background noise is considered. This section will also only cover the results from the “Get Best Image” subprogram since the output is always consistent. Although the project can run, and oftentimes be more accurate, with a clean image counterpart, the results from this section will be based on the assumption that no clean counterpart is accessible to the user, which will replicate the realistic utilization of this project. With that out of the way, here are the resulting binary images, histograms, accuracies, and extracted texts for each of the five noisy image samples after going through the “Get Best Image” subprogram without a clean image counterpart:

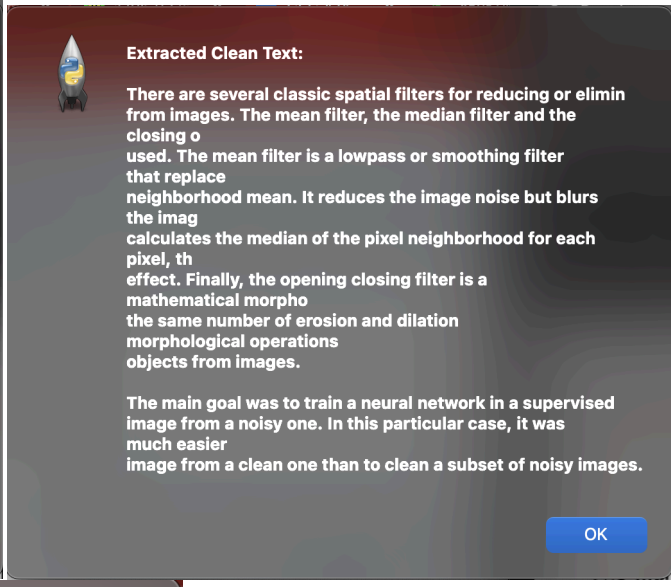
(1) Coffee-stained:

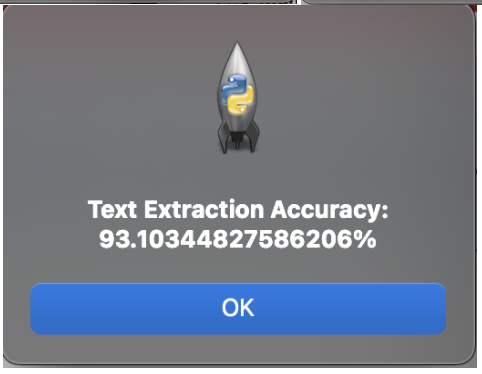




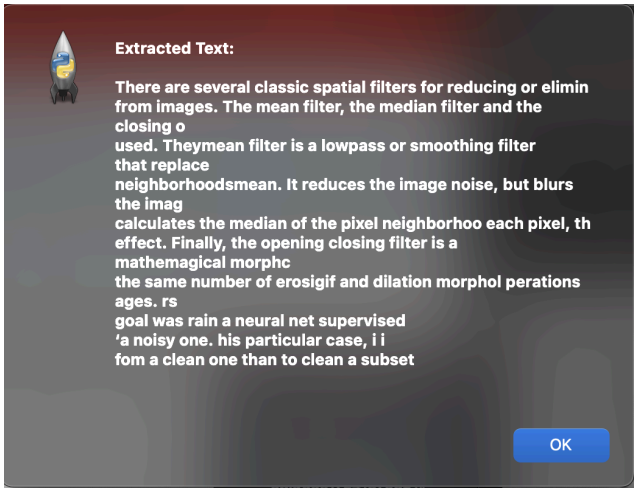
A clean image counterpart is provided at this point in order to calculate a text extraction accuracy:

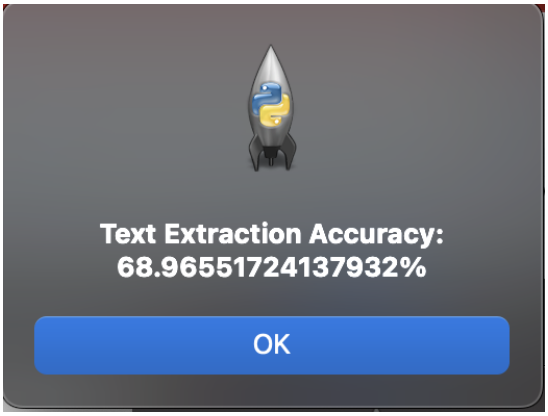




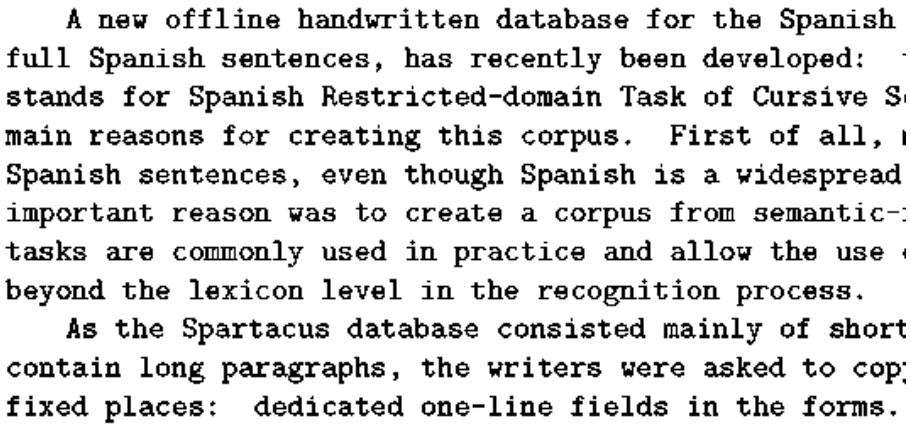


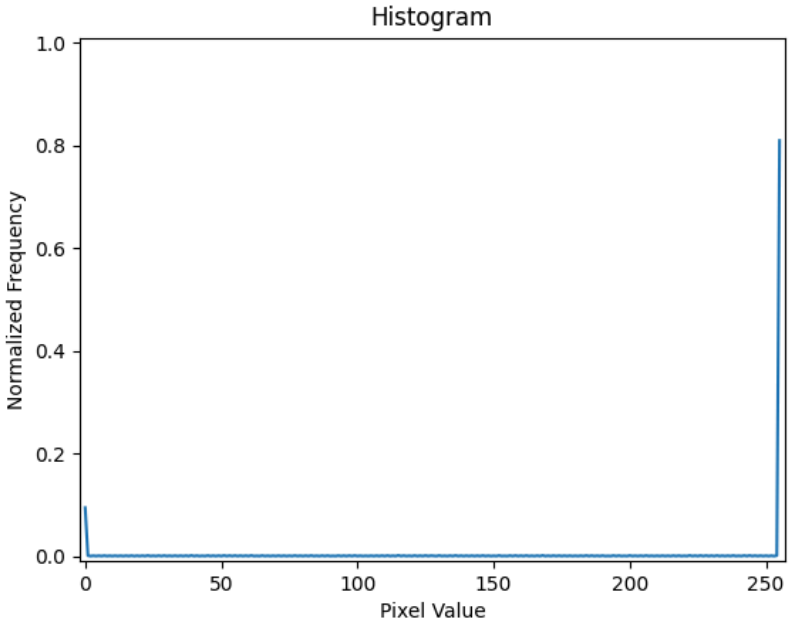
For reference, the text extraction accuracy of the original noisy image without any preprocessing is shown as well:



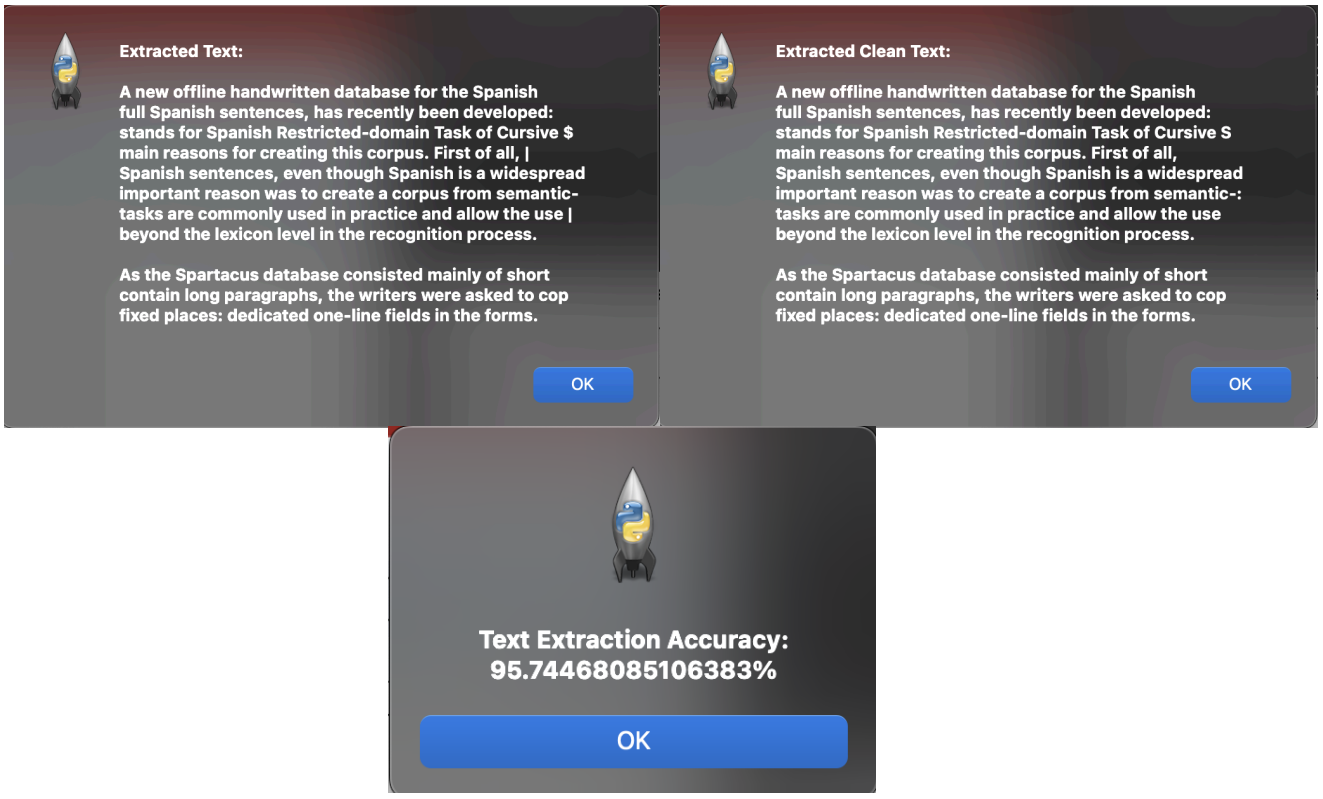


(2) Folded:

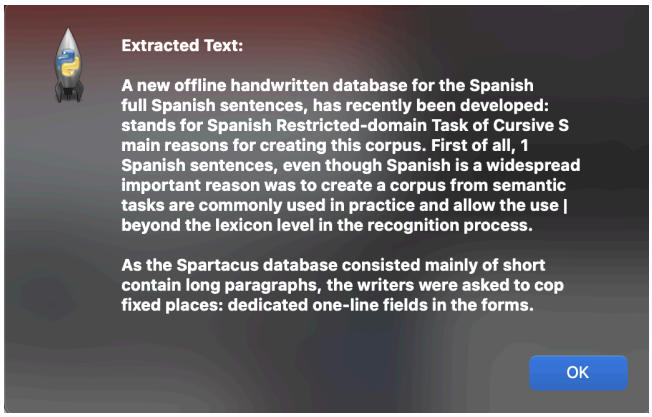


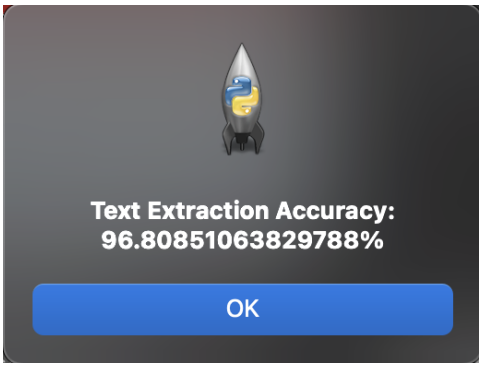


Text extraction accuracy:

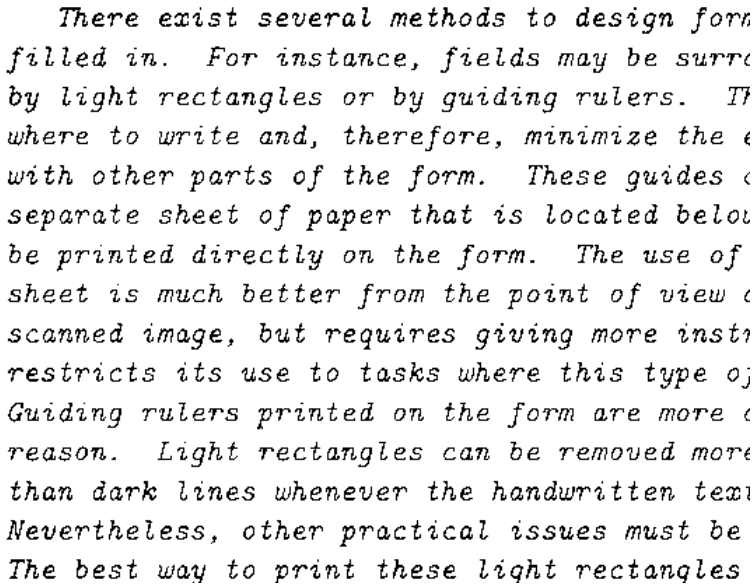


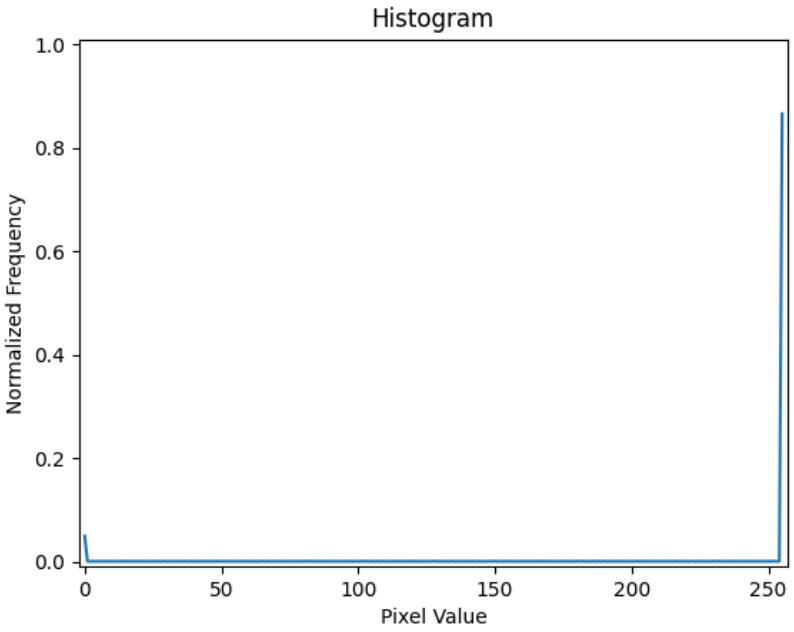
Text extraction accuracy with no preprocessing:



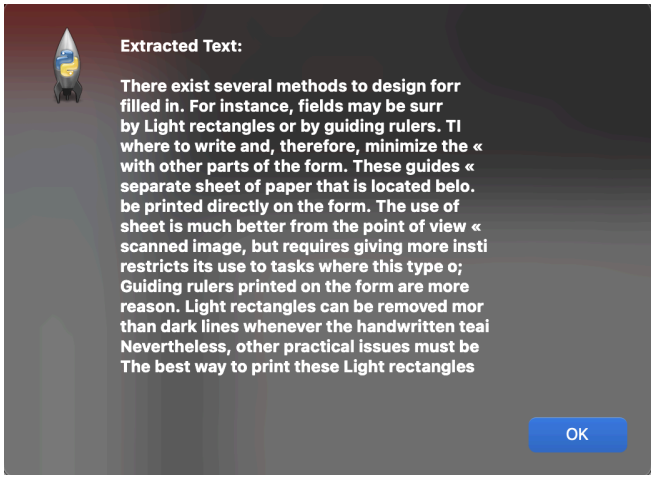


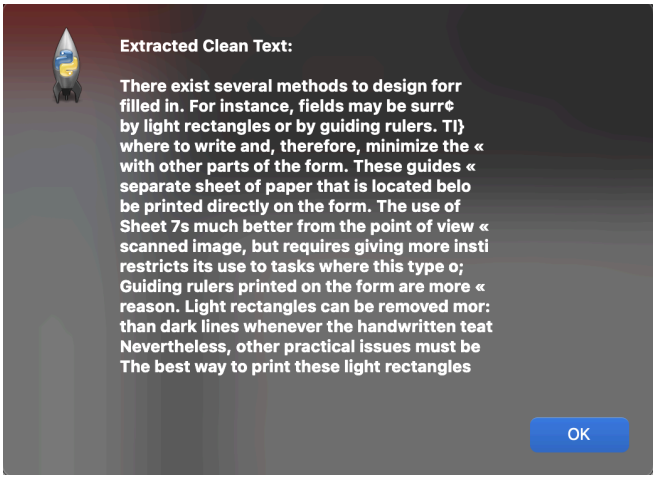
(3) Eraser-marked:

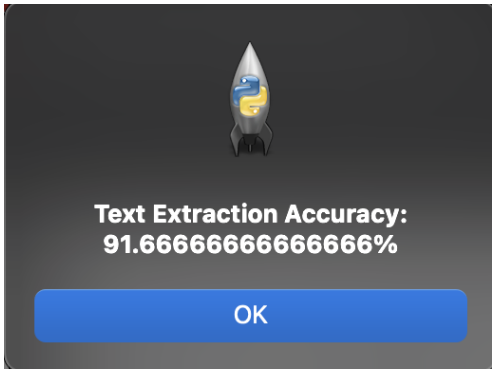




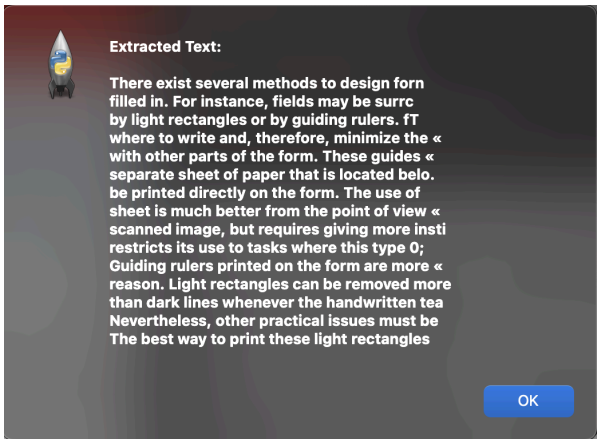
Text extraction accuracy:

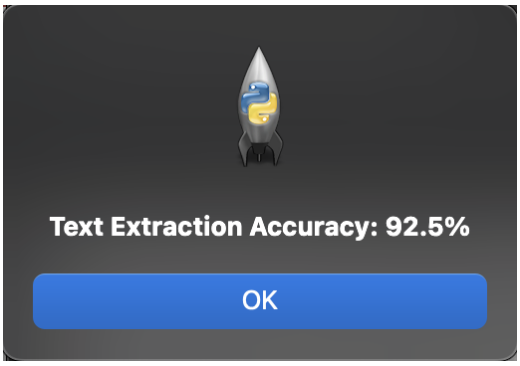




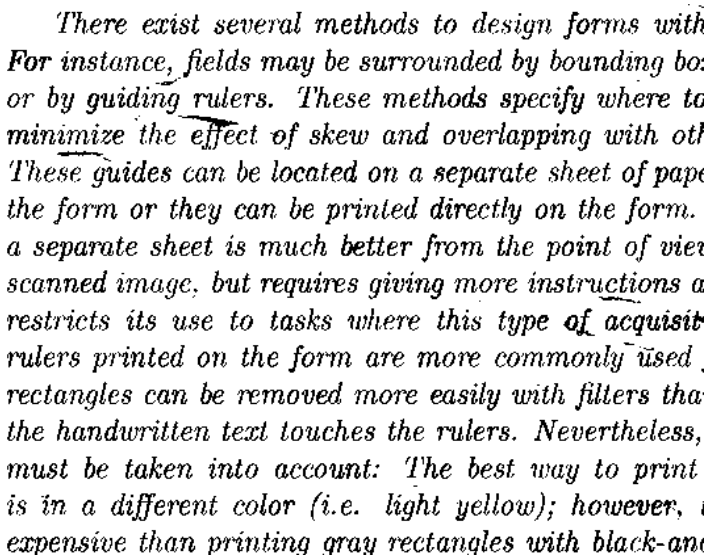


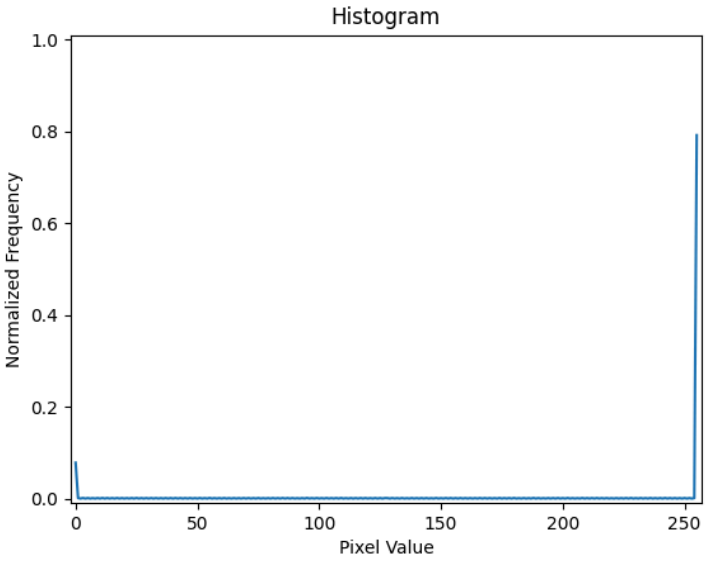
Text extraction accuracy with no preprocessing:



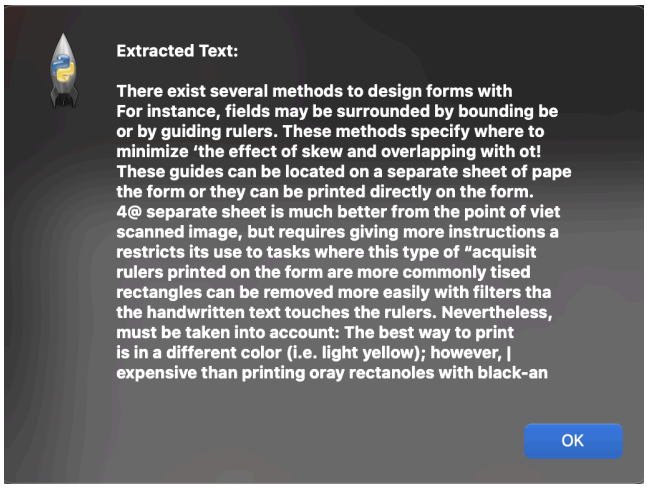


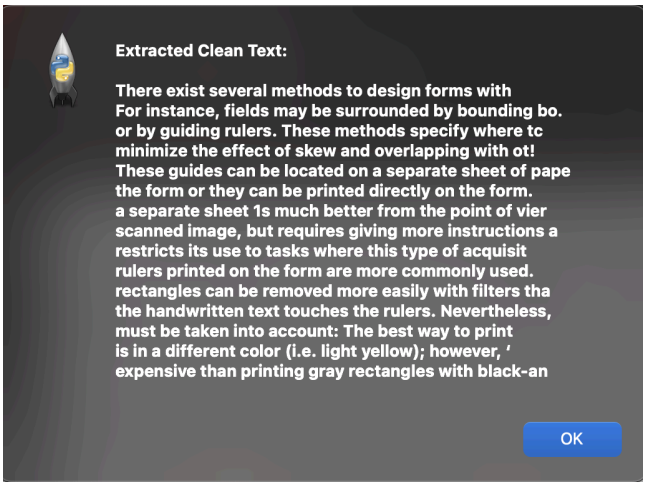
(4) Crumpled-up:

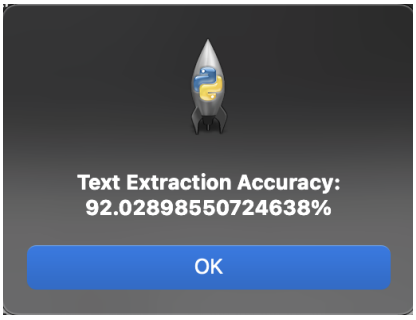




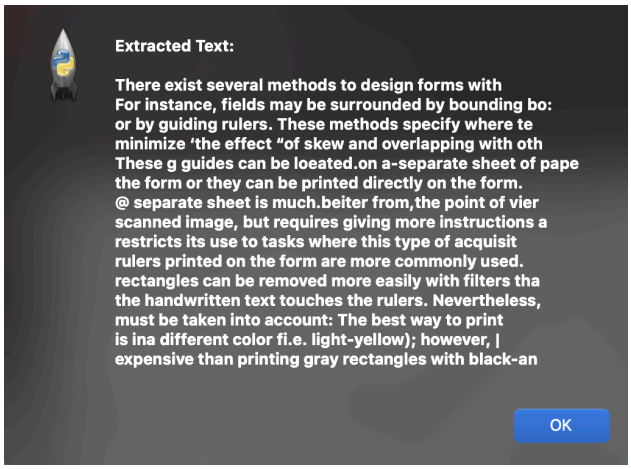
Text extraction accuracy:

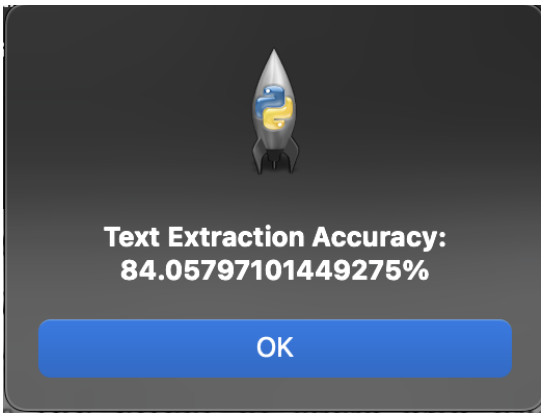




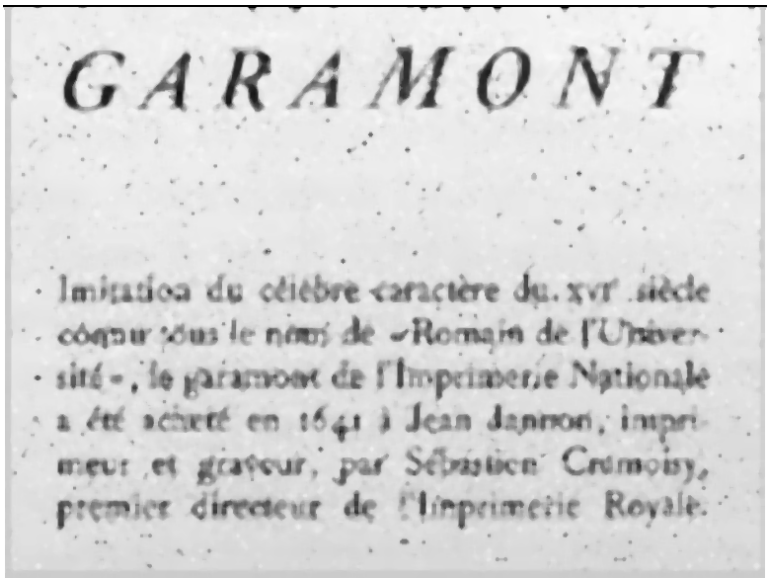


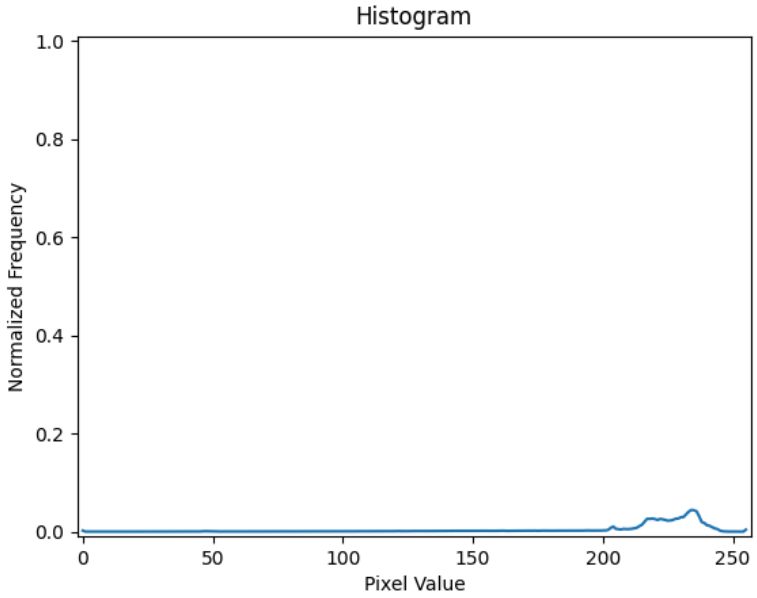
Text extraction accuracy with no preprocessing:



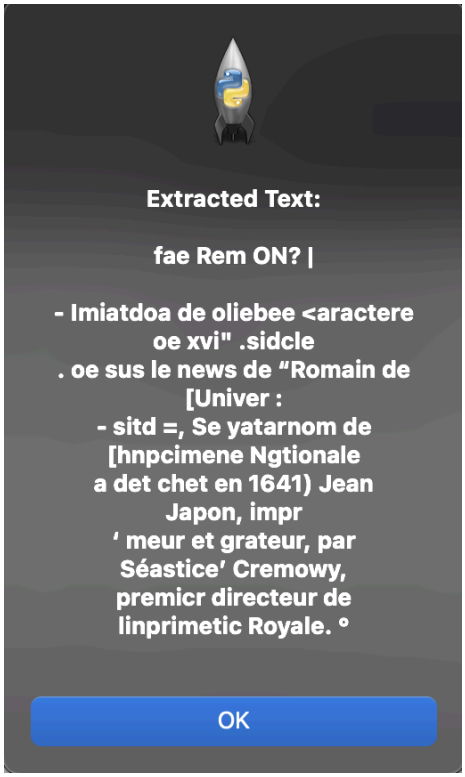


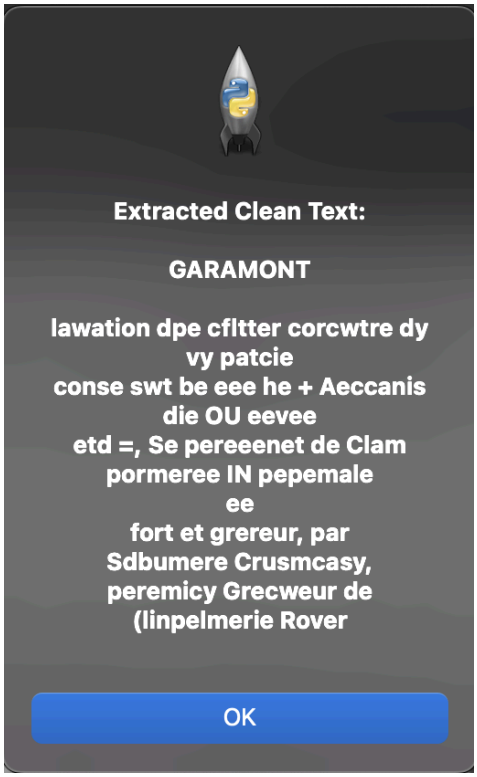
(5) Salt-and-pepper noise:

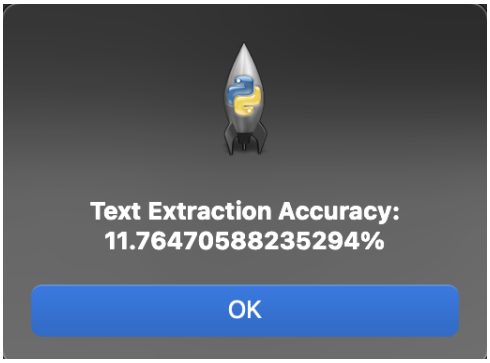




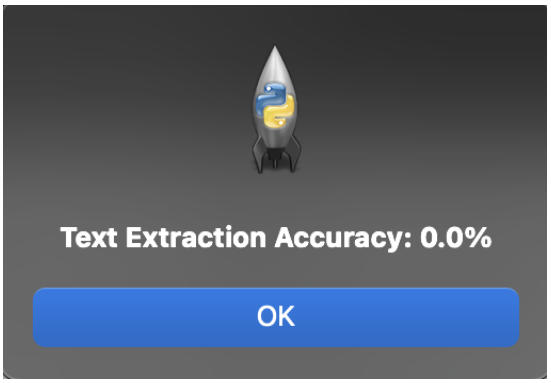
Text extraction accuracy:



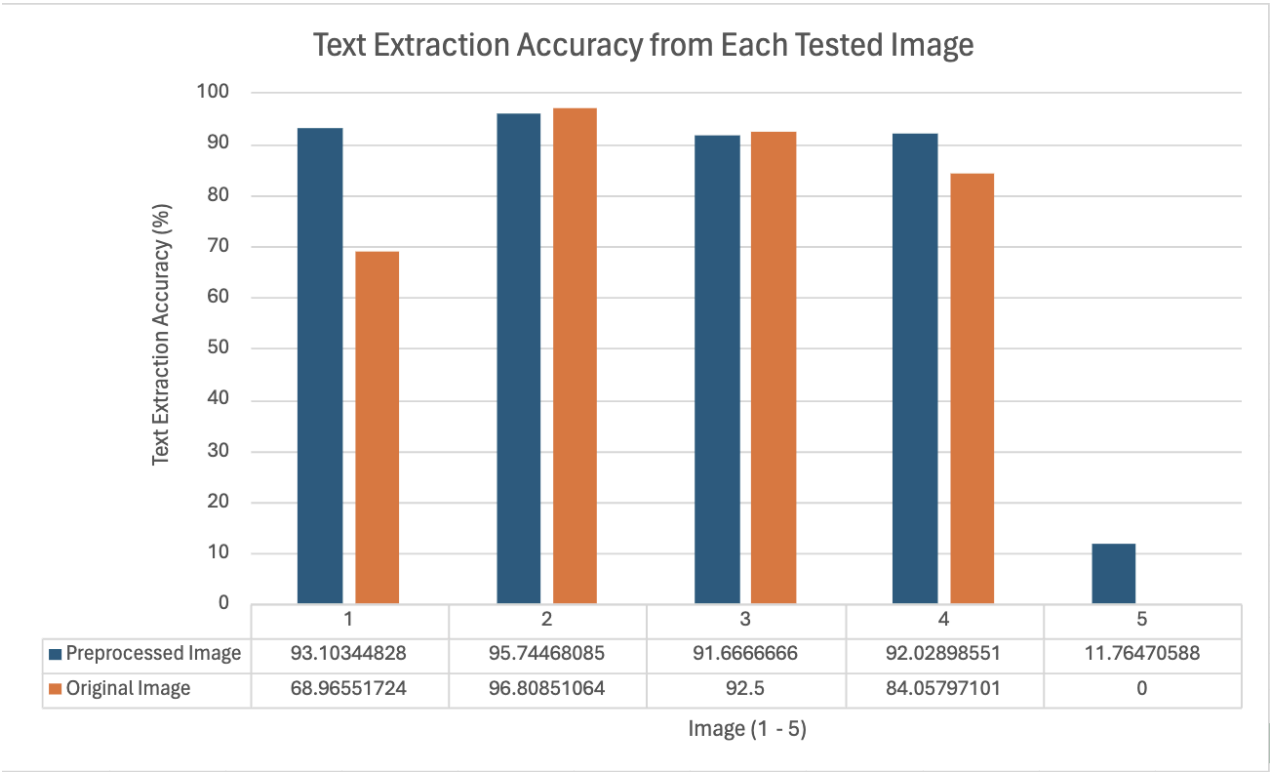


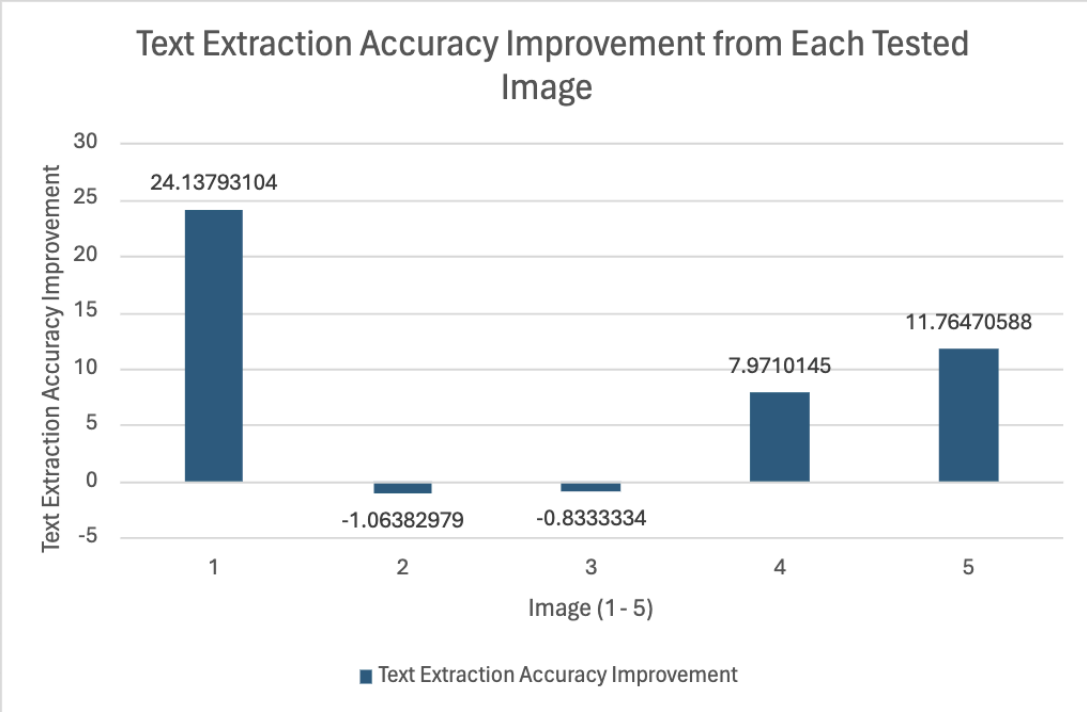


Text extraction accuracy with no preprocessing:



Below is a visual representation of the text extraction accuracies for the original and preprocessed versions of the five tested image samples, along with their accuracy improvements:





Section 5: Discussion of Project Results and What's Next?

After running each noisy image from section 1 into the "Get Best Image" subprogram, it seems that there were quite a few interesting findings. Firstly, the results from section 4 indicated that the subprogram increased the text extraction accuracies of the coffee-stained, crumpled-up, and salt-and-pepper sample images by about 24.1%, 8.0%, and 11.8% respectively. However, the images with folding creases and eraser marks had a drop in text extraction accuracy by approximately 1.1% and 0.8% respectively. A reason for this relationship is that Tesseract was already able to comprehend the image with high confidence under the light noisy conditions of folding creases and eraser marks with no preprocessing necessary. On the other hand, the more substantial background noises, specifically coffee-stains, crumpled-up paper, and salt-and-pepper, resulted in Tesseract having a lower confidence in identifying the exact text in the original image, but did better after preprocessing. Thus, it is fair to say that the lighter the noise, the more likely Tesseract is able to recognize the text without preprocessing, and may perform worse if preprocessing occurs, even if the resulting image is considered to be more user-readable. If instead the noise is more intense, it is plausible that Tesseract will struggle to read the original image, but will almost certainly fare better with a preprocessed binary image from the "Get Best Image" subprogram.

All in all, there is no "one-size-fits-all" solution to image processing, it mostly deals with trial and error. The planning and implementation of this project put that concept to the test so that individual solutions can be found instead of worrying about a general solution. Testing the subprograms that allow the user to provide parameters assisted towards determining those

individual solutions for the “Get Best Image” subprogram, which does its best at solving the given problem of enhancing a noisy text document that consists of one of many different kinds of background noise. It is clear now that this project serves its intended purpose, while also adhering to the beneficial aspects mentioned in section 1, specifically involving text clarity and image preprocessing research. On top of that, the skills obtained from this project may transfer over to other projects of similar nature, like PDF scanning applications. Although the concept of this application seems straightforward, the algorithms used in this project can be appreciated in other fields, like machine learning, medical imaging, and object detection.

Section 6: Project Code

src.filtering.bilateral_filtering.py:

```
import cv2
import numpy as np
import pytesseract
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QSlider,
    QVBoxLayout,
    QComboBox,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 450
max_height = 450


class BilateralFiltering(QWidget):
    """
    A GUI widget for applying bilateral filtering to an image.

    This class provides a GUI for selecting an image and applying bilateral filtering to it.
    Users can adjust the diameter and sigmaColor parameters to customize the filtering. Additionally,
```

the user can view the original grayscale image and a bilateral filtered image, as well as obtain computer-extracted text from either of these images. The user may also save the preprocessed binary image with a specific filename, image format, and directory. The user can also display and save the corresponding histogram graphs of these images as well.

If the user provides a clean version of the image, the user can see the differences between the extracted text of the clean image and any of the other images. On top of that, the feature will also provide a text extraction accuracy value.

The user can also check a box to bold the text through a dilation of size 2x2.

Attributes:
    image (np.ndarray): The currently displayed image.
    clean_image (np.ndarray): The clean version of the image for accuracy calculation.
    titles (list): A list of titles for different image display options.
"""

image: np.ndarray
clean_image: np.ndarray
titles = ["Original Image", "Bilateral Filtering"]

```
def __init__(self):
    """
    Initializes the bilateral filtering window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Bilateral Filtering")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")

    # Set initial parameters
    self.diameter = 5
    self.sigma_color = 75
```

```
# Label that will keep track of the user-inputted diameter
self.diameter_label = QLabel(f"Diameter: {self.diameter}")

# Create a QSlider that the user can interact with to dynamically change the diameter value
self.diameter_slider = QSlider()
self.diameter_slider.setOrientation(Qt.Horizontal)
self.diameter_slider.setTickPosition(QSlider.TicksBelow)
self.diameter_slider.setMinimum(1)
self.diameter_slider.setMaximum(21)
self.diameter_slider.setTickInterval(2)
self.diameter_slider.setValue(self.diameter)

# When the diameter slider's value is changed, update the instance's diameter and the image label
self.diameter_slider.valueChanged.connect(self.on_diameter_change)

# Label that will keep track of the user-inputted sigmaColor
self.sigma_color_label = QLabel(f"Sigma Color: {self.sigma_color}")

# Create a QSlider that the user can interact with to dynamically change the sigmaColor value
self.sigma_color_slider = QSlider()
self.sigma_color_slider.setOrientation(Qt.Horizontal)
self.sigma_color_slider.setTickPosition(QSlider.TicksBelow)
self.sigma_color_slider.setMinimum(1)
self.sigma_color_slider.setMaximum(255)
self.sigma_color_slider.setValue(self.sigma_color)

# When the sigmaColor slider's value is changed, update instance's sigmaColor and the image label
self.sigma_color_slider.valueChanged.connect(self.on_sigma_color_change)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Initialize the image label
self.image = np.tile(np.arange(225, dtype=np.uint8).repeat(2), (450, 1))
q_img = QImage(self.image.data, 450, 450, 450, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
```

```
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
button_layout.addWidget(provide_clean_image_btn)
button_layout.addWidget(show_histogram_btn)
button_layout.addWidget(provide_help_btn)

preprocessing_layout = QHBoxLayout()
preprocessing_layout.addWidget(self.method_combobox)
preprocessing_layout.addWidget(self.dilation_request)

image_layout = QHBoxLayout()
image_layout.addWidget(self.image_label)
image_layout.addWidget(self.clean_image_label)

# Create layout and add widgets
layout = QVBoxLayout()
layout.addWidget(open_image_btn)
layout.addLayout(preprocessing_layout)
layout.addWidget(self.noisy_label)
layout.addWidget(self.clean_label)
layout.addWidget(self.diameter_label)
layout.addWidget(self.diameter_slider)
layout.addWidget(self.sigma_color_label)
layout.addWidget(self.sigma_color_slider)
layout.addLayout(image_layout)
layout.addLayout(button_layout)
layout.addWidget(save_btn)

# Set dialog layout
self.setLayout(layout)
```

```
def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Apply bilateral filtering with the selected image, diameter, sigma color, and sigma space
    if method_idx == 1:
        # Choice 1: Bilateral Filtering
        image = cv2.bilateralFilter(self.image, self.diameter, self.sigma_color, 200)
    else:
        # Choice 0: Original Image
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    if self.compressed_img is not None:
        compressed_h, compressed_w = self.compressed_img.shape
        image = cv2.resize(image, (compressed_w, compressed_h))
```

```
    # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
    image_h, image_w = image.shape
    q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
    self.image_label.setPixmap(QPixmap.fromImage(q_img))

    # Update the histogram if the window is displayed
    if self.histogram_window is not None:
        self.histogram_window.update_histogram(image)
        self.histogram_window.show()

def on_diameter_change(self, diameter):
    """
    Sets the diameter.
    """

    self.diameter = diameter
    self.diameter_label.setText(f"Diameter: {self.diameter}")
    self.update_image()

def on_sigma_color_change(self, sigma_color):
    """
    Sets the sigmaColor.
    """

    self.sigma_color = sigma_color
    self.sigma_color_label.setText(f"Sigma Color: {self.sigma_color}")
    self.update_image()

def extract_text(self):
    """
    Run Tesseract OCR using the user-selected image or the same image after going through bilateral filtering
    based on the current diameter, sigma color, and sigma space to extract and display the text on the image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    if method_idx == 1:
        image = cv2.bilateralFilter(self.image, self.diameter, self.sigma_color, 200)
    else:
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    # Run Tesseract OCR on the image
    text = pytesseract.image_to_string(image)

    # Display the extracted text
```

```
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        image = cv2.bilateralFilter(self.image, self.diameter, self.sigma_color, 200)
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """
```

```
        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Bilateral Filtering" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Bilateral Filtering:

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose "Bilateral Filtering" from the drop-down menu. You will then see the resulting binary
image.
                Step 3: Play with the "Diameter" and "Sigma Color" sliders to update the binary image.
                Step 4: Click on "Extract Text" to perform text extraction.
                Step 5: Save the preprocessed binary image if needed.

                Additional Features:

                If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract
Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction
accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted
text from the clean image.

                Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

                You can also bold the text be clicking the "Bold Text" checkbox.

                BIG Note:

                Your image will be compressed in the application if it is over 450x450. However, saving the binary image
will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy
calculations, and histograms.
                """

        QMessageBox.information(self, "Help", help_text)
```

src.filtering.gaussian_blur.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
```

```
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QSlider,
    QVBoxLayout,
    QComboBox,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.math import round_up_to_odd
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 512
max_height = 512

class GaussianBlur(QWidget):
    """
    A GUI widget for applying Gaussian blur to an image.

    This class provides a GUI for selecting an image and applying Gaussian blur to it.
    The user is able to manually indicate a kernel size with a slider. Additionally, the user can view the original
    grayscale image and a Gaussian blurred image, as well as obtain computer-extracted text from either of these
images.
    The user may also save the preprocessed binary image with
    a specific filename, image format, and directory. The user can also display and save the corresponding histogram
    graphs of these images as well.

    If the user provides a clean version of the image, the user can see the differences between the extracted
    text of the clean image and any of the other images. On top of that, the feature will also provide a text
    extraction accuracy value.

    The user can also check a box to bold the text through a dilation of size 2x2.

    Attributes:
        image (np.ndarray): The currently displayed image.
        clean_image (np.ndarray): The clean version of the image for accuracy calculation.
        titles (list): A list of titles for different image display options.
    """

    image: np.ndarray
    clean_image: np.ndarray
```

```
titles = ["Original Image", "Gaussian Blur"]

def __init__(self):
    """
    Initializes the Gaussian blur window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Gaussian Blur Filtering")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")

    # Set an initial kernel size
    self.kernel_size = 3

    # Label that will keep track of the user-inputted kernel size
    self.kernel_size_label = QLabel(f"Kernel Size: {self.kernel_size}")

    # Create a QSlider that the user can interact with to dynamically change the kernel size value
    self.kernel_size_slider = QSlider()
    self.kernel_size_slider.setOrientation(Qt.Horizontal)
    self.kernel_size_slider.setTickPosition(QSlider.TicksBelow)
    self.kernel_size_slider.setMinimum(3)
    self.kernel_size_slider.setMaximum(25)
    self.kernel_size_slider.setTickInterval(2)
    self.kernel_size_slider.setSingleStep(2)
    self.kernel_size_slider.setValue(self.kernel_size)

    # When the kernel size slider's value is changed, update the instance's kernel size and the image label
    self.kernel_size_slider.valueChanged.connect(self.update_image)

    # Label that will hold the desired image
    self.image_label = QLabel()
    self.clean_image_label = QLabel()
```

```
# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(256, dtype=np.uint8).repeat(2), (512, 1))
q_img = QImage(self.image.data, 512, 512, 512, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
button_layout.addWidget(provide_clean_image_btn)
button_layout.addWidget(show_histogram_btn)
button_layout.addWidget(provide_help_btn)

preprocessing_layout = QHBoxLayout()
preprocessing_layout.addWidget(self.method_combobox)
preprocessing_layout.addWidget(self.dilation_request)

image_layout = QHBoxLayout()
image_layout.addWidget(self.image_label)
image_layout.addWidget(self.clean_image_label)

# Create layout and add widgets
```

```
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.kernel_size_label)
    layout.addWidget(self.kernel_size_slider)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Get the new kernel size from the slider
```

```
    self.kernel_size = round_up_to_odd(self.kernel_size_slider.value())

    # Display the new kernel size value
    self.kernel_size_label.setText(f"Kernel Size: {self.kernel_size}")

    # Apply Gaussian blur to the image with the selected kernel size
    if method_idx == 1:
        # Choice 1: Gaussian Blur
        image = cv2.GaussianBlur(self.image, (self.kernel_size, self.kernel_size), 0)
    else:
        # Choice 0: Original Image
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    if self.compressed_img is not None:
        compressed_h, compressed_w = self.compressed_img.shape
        image = cv2.resize(image, (compressed_w, compressed_h))

    # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
    image_h, image_w = image.shape
    q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
    self.image_label.setPixmap(QPixmap.fromImage(q_img))

    # Update the histogram if the window is displayed
    if self.histogram_window is not None:
        self.histogram_window.update_histogram(image)
        self.histogram_window.show()

def extract_text(self):
    """
    Run Tesseract OCR using the user-selected image or the same image after going through Gaussian blurring
    based on the current kernel size to extract and display the text on the image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    self.kernel_size = round_up_to_odd(self.kernel_size_slider.value())
    self.kernel_size_label.setText(f"Kernel Size: {self.kernel_size}")

    if method_idx == 1:
        image = cv2.GaussianBlur(self.image, (self.kernel_size, self.kernel_size), 0)
    else:
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)
```

```
        # Run Tesseract OCR on the filtered image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        image = cv2.GaussianBlur(self.image, (self.kernel_size, self.kernel_size), 0)
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)
```

```
    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Gaussian Blur Filtering" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Gaussian Blur Filtering:

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose "Gaussian Blur" from the drop-down menu. You will then see the resulting binary image.
                Step 3: Play with the "Kernel Size" slider to update the binary image.
                Step 4: Click on "Extract Text" to perform text extraction.
                Step 5: Save the preprocessed binary image if needed.

                Additional Features:

            If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

                Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

                You can also bold the text be clicking the "Bold Text" checkbox.

                BIG Note:

            Your image will be compressed in the application if it is over 512x512. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
                """

        QMessageBox.information(self, "Help", help_text)
```

src.filtering.median_filtering.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QSlider,
    QVBoxLayout,
    QComboBox,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.math import round_up_to_odd
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 512
max_height = 512


class MedianFiltering(QWidget):
    """
    A GUI widget for applying median filtering to an image.

    This class provides a GUI for selecting an image and applying median filtering to it.
    The user is able to manually indicate a kernel size with a slider. Additionally, the user can view the original
    grayscale image and a median filtered image, as well as obtain computer-extracted text from either of these
images.
    The user may also save the preprocessed binary image with
    a specific filename, image format, and directory. The user can also display and save the corresponding histogram
    graphs of these images as well.

    If the user provides a clean version of the image, the user can see the differences between the extracted
    text of the clean image and any of the other images. On top of that, the feature will also provide a text
    extraction accuracy value.

    The user can also check a box to bold the text through a dilation of size 2x2.

    Attributes:
        image (np.ndarray): The currently displayed image.
        clean_image (np.ndarray): The clean version of the image for accuracy calculation.
```

```
    titles (list): A list of titles for different image display options.
"""

image: np.ndarray
clean_image: np.ndarray
titles = ["Original Image", "Median Filtering"]

def __init__(self):
    """
    Initializes the median filtering window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Median Filtering")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")

    # Set an initial kernel size
    self.kernel_size = 3

    # Label that will keep track of the user-inputted kernel size
    self.kernel_size_label = QLabel(f"Kernel Size: {self.kernel_size}")

    # Create a QSlider that the user can interact with to dynamically change the kernel size value to an odd integer
    # within the range of 3 and 21
    self.kernel_size_slider = QSlider()
    self.kernel_size_slider.setOrientation(Qt.Horizontal)
    self.kernel_size_slider.setTickPosition(QSlider.TicksBelow)
    self.kernel_size_slider.setMinimum(3)
    self.kernel_size_slider.setMaximum(21)
    self.kernel_size_slider.setTickInterval(2)
    self.kernel_size_slider.setSingleStep(2)
    self.kernel_size_slider.setValue(self.kernel_size)

    # When the kernel size slider's value is changed, update the instance's kernel size and the image label
```

```
self.kernel_size_slider.valueChanged.connect(self.on_kernel_size_change)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(256, dtype=np.uint8).repeat(2), (512, 1))
q_img = QImage(self.image.data, 512, 512, 512, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
button_layout.addWidget(provide_clean_image_btn)
button_layout.addWidget(show_histogram_btn)
button_layout.addWidget(provide_help_btn)

preprocessing_layout = QHBoxLayout()
preprocessing_layout.addWidget(self.method_combobox)
preprocessing_layout.addWidget(self.dilation_request)
```

```
    image_layout = QHBoxLayout()
    image_layout.addWidget(self.image_label)
    image_layout.addWidget(self.clean_image_label)

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.kernel_size_label)
    layout.addWidget(self.kernel_size_slider)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
```

```
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        # Apply median filtering to the image with the selected kernel size
        if method_idx == 1:
            # Choice 1: Median Filtering
            image = cv2.medianBlur(self.image, self.kernel_size)
        else:
            # Choice 0: Original Image
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        if self.compressed_img is not None:
            compressed_h, compressed_w = self.compressed_img.shape
            image = cv2.resize(image, (compressed_w, compressed_h))

        # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
        image_h, image_w = image.shape
        q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
        self.image_label.setPixmap(QPixmap.fromImage(q_img))

        # Update the histogram if the window is displayed
        if self.histogram_window is not None:
            self.histogram_window.update_histogram(image)
            self.histogram_window.show()

    def on_kernel_size_change(self):
        """
        Sets the kernel size and updates the image.
        """

        self.kernel_size = round_up_to_odd(self.kernel_size_slider.value())
        self.kernel_size_label.setText(f"Kernel Size: {self.kernel_size}")
        self.update_image()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image or the same image after going through median filtering
        based on the current kernel size to extract and display the text on the image.
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        if method_idx == 1:
```

```
            image = cv2.medianBlur(self.image, self.kernel_size)
        else:
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        # Run Tesseract OCR on the image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
```

```
        """

        image = cv2.medianBlur(self.image, self.kernel_size)
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Median Filtering" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Median Filtering:

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose "Median Filtering" from the drop-down menu. You will then see the resulting binary image.
                Step 3: Play with the "Kernel Size" slider to update the binary image.
                Step 4: Click on "Extract Text" to perform text extraction.
                Step 5: Save the preprocessed binary image if needed.

                Additional Features:

                If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

                Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

                You can also bold the text be clicking the "Bold Text" checkbox.

                BIG Note:

                Your image will be compressed in the application if it is over 512x512. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
```

```
            """

        QMessageBox.information(self, "Help", help_text)
```

src.morphology.closing_opening.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QSlider,
    QVBoxLayout,
    QComboBox,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 450
max_height = 450


class ClosingOpening(QWidget):
    """
    A GUI widget for applying a closing-opening morphological operation to an image.

    This class provides a GUI for selecting an image and applying a closing-opening morphological operation to it.
    Users can adjust the kernel size and the number of iterations to customize the operation.
    Additionally, the user can view the original grayscale image and a closing-opening image, as well as obtain
    computer-extracted text from either of these images. The user may also save the preprocessed binary image with
    a specific filename, image format, and directory. The user can also display and save the corresponding histogram
    graphs of these images as well.

    If the user provides a clean version of the image, the user can see the differences between the extracted
    text of the clean image and any of the other images. On top of that, the feature will also provide a text
    extraction accuracy value.
```

The user can also check a box to bold the text through a dilation of size 2x2.

Attributes:
    image (np.ndarray): The currently displayed image.
    clean_image (np.ndarray): The clean version of the image for accuracy calculation.
    titles (list): A list of titles for different image display options.
"""

image: np.ndarray
clean_image: np.ndarray
titles = ["Original Image", "Closing-Opening"]

def __init__(self):
    """
    Initializes the closing-opening window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Closing-Opening Morphology")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")

    # Set initial kernel size and iterations
    self.kernel_size = 2
    self.iterations = 1

    # Label that will keep track of the user-inputted kernel size
    self.kernel_size_label = QLabel(f"Kernel Size: {self.kernel_size}")

    # Create a QSlider that the user can interact with to dynamically change the kernel size value
    self.kernel_size_slider = QSlider()
    self.kernel_size_slider.setOrientation(Qt.Horizontal)
    self.kernel_size_slider.setTickPosition(QSlider.TicksBelow)
    self.kernel_size_slider.setMinimum(1)
    self.kernel_size_slider.setMaximum(21)

```
self.kernel_size_slider.setTickInterval(2)
self.kernel_size_slider.setValue(self.kernel_size)

# When the kernel size slider's value is changed, update the instance's kernel size and the image label
self.kernel_size_slider.valueChanged.connect(self.on_kernel_size_change)

# Label that will keep track of the user-inputted number of iterations
self.iterations_label = QLabel(f"Iterations: {self.iterations}")

# Create a QSlider that the user can interact with to dynamically change the number of iterations value
self.iterations_slider = QSlider()
self.iterations_slider.setOrientation(Qt.Horizontal)
self.iterations_slider.setTickPosition(QSlider.TicksBelow)
self.iterations_slider.setMinimum(1)
self.iterations_slider.setMaximum(10)
self.iterations_slider.setValue(self.iterations)

# When the iterations slider's value is changed, update the instance's iterations and the image label
self.iterations_slider.valueChanged.connect(self.on_iterations_change)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(225, dtype=np.uint8).repeat(2), (450, 1))
q_img = QImage(self.image.data, 450, 450, 450, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
```

```
    save_btn.clicked.connect(self.prepare_to_save)

    # Prepare a button to show the histogram
    show_histogram_btn = QPushButton("Show/Hide Histogram")
    show_histogram_btn.clicked.connect(self.show_histogram)

    provide_help_btn = QPushButton("Help", self)
    provide_help_btn.clicked.connect(self.provide_help)

    button_layout = QHBoxLayout()
    button_layout.addWidget(extract_text_btn)
    button_layout.addWidget(provide_clean_image_btn)
    button_layout.addWidget(show_histogram_btn)
    button_layout.addWidget(provide_help_btn)

    preprocessing_layout = QHBoxLayout()
    preprocessing_layout.addWidget(self.method_combobox)
    preprocessing_layout.addWidget(self.dilation_request)

    image_layout = QHBoxLayout()
    image_layout.addWidget(self.image_label)
    image_layout.addWidget(self.clean_image_label)

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.kernel_size_label)
    layout.addWidget(self.kernel_size_slider)
    layout.addWidget(self.iterations_label)
    layout.addWidget(self.iterations_slider)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
```

```
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Create a kernel for the morphological operations
    kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

    # Apply closing-opening morphology with the current image, kernel size, and number of iterations
    if method_idx == 1:
        # Choice 1: Closing-Opening

        image = 255 - self.image  # Invert the image

        # Apply closing morphological operation
        closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)

        # Apply opening morphological operation
        image = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

        image = 255 - image
    else:
        # Choice 0: Original Image
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    if self.compressed_img is not None:
```

```
            compressed_h, compressed_w = self.compressed_img.shape
            image = cv2.resize(image, (compressed_w, compressed_h))

        # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
        image_h, image_w = image.shape
        q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
        self.image_label.setPixmap(QPixmap.fromImage(q_img))

        # Update the histogram if the window is displayed
        if self.histogram_window is not None:
            self.histogram_window.update_histogram(image)
            self.histogram_window.show()

    def on_kernel_size_change(self, kernel_size):
        """
        Sets the kernel size.
        """

        self.kernel_size = kernel_size
        self.kernel_size_label.setText(f"Kernel Size: {self.kernel_size}")
        self.update_image()

    def on_iterations_change(self, iterations):
        """
        Sets the number of iterations.
        """

        self.iterations = iterations
        self.iterations_label.setText(f"Iterations: {self.iterations}")
        self.update_image()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image or the same image after going through closing-opening
        morphology based on the current kernel size and number of iterations to extract and display the text on the
        image.
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

        if method_idx == 1:
            image = 255 - self.image  # Invert the image

            # Apply closing morphological operation
            closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)
```

```
            # Apply opening morphological operation
            image = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

            image = 255 - image
        else:
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        # Run Tesseract OCR on the image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")
```

```
    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

        image = 255 - self.image  # Invert the image

        # Apply closing morphological operation
        closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)

        # Apply opening morphological operation
        image = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

        image = 255 - image

        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Closing-Opening Morphology" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Closing-Opening Morphology:

            Step 1: Pass in your noisy image with "Open Image."
            Step 2: Choose "Closing-Opening" from the drop-down menu. You will then see the resulting binary
image.
            Step 3: Play with the "Kernel Size" and "Iterations" sliders to update the binary image.
            Step 4: Click on "Extract Text" to perform text extraction.
            Step 5: Save the preprocessed binary image if needed.

            Additional Features:
```

If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

You can also bold the text be clicking the "Bold Text" checkbox.

BIG Note:

Your image will be compressed in the application if it is over 450x450. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
"""

QMessageBox.information(self, "Help", help_text)

src.morphology.opening_closing.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QSlider,
    QVBoxLayout,
    QComboBox,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 450
max_height = 450


class OpeningClosing(QWidget):
```

"""
A GUI widget for applying an opening-closing morphological operation to an image.

This class provides a GUI for selecting an image and applying an opening-closing morphological operation to it. Users can adjust the kernel size and the number of iterations to customize the operation.
Additionally, the user can view the original grayscale image and an opening-closing image, as well as obtain computer-extracted text from either of these images. The user may also save the preprocessed binary image with a specific filename, image format, and directory. The user can also display and save the corresponding histogram graphs of these images as well.

If the user provides a clean version of the image, the user can see the differences between the extracted text of the clean image and any of the other images. On top of that, the feature will also provide a text extraction accuracy value.

The user can also check a box to bold the text through a dilation of size 2x2.

Attributes:
    image (np.ndarray): The currently displayed image.
    clean_image (np.ndarray): The clean version of the image for accuracy calculation.
    titles (list): A list of titles for different image display options.
"""

image: np.ndarray
clean_image: np.ndarray
titles = ["Original Image", "Opening-Closing"]

```
def __init__(self):
    """
    Initializes the opening-closing window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Opening-Closing Morphology")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")
```

```
# Set initial kernel size and iterations
self.kernel_size = 2
self.iterations = 1

# Label that will keep track of the user-inputted kernel size
self.kernel_size_label = QLabel(f"Kernel Size: {self.kernel_size}")

# Create a QSlider that the user can interact with to dynamically change the kernel size value
self.kernel_size_slider = QSlider()
self.kernel_size_slider.setOrientation(Qt.Horizontal)
self.kernel_size_slider.setTickPosition(QSlider.TicksBelow)
self.kernel_size_slider.setMinimum(1)
self.kernel_size_slider.setMaximum(21)
self.kernel_size_slider.setTickInterval(2)
self.kernel_size_slider.setValue(self.kernel_size)

# When the kernel size slider's value is changed, update the instance's kernel size and the image label
self.kernel_size_slider.valueChanged.connect(self.on_kernel_size_change)

# Label that will keep track of the user-inputted number of iterations
self.iterations_label = QLabel(f"Iterations: {self.iterations}")

# Create a QSlider that the user can interact with to dynamically change the number of iterations value
self.iterations_slider = QSlider()
self.iterations_slider.setOrientation(Qt.Horizontal)
self.iterations_slider.setTickPosition(QSlider.TicksBelow)
self.iterations_slider.setMinimum(1)
self.iterations_slider.setMaximum(10)
self.iterations_slider.setValue(self.iterations)

# When the iterations slider's value is changed, update instance's iterations and the image label
self.iterations_slider.valueChanged.connect(self.on_iterations_change)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(225, dtype=np.uint8).repeat(2), (450, 1))
q_img = QImage(self.image.data, 450, 450, 450, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None
```

```
# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
button_layout.addWidget(provide_clean_image_btn)
button_layout.addWidget(show_histogram_btn)
button_layout.addWidget(provide_help_btn)

preprocessing_layout = QHBoxLayout()
preprocessing_layout.addWidget(self.method_combobox)
preprocessing_layout.addWidget(self.dilation_request)

image_layout = QHBoxLayout()
image_layout.addWidget(self.image_label)
image_layout.addWidget(self.clean_image_label)

# Create layout and add widgets
layout = QVBoxLayout()
layout.addWidget(open_image_btn)
layout.addLayout(preprocessing_layout)
layout.addWidget(self.noisy_label)
layout.addWidget(self.clean_label)
layout.addWidget(self.kernel_size_label)
layout.addWidget(self.kernel_size_slider)
layout.addWidget(self.iterations_label)
layout.addWidget(self.iterations_slider)
layout.addLayout(image_layout)
layout.addLayout(button_layout)
```

```
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Create a kernel for the morphological operations
    kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

    # Apply opening-closing morphology with the current image, kernel size, and number of iterations
    if method_idx == 1:
        # Choice 1: Opening-Closing

        image = 255 - self.image  # Invert the image

        # Apply opening morphological operation
```

```
            opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

            # Apply closing morphological operation
            image = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)

            image = 255 - image
        else:
            # Choice 0: Original Image
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        if self.compressed_img is not None:
            compressed_h, compressed_w = self.compressed_img.shape
            image = cv2.resize(image, (compressed_w, compressed_h))

        # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
        image_h, image_w = image.shape
        q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
        self.image_label.setPixmap(QPixmap.fromImage(q_img))

        # Update the histogram if the window is displayed
        if self.histogram_window is not None:
            self.histogram_window.update_histogram(image)
            self.histogram_window.show()

    def on_kernel_size_change(self, kernel_size):
        """
        Sets the kernel size.
        """

        self.kernel_size = kernel_size
        self.kernel_size_label.setText(f"Kernel Size: {self.kernel_size}")
        self.update_image()

    def on_iterations_change(self, iterations):
        """
        Sets the number of iterations.
        """

        self.iterations = iterations
        self.iterations_label.setText(f"Iterations: {self.iterations}")
        self.update_image()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image or the same image after going through opening-closing
        morphology based on the current kernel size and number of iterations to extract and display the text on the
```

```
    image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

    if method_idx == 1:
        image = 255 - self.image  # Invert the image

        # Apply opening morphological operation
        opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

        # Apply closing morphological operation
        image = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)

        image = 255 - image
    else:
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    # Run Tesseract OCR on the image
    text = pytesseract.image_to_string(image)

    # Display the extracted text
    QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

    # Calculate text extraction accuracy if a clean image is provided
    if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
        clean_text = pytesseract.image_to_string(self.clean_image)
        accuracy = calculate_accuracy(text, clean_text)
        QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
        QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

def provide_clean_image(self):
    """
    Allows the user to provide a clean version of the image for accuracy calculation.
    """

    clean_image, clean_file_name = open_image_dialog()
    if clean_image is not None:
        self.clean_image = np.array(clean_image)
        self.clean_label.setText("Clean File: " + clean_file_name)
        if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
            clean_image = cv2.resize(clean_image, (max_width, max_height))
        elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
```

```
            clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
        elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
            clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

        clean_image_h, clean_image_w = clean_image.shape
        q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
        self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

        QMessageBox.information(self, "Success",
                                "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

def prepare_to_save(self):
    """
    Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
    """

    kernel = np.ones((self.kernel_size, self.kernel_size), np.uint8)

    image = 255 - self.image  # Invert the image

    # Apply opening morphological operation
    opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel, iterations=self.iterations)

    # Apply closing morphological operation
    image = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel, iterations=self.iterations)

    image = 255 - image

    if self.dilation_request.isChecked():
        image = dilate_image(image)
    save_image(self, image)

def show_histogram(self):
    """
    Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
    """

    # Create the histogram window if it doesn't exist, otherwise remove it
    if self.histogram_window is None:
        self.histogram_window = HistogramWindow(self.image)
    else:
        self.histogram_window = None
    self.update_image()

def provide_help(self):
    """
```

Display help information for the "Opening-Closing Morphology" program.
"""

help_text = """How to Enhance and Extract Text from an Image with Opening-Closing Morphology:

Step 1: Pass in your noisy image with "Open Image."
Step 2: Choose "Opening-Closing" from the drop-down menu. You will then see the resulting binary image.
Step 3: Play with the "Kernel Size" and "Iterations" sliders to update the binary image.
Step 4: Click on "Extract Text" to perform text extraction.
Step 5: Save the preprocessed binary image if needed.

Additional Features:

If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

You can also bold the text be clicking the "Bold Text" checkbox.

BIG Note:

Your image will be compressed in the application if it is over 450x450. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
"""

QMessageBox.information(self, "Help", help_text)

src.thresholding.adaptive_thresholding.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QComboBox,
    QLabel,
    QSlider,
    QVBoxLayout,
    QMessageBox,
    QHBoxLayout,
```

```
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.math import round_up_to_odd
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 450
max_height = 450


class AdaptiveThresholding(QWidget):
    """
    A GUI widget for applying adaptive thresholding to an image.

    This class provides a GUI for selecting an image and applying adaptive thresholding to it. The user is able to
    manually indicate a block size and a C constant with sliders. Users can view the original grayscale image and
    two binary images that use adaptive thresholding
    (cv2.ADAPTIVE_THRESH_MEAN_C & cv2.ADAPTIVE_THRESH_GAUSSIAN_C), as well as obtain
computer-extracted text from any of
    these images. The user may also save the preprocessed binary image with a specific filename, image format, and
    directory. The user can also display and save the corresponding histogram graphs of these images as well.

    If the user provides a clean version of the image, the user can see the differences between the extracted
    text of the clean image and any of the other images. On top of that, the feature will also provide a text
    extraction accuracy value.

    The user can also check a box to bold the text through a dilation of size 2x2.

    Attributes:
        image (np.ndarray): The currently displayed image.
        clean_image (np.ndarray): The clean version of the image for accuracy calculation.
        titles (list): A list of titles for different image display options.
    """

    image: np.ndarray
    clean_image: np.ndarray
    titles = ["Original Image", "ADAPTIVE_THRESH_MEAN_C", "ADAPTIVE_THRESH_GAUSSIAN_C"]

    def __init__(self):
        """
        Initializes the adaptive thresholding window of the application.
        """

        # Initialize the parent class (QWidget)
        super().__init__()
```

```
self.setWindowTitle("Adaptive Thresholding")

# Go through the titles of all the adaptive thresholding methods and
# allow the user to see them and choose one through a combo box.
self.method_combobox = QComboBox()
for title in self.titles:
    self.method_combobox.addItem(title)
self.method_combobox.currentIndexChanged.connect(self.update_image)

# Allow the user to check a box to "bold the text" by performing dilation
self.dilation_request = QCheckBox("Bold Text")
self.dilation_request.stateChanged.connect(self.update_image)

# Have labels to keep track of the given noisy and clean files
self.noisy_label = QLabel("Noisy File: N/A")
self.clean_label = QLabel("Clean File: N/A")

# Set an initial block size
self.block_size = 11

# Label that will keep track of the user-inputted block size
self.block_size_label = QLabel(f"Block Size: {self.block_size}")

# Create a QSlider that the user can interact with to dynamically change the block size value to an odd integer
# within the range of 3 and 255
self.block_size_slider = QSlider()
self.block_size_slider.setOrientation(Qt.Horizontal)
self.block_size_slider.setTickPosition(QSlider.TicksBelow)
self.block_size_slider.setMinimum(3)
self.block_size_slider.setMaximum(255)
self.block_size_slider.setTickInterval(10)
self.block_size_slider.setSingleStep(2)
self.block_size_slider.setValue(self.block_size)

# When the block size slider's value is changed, update the instance's block size and the image label
self.block_size_slider.valueChanged.connect(self.on_block_size_change)

# Set an initial C constant
self.c_constant = 0

# Label that will keep track of the user-inputted C constant
self.c_constant_label = QLabel(f"C Constant: {self.c_constant}")

# Create a QSlider that the user can interact with to dynamically change the C constant value
# within the range of 0 to 100
self.c_constant_slider = QSlider()
self.c_constant_slider.setOrientation(Qt.Horizontal)
self.c_constant_slider.setTickPosition(QSlider.TicksBelow)
```

```
self.c_constant_slider.setMinimum(0)
self.c_constant_slider.setMaximum(100)
self.c_constant_slider.setValue(self.c_constant)

# When the C constant slider's value is changed, update the instance's C constant and the image label
self.c_constant_slider.valueChanged.connect(self.on_c_constant_change)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(225, dtype=np.uint8).repeat(2), (450, 1))
q_img = QImage(self.image.data, 450, 450, 450, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

# Prepare a button that when pushed will provide help regarding adaptive thresholding and the features of this
# program
provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
```

```
    button_layout.addWidget(provide_clean_image_btn)
    button_layout.addWidget(show_histogram_btn)
    button_layout.addWidget(provide_help_btn)

    preprocessing_layout = QHBoxLayout()
    preprocessing_layout.addWidget(self.method_combobox)
    preprocessing_layout.addWidget(self.dilation_request)

    image_layout = QHBoxLayout()
    image_layout.addWidget(self.image_label)
    image_layout.addWidget(self.clean_image_label)

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.block_size_label)
    layout.addWidget(self.block_size_slider)
    layout.addWidget(self.c_constant_label)
    layout.addWidget(self.c_constant_slider)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
```

```
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Display the new block size value
    self.block_size_label.setText(f"Block Size: {self.block_size}")

    # Using the grayscale image, perform the requested adaptive thresholding method
    # with the user-selected block size and C constant values
    if method_idx == 1:
        # Choice 1: cv2.ADAPTIVE_THRESH_MEAN_C
        image = cv2.adaptiveThreshold(self.image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY,
                                      self.block_size, self.c_constant,)
    elif method_idx == 2:
        # Choice 2: cv2.ADAPTIVE_THRESH_GAUSSIAN_C
        image = cv2.adaptiveThreshold(self.image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY,
                                      self.block_size, self.c_constant,)
    else:
        # Choice 0: Original Image
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    if self.compressed_img is not None:
        compressed_h, compressed_w = self.compressed_img.shape
        image = cv2.resize(image, (compressed_w, compressed_h))

    # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
    image_h, image_w = image.shape
    q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
    self.image_label.setPixmap(QPixmap.fromImage(q_img))

    # Update the histogram if the window is displayed
    if self.histogram_window is not None:
        self.histogram_window.update_histogram(image)
        self.histogram_window.show()
```

```
    def on_block_size_change(self):
        """
        Sets the block size.
        """

        self.block_size = round_up_to_odd(self.block_size_slider.value())
        self.block_size_label.setText(f"Block Size: {self.block_size}")
        self.update_image()

    def on_c_constant_change(self):
        """
        Sets the C constant.
        """

        self.c_constant = self.c_constant_slider.value()
        self.c_constant_label.setText(f"C Constant: {self.c_constant}")
        self.update_image()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image, adaptive thresholding method, block size value,
        and C constant value to extract and display the text on the image.
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        if method_idx == 1:
            image = cv2.adaptiveThreshold(
                self.image, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, self.block_size,
self.c_constant,
            )
        elif method_idx == 2:
            image = cv2.adaptiveThreshold(
                self.image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, self.block_size,
self.c_constant,
            )
        else:
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        # Run Tesseract OCR on the image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)
```

```
        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        method_idx = self.method_combobox.currentIndex()

        if method_idx == 1:
            image = cv2.adaptiveThreshold(
                self.image, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, self.block_size,
self.c_constant,
            )
        elif method_idx == 2:
            image = cv2.adaptiveThreshold(
```

```
            self.image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, self.block_size,
self.c_constant,
            )
        else:
            QMessageBox.warning(self, "No Box Selected", "Please select an adaptive thresholding method before "
                                                         "requesting to save a binary image.")
            return
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Get Best Image" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Adaptive Thresholding:

            Step 1: Pass in your noisy image with "Open Image."
            Step 2: Choose one of two adaptive thresholding methods from the drop-down menu. You will then see the
resulting binary image.
            Step 3: Play with the "Block Size" and "C Constant" sliders to update the binary image.
            Step 4: Click on "Extract Text" to perform text extraction.
            Step 5: Save the preprocessed binary image if needed.

            Additional Features:

            If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract
Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction
accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted
text from the clean image.

            Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

            You can also bold the text be clicking the "Bold Text" checkbox.

            BIG Note:
```

Your image will be compressed in the application if it is over 450x450. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
"""

QMessageBox.information(self, "Help", help_text)

src.thresholding.binary_thresholding.py:

```
import cv2
import pytesseract
import numpy as np
from PySide6.QtCore import Qt
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QComboBox,
    QLabel,
    QSlider,
    QVBoxLayout,
    QMessageBox,
    QHBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 512
max_height = 512


class BinaryThresholding(QWidget):
    """
    A GUI widget for applying binary thresholding to an image.

    This class provides a GUI for selecting an image and applying binary thresholding to it.
    Users can view the original grayscale image and several binary images that use binary thresholding, as well as
    obtain computer-extracted text from any of these images. The user may also save the preprocessed binary image with
    a specific filename, image format, and directory. The user can also display and save the corresponding histogram
    graphs of these images as well.
```

If the user provides a clean version of the image, the user can see the differences between the extracted text of the clean image and any of the other images. On top of that, the feature will also provide a text extraction accuracy value.

The user can also check a box to bold the text through a dilation of size 2x2.

```
Attributes:
    image (np.ndarray): The currently displayed image.
    clean_image (np.ndarray): The clean version of the image for accuracy calculation.
    titles (list): A list of titles for different image display options.
"""

image: np.ndarray
clean_image: np.ndarray
titles = [
    "Original Image",
    "THRESH_BINARY",
    "THRESH_BINARY_INV",
    "THRESH_TRUNC",
    "THRESH_TOZERO",
    "THRESH_TOZERO_INV",
]

def __init__(self):
    """
    Initializes the binary thresholding window of the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Binary Thresholding")

    # Go through the titles of all the binary thresholding methods and
    # allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")

    # Label that will keep track of the user-inputted threshold
```

```
self.threshold_label = QLabel("Threshold Value: 127")

# Create a QSlider that the user can interact with to dynamically change the threshold value
# within the range of 0 to 255
self.threshold_slider = QSlider()
self.threshold_slider.setOrientation(Qt.Horizontal)
self.threshold_slider.setTickPosition(QSlider.TicksBelow)
self.threshold_slider.setTickInterval(10)
self.threshold_slider.setMinimum(0)
self.threshold_slider.setMaximum(255)
self.threshold_slider.setValue(127)

# When the threshold slider's value is changed, update the image label
self.threshold_slider.valueChanged.connect(self.update_image)

# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(256, dtype=np.uint8).repeat(2), (512, 1))
q_img = QImage(self.image.data, 512, 512, 512, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)
```

```
    provide_help_btn = QPushButton("Help", self)
    provide_help_btn.clicked.connect(self.provide_help)

    button_layout = QHBoxLayout()
    button_layout.addWidget(extract_text_btn)
    button_layout.addWidget(provide_clean_image_btn)
    button_layout.addWidget(show_histogram_btn)
    button_layout.addWidget(provide_help_btn)

    preprocessing_layout = QHBoxLayout()
    preprocessing_layout.addWidget(self.method_combobox)
    preprocessing_layout.addWidget(self.dilation_request)

    image_layout = QHBoxLayout()
    image_layout.addWidget(self.image_label)
    image_layout.addWidget(self.clean_image_label)

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.threshold_label)
    layout.addWidget(self.threshold_slider)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
```

```
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    # Get the new threshold value from the slider
    threshold = self.threshold_slider.value()

    # Display the new threshold value
    self.threshold_label.setText(f"Threshold Value: {threshold}")

    # Using the grayscale image, perform the requested binary thresholding method
    # with the user-selected threshold value
    if method_idx == 1:
        # Choice 1: cv2.THRESH_BINARY
        _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY)
    elif method_idx == 2:
        # Choice 2: cv2.THRESH_BINARY_INV
        _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY_INV)
    elif method_idx == 3:
        # Choice 3: cv2.THRESH_TRUNC
        _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TRUNC)
    elif method_idx == 4:
        # Choice 4: cv2.THRESH_TOZERO
        _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO)
    elif method_idx == 5:
        # Choice 5: cv2.THRESH_TOZERO_INV
        _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO_INV)
    else:
        # Choice 0: Original Image
        image = self.image

    if self.dilation_request.isChecked():
        image = dilate_image(image)

    if self.compressed_img is not None:
```

```
            compressed_h, compressed_w = self.compressed_img.shape
            image = cv2.resize(image, (compressed_w, compressed_h))

        # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
        image_h, image_w = image.shape
        q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
        self.image_label.setPixmap(QPixmap.fromImage(q_img))

        # Update the histogram if the window is displayed
        if self.histogram_window is not None:
            self.histogram_window.update_histogram(image)
            self.histogram_window.show()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image, binary thresholding method, and threshold value
        to extract and display the text on the image.
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        # Get the new threshold value from the slider
        threshold = self.threshold_slider.value()

        if method_idx == 1:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY)
        elif method_idx == 2:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY_INV)
        elif method_idx == 3:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TRUNC)
        elif method_idx == 4:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO)
        elif method_idx == 5:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO_INV)
        else:
            image = self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        # Run Tesseract OCR on the image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
```

```
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for accuracy calculation.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        method_idx = self.method_combobox.currentIndex()
        threshold = self.threshold_slider.value()

        if method_idx == 1:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY)
        elif method_idx == 2:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_BINARY_INV)
        elif method_idx == 3:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TRUNC)
        elif method_idx == 4:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO)
        elif method_idx == 5:
            _, image = cv2.threshold(self.image, threshold, 255, cv2.THRESH_TOZERO_INV)
```

```
        else:
            QMessageBox.warning(self, "No Box Selected", "Please select a binary thresholding method before requesting"
                                                         " to save a binary image.")
            return
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Binary Thresholding" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Binary Thresholding:

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose one of several binary thresholding methods from the drop-down menu. You will then see the resulting binary image.
                Step 3: Play with the "Threshold" slider to update the binary image.
                Step 4: Click on "Extract Text" to perform text extraction.
                Step 5: Save the preprocessed binary image if needed.

                Additional Features:

                If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

                Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

                You can also bold the text be clicking the "Bold Text" checkbox.

                BIG Note:
```

Your image will be compressed in the application if it is over 512x512. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
"""

QMessageBox.information(self, "Help", help_text)

src.thresholding.otsus_thresholding.py:

```
import cv2
import numpy as np
import pytesseract
import os
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QHBoxLayout,
    QFileDialog,
    QComboBox,
    QMessageBox,
    QVBoxLayout,
    QCheckBox
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 512
max_height = 512


class OtsusThresholding(QWidget):
    """
    A GUI widget for applying Otsu's Method for thresholding to an image.

    This class provides a GUI for selecting an image and applying Otsu's Method for thresholding to it.
    Users can view the original grayscale image and the binary image created via Otsu's Method, as well as
    obtain computer-extracted text from either of these images. The user may also save the preprocessed binary image
    with a specific filename, image format, and directory. The user can also display and save the corresponding
    histogram graphs of these images as well.

    If the user provides a clean version of the image, the user can see the differences between the extracted
    text of the clean image and any of the other images. On top of that, the feature will also provide a text
```

```
extraction accuracy value.

This specific widget also has a feature that when given two directories (one noisy and one clean),
a bulk text extraction accuracy value can be calculated.

The user can also check a box to bold the text through a dilation of size 2x2.

Attributes:
    image (np.ndarray): The currently displayed image.
    compressed_img (np.ndarray): The compressed version of the currently displayed image (Only use if the image
    size is too large for the GUI)
    clean_image (np.ndarray): The clean version of the image for text extraction accuracy calculations.
    titles (list): A list of titles for different image display options.
    noisy_directory (str): The file path of a directory that has noisy images for bulk text extraction
    accuracy calculations.
    clean_directory (str): The file path of a directory that has clean images for bulk text extraction
    accuracy calculations.
"""

image: np.ndarray
compressed_img: np.ndarray
clean_image: np.ndarray
titles = ["Original Image", "THRESH_BINARY + cv2.THRESH_OTSU"]
noisy_directory: str
clean_directory: str

def __init__(self):
    """
    Initializes the Otsu's thresholding window for the application.
    """

    # Initialize the parent class (QWidget)
    super().__init__()

    self.setWindowTitle("Otsu's Thresholding")

    # Go through the titles and allow the user to see them and choose one through a combo box.
    self.method_combobox = QComboBox()
    for title in self.titles:
        self.method_combobox.addItem(title)
    self.method_combobox.currentIndexChanged.connect(self.update_image)

    # Allow the user to check a box to "bold the text" by performing dilation
    self.dilation_request = QCheckBox("Bold Text")
    self.dilation_request.stateChanged.connect(self.update_image)

    # Have labels to keep track of the given noisy and clean files
    self.noisy_label = QLabel("Noisy File: N/A")
    self.clean_label = QLabel("Clean File: N/A")
```

```
# Label that will hold the desired image
self.image_label = QLabel()
self.clean_image_label = QLabel()

# Prepare to use a compressed image if the provided image is too large to fit in the GUI
self.compressed_img = None

# Initialize the image label
self.image = np.tile(np.arange(256, dtype=np.uint8).repeat(2), (512, 1))
q_img = QImage(self.image.data, 512, 512, 512, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Label that will keep track of the calculated threshold values
self.threshold_label = QLabel("Calculated Threshold: N/A")

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to provide a noisy directory
provide_noisy_directory_btn = QPushButton("Provide Noisy Directory", self)
provide_noisy_directory_btn.clicked.connect(self.provide_noisy_directory)

# Prepare a button to provide a clean directory
provide_clean_directory_btn = QPushButton("Provide Clean Directory", self)
provide_clean_directory_btn.clicked.connect(self.provide_clean_directory)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

# Prepare a button that when pushed will provide help regarding Otsu's thresholding and the features of this
# program
```

```
    provide_help_btn = QPushButton("Help", self)
    provide_help_btn.clicked.connect(self.provide_help)

    button_layout = QHBoxLayout()
    button_layout.addWidget(extract_text_btn)
    button_layout.addWidget(provide_clean_image_btn)
    button_layout.addWidget(show_histogram_btn)
    button_layout.addWidget(provide_help_btn)

    preprocessing_layout = QHBoxLayout()
    preprocessing_layout.addWidget(self.method_combobox)
    preprocessing_layout.addWidget(self.dilation_request)

    # Put the clean and noisy directory buttons side to side (and the help button)
    directory_layout = QHBoxLayout()
    directory_layout.addWidget(provide_noisy_directory_btn)
    directory_layout.addWidget(provide_clean_directory_btn)

    image_layout = QHBoxLayout()
    image_layout.addWidget(self.image_label)
    image_layout.addWidget(self.clean_image_label)

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addLayout(preprocessing_layout)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.threshold_label)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addLayout(directory_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
```

```
            self.noisy_label.setText("Noisy File: " + file_name)
            if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
                image = cv2.resize(image, (max_width, max_height))
                self.compressed_img = np.array(image)
            elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
                image = cv2.resize(image, (image.shape[1], max_height))
                self.compressed_img = np.array(image)
            elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
                image = cv2.resize(image, (max_width, image.shape[0]))
                self.compressed_img = np.array(image)
            self.update_image()
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid image!")

    def update_image(self):
        """
        Updates the displayed image.
        """

        # Get the index of the selected combo box item
        method_idx = self.method_combobox.currentIndex()

        if method_idx == 1:
            # Choice 1: cv2.THRESH_BINARY + cv2.THRESH_OTSU

            # Using the grayscale image, calculate the new threshold value using Otsu's method
            # and retrieve the binary image that uses that value.
            ret, image = cv2.threshold(self.image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        else:
            # Choice 0: Original Image
            ret, image = "N/A", self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        if self.compressed_img is not None:
            compressed_h, compressed_w = self.compressed_img.shape
            image = cv2.resize(image, (compressed_w, compressed_h))

        # Display the new threshold value
        self.threshold_label.setText(f"Calculated Threshold: {ret}")

        # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
        image_h, image_w = image.shape
        q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
        self.image_label.setPixmap(QPixmap.fromImage(q_img))

        # Update the histogram if the window is displayed
        if self.histogram_window is not None:
```

```
        self.histogram_window.update_histogram(image)
        self.histogram_window.show()

    def extract_text(self):
        """
        Run Tesseract OCR using the user-selected image or its binary image after going through Otsu's method
        to extract and display the text on the image.
        """

        method_idx = self.method_combobox.currentIndex()
        if method_idx == 1:
            _, image = cv2.threshold(self.image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        else:
            _, image = "N/A", self.image

        if self.dilation_request.isChecked():
            image = dilate_image(image)

        # Run Tesseract OCR on the image
        text = pytesseract.image_to_string(image)

        # Display the extracted text
        QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

        # Calculate the text extraction accuracy if a clean image is provided
        if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
            clean_text = pytesseract.image_to_string(self.clean_image)
            accuracy = calculate_accuracy(text, clean_text)
            QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
            QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for text extraction accuracy calculations.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
```

```
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def provide_noisy_directory(self):
        """
        Allows the user to provide a noisy directory of images for bulk text extraction accuracy calculations.
        """

        noisy_directory = QFileDialog.getExistingDirectory(self, "Select Noisy Directory")
        if noisy_directory != "":
            self.noisy_directory = noisy_directory
            QMessageBox.information(self, "Success", "Received noisy directory!")
            self.process_directory()
        else:
            QMessageBox.warning(self, "Error", "Did not receive a noisy directory!")

    def provide_clean_directory(self):
        """
        Allows the user to provide a clean directory of images for bulk text extraction accuracy calculations.
        """

        clean_directory = QFileDialog.getExistingDirectory(self, "Select Clean Directory")
        if clean_directory != "":
            self.clean_directory = clean_directory
            QMessageBox.information(self, "Success", "Received clean directory!")
            self.process_directory()
        else:
            QMessageBox.warning(self, "Error", "Did not receive a clean directory!")

    def process_directory(self):
        """
        Checks that a noisy and clean directory have been provided. If so, iterates through both directories
        (assuming that both are alphabetically ordered so that the current file of each directory are the noisy and
        clean versions of the same image). Afterwards, a bulk text extraction accuracy will be calculated, along with
        the number of image comparisons used.
        """

        accuracy_sum = 0.0
        num_image_comparisons = 0

        # Check for a noisy and clean directory
        if hasattr(self, 'noisy_directory') and hasattr(self, 'clean_directory'):
```

```
            QMessageBox.information(self, "Got Both Directories", "Both directories received! Press OK to start, "
                                                    "bulk text extraction!")

            # Get the list of files from both directories (Remove .DS_Store if running on a Mac)
            noisy_files = sorted([f for f in os.listdir(self.noisy_directory) if f != '.DS_Store'])
            clean_files = sorted([f for f in os.listdir(self.clean_directory) if f != '.DS_Store'])

            for i, cur_noisy_file in enumerate(noisy_files):
                if cur_noisy_file.lower().endswith((".tiff", ".png", ".jpeg", ".jpg", ".bmp")):
                    if clean_files[i].lower().endswith((".tiff", ".png", ".jpeg", ".jpg", ".bmp")):

                        # Get the grayscale image of the current noisy and clean valid images
                        valid_noisy_img = cv2.imread(self.noisy_directory + "/" + cur_noisy_file, 0)
                        valid_clean_img = cv2.imread(self.clean_directory + "/" + clean_files[i], 0)

                        # Perform Otsu's thresholding on the noisy image
                        _, otsu_noisy_img = cv2.threshold(valid_noisy_img, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)

                        # Run Tesseract OCR on both images
                        noisy_text = pytesseract.image_to_string(otsu_noisy_img)
                        clean_text = pytesseract.image_to_string(valid_clean_img)

                        # Keep track of the accuracy sum and total number of image comparisons
                        cur_accuracy = calculate_accuracy(noisy_text, clean_text)
                        accuracy_sum += cur_accuracy
                        num_image_comparisons += 1

            bulk_accuracy = accuracy_sum / num_image_comparisons

            QMessageBox.information(self, "Image Comparisons", "# of Image Comparisons: " +
str(num_image_comparisons))
            QMessageBox.information(self, "Bulk Accuracy", "Bulk Text Extraction Accuracy: " + str(bulk_accuracy) +
"%")
        else:
            QMessageBox.warning(self, "Got One Directory",
                                "Provide the other directory to perform bulk text extraction!")

    def prepare_to_save(self):
        """
        Obtain the user-selected preprocessed binary image to prepare for saving to a directory.
        """

        _, image = cv2.threshold(self.image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        if self.dilation_request.isChecked():
            image = dilate_image(image)
        save_image(self, image)

    def show_histogram(self):
```

```
        """
        Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
        """

        # Create the histogram window if it doesn't exist, otherwise remove it
        if self.histogram_window is None:
            self.histogram_window = HistogramWindow(self.image)
        else:
            self.histogram_window = None
        self.update_image()

    def provide_help(self):
        """
        Display help information for the "Otsu's Thresholding" program.
        """

        help_text = """How to Enhance and Extract Text from an Image with Otsu's Method:

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose "THRESH_BINARY + cv2.THRESH_OTSU" from the drop-down menu. You will then see the resulting binary image that went through Otsu's thresholding.
                Step 3: Click on "Extract Text" to perform text extraction.
                Step 4: Save the preprocessed binary image if needed.

                Additional Features:

                If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted text from the clean image.

                Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

                Provide a noisy and clean directory using the "Provide Noisy Directory" and "Provide Clean Directory" buttons to calculate a bulk text extraction accuracy for all of your images.

                You can also bold the text be clicking the "Bold Text" checkbox.

                Note:

                If you want to calculate a bulk text extraction accuracy, make sure the files in both of your directories line up ALPHABETICALLY so that the first image in the noisy and clean directories are the noisy and clean versions of the same image, do this again for the second pair of images, and so on.

                BIG Note:

                Your image will be compressed in the application if it is over 512x512. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
```

```
        """

    QMessageBox.information(self, "Help", help_text)
```

src.util.accuracy.py:

```
import numpy as np


def calculate_accuracy(text, clean_text):
    """
    Calculates the accuracy of how well Tesseract OCR was able to read the preprocessed image in contrast to its
    clean image counterpart.

    :param text: The text extracted by Tesseract OCR from the preprocessed image.
    :type text: str
    :param clean_text: The text extracted by Tesseract OCR from the clean image counterpart.
    :type clean_text: str

    :return: The percentage of words between text and clean_text that match and are in the correct order.
    :rtype: float
    """

    text_words = text.split()
    clean_text_words = clean_text.split()

    # Implement the Levenshtein distance algorithm between the two texts

    # Create a distance_matrix to keep track of the total cost distance between the two texts
    # (add 1 extra row and column for padding)
    distance_matrix = np.zeros((len(text_words) + 1, len(clean_text_words) + 1))
    for i in range(len(text_words) + 1):
        for j in range(len(clean_text_words) + 1):

            # Add padding for the first row and column of the distance_matrix. These are the base costs/distances
            # for comparing an empty string to a non-empty string.
            if i == 0:
                distance_matrix[i][j] = j
            elif j == 0:
                distance_matrix[i][j] = i

            # Check if the current text_word matches the current clean_text_word. If so, no changes need to be made,
            # and we keep the cost the same.
            elif text_words[i - 1] == clean_text_words[j - 1]:
                distance_matrix[i][j] = distance_matrix[i - 1][j - 1]

            # If it isn't, check neighbors that have already been calculated, find the minimum distance, and add 1
            # (1 is added because there has to be a cost when the words don't match)
```

```
            else:
                distance_matrix[i][j] = 1 + min(distance_matrix[i - 1][j],
                                                distance_matrix[i][j - 1],
                                                distance_matrix[i - 1][j - 1])

    # Get the maximum possible distance between the two texts
    max_distance = max(len(text_words), len(clean_text_words))

    # Calculate the accuracy using max_distance and the final calculated distance
    accuracy = ((max_distance - distance_matrix[-1][-1]) / max_distance) * 100

    return accuracy
```

src.util.dialog.py:

```
from typing import Optional
import cv2
from PySide6.QtWidgets import QFileDialog, QWidget


def open_image_dialog(parent: Optional[QWidget] = None):
    """
    Opens a file dialog for the user to select an image file.

    :param parent: The parent widget for the file dialog. Set to None as default.
    :type parent: Optional[QWidget]

    :return: The grayscale version of the selected image as a NumPy array, or None if no image is selected that is in
            an acceptable format or the operation is cancelled by the user.
    :rtype: Optional[numpy.ndarray]

    :return: The file name of the given image or None if no image is selected that is in an acceptable format or the
            operation is cancelled by the user.

    """

    # Image must satisfy the filter, otherwise image_path will just be an empty string
    image_path, _ = \
        QFileDialog.getOpenFileName(parent, "Load Image", filter="Image Files (*.tiff *.png *.jpeg *.jpg *.bmp)")

    if image_path:
        # Extract the file name from the full path
        file_name = image_path.split("/")[-1]  # Unix-like path (Mac & Linux)
        file_name = file_name.split("\\")[-1]  # Windows path

        return cv2.imread(image_path, 0), file_name

    return None, None
```

src.util.dilate_image.py:

```
import numpy as np
import cv2


def dilate_image(image):
    """
    Dilates the foreground of the given image.
    :param image: The image of which to dilate the foreground
    :type image: np.ndarray

    :return: The image after the foreground undergoes dilation.
    :rtype: np.ndarray
    """

    # Bold text with have a dilation with a matrix size of 2x2.
    kernel = np.ones((2, 2), np.uint8)

    # Invert the image
    image = 255 - image

    # Dilate the image
    image = cv2.dilate(image, kernel, iterations=1)

    # Un-invert the dilated image
    image = 255 - image  # Un-invert the dilated image

    return image
```

src.util.get_best_image.py:

```
import cv2
import pytesseract
from pytesseract import Output
import threading
from concurrent.futures import ThreadPoolExecutor
import multiprocessing
import numpy as np
from PySide6.QtGui import QImage, QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QPushButton,
    QLabel,
    QMessageBox,
    QVBoxLayout,
    QComboBox,
    QHBoxLayout
```

```
)

from src.util.dialog import open_image_dialog
from src.util.accuracy import calculate_accuracy
from src.util.save import save_image
from src.util.histogram import HistogramWindow
from src.util.dilate_image import dilate_image

max_width = 512
max_height = 512


class GetBestImage(QWidget):

    image: np.ndarray
    best_image: np.ndarray
    best_image_confidence: float
    best_accuracy: float
    titles = ["Original Image", "Get Best Image"]
    clean_image: np.ndarray
    lock: threading.Lock

    def __init__(self):
        """
        Initializes the "Find Best Image" window for the application.
        """

        # Initialize the parent class (QWidget)
        super().__init__()

        self.setWindowTitle("Get Best Image")

        # Go through the titles and allow the user to see them and choose one through a combo box.
        self.method_combobox = QComboBox()
        for title in self.titles:
            self.method_combobox.addItem(title)
        self.method_combobox.currentIndexChanged.connect(self.update_image)

        # Have labels to keep track of the given noisy and clean files
        self.noisy_label = QLabel("Noisy File: N/A")
        self.clean_label = QLabel("Clean File: N/A")

        # Label that will hold the desired image
        self.image_label = QLabel()
        self.clean_image_label = QLabel()

        # Prepare to use a compressed image if the provided image is too large to fit in the GUI
        self.compressed_img = None
```

```
# Initialize the image label
self.image = np.tile(np.arange(256, dtype=np.uint8).repeat(2), (512, 1))
q_img = QImage(self.image.data, 512, 512, 512, QImage.Format_Indexed8)
self.image_label.setPixmap(QPixmap.fromImage(q_img))

# Label that will hold the histogram of the desired image
self.histogram_label = QLabel()

# Wait to display the histogram window until the user requests it
self.histogram_window = None

# Prepare a button that when pushed will open the file dialog for the user
open_image_btn = QPushButton("Open Image", self)
open_image_btn.clicked.connect(self.open_image)

# Prepare an extract text button that will utilize the current image and slider values
extract_text_btn = QPushButton("Extract Text", self)
extract_text_btn.clicked.connect(self.extract_text)

# Prepare a button to provide a clean version of the image
provide_clean_image_btn = QPushButton("Provide Clean Image", self)
provide_clean_image_btn.clicked.connect(self.provide_clean_image)

# Prepare a button to allow the user to save the preprocessed binary image
save_btn = QPushButton("Save Preprocessed Binary Image", self)
save_btn.clicked.connect(self.prepare_to_save)

# Prepare a button to show the histogram
show_histogram_btn = QPushButton("Show/Hide Histogram")
show_histogram_btn.clicked.connect(self.show_histogram)

# Prepare a button that when pushed will provide help regarding Otsu's thresholding and the features of this
# program
provide_help_btn = QPushButton("Help", self)
provide_help_btn.clicked.connect(self.provide_help)

button_layout = QHBoxLayout()
button_layout.addWidget(extract_text_btn)
button_layout.addWidget(provide_clean_image_btn)
button_layout.addWidget(show_histogram_btn)
button_layout.addWidget(provide_help_btn)

image_layout = QHBoxLayout()
image_layout.addWidget(self.image_label)
image_layout.addWidget(self.clean_image_label)

# Label to display the current best preprocessing method and parameters
self.method_label = QLabel("Best Preprocessing Method: N/A")
```

```
    self.best_image = None
    self.best_image_confidence = 0.0
    self.best_accuracy = 0.0
    self.lock = threading.Lock()

    # Create layout and add widgets
    layout = QVBoxLayout()
    layout.addWidget(open_image_btn)
    layout.addWidget(self.method_combobox)
    layout.addWidget(self.noisy_label)
    layout.addWidget(self.clean_label)
    layout.addWidget(self.method_label)
    layout.addLayout(image_layout)
    layout.addLayout(button_layout)
    layout.addWidget(save_btn)

    # Set dialog layout
    self.setLayout(layout)

def open_image(self):
    """
    Opens a file dialog and displays the user-selected image.
    """

    # Allow the user to select an image
    image, file_name = open_image_dialog()

    # Check if the user gave an image with a valid format. If so, update the instance's image
    # attribute and update the image label.
    if image is not None:
        self.image = np.array(image)
        self.noisy_label.setText("Noisy File: " + file_name)
        if image.shape[0] > max_height and image.shape[1] > max_width:     # Both height and width are too large
            image = cv2.resize(image, (max_width, max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] > max_height and image.shape[1] <= max_width:  # Height is too large
            image = cv2.resize(image, (image.shape[1], max_height))
            self.compressed_img = np.array(image)
        elif image.shape[0] <= max_height and image.shape[1] > max_width:  # Width is too large
            image = cv2.resize(image, (max_width, image.shape[0]))
            self.compressed_img = np.array(image)
        self.best_image = None
        self.best_image_confidence = 0.0
        self.best_accuracy = 0.0
        self.update_image()
    else:
        QMessageBox.warning(self, "Error", "Did not receive a valid image!")

def update_image(self):
```

```
    """
    Updates the displayed image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    if method_idx == 0:
        # Choice 0: Original Image
        image = self.image
    else:
        # Choice 1: Find Best Image
        if self.best_image is not None:
            image = self.best_image
        else:
            self.find_best_image()
            if self.best_image is not None:
                image = self.best_image
            else:
                image = self.image
                QMessageBox.warning(self, "Error", "Could not calculate an ideal image!")

    if self.compressed_img is not None:
        compressed_h, compressed_w = self.compressed_img.shape
        image = cv2.resize(image, (compressed_w, compressed_h))

    # Update the image label by converting the image to a QImage and setting it as the pixmap for the image label
    image_h, image_w = image.shape
    q_img = QImage(image.data, image_w, image_h, image_w, QImage.Format_Indexed8)
    self.image_label.setPixmap(QPixmap.fromImage(q_img))

    # Update the histogram if the window is displayed
    if self.histogram_window is not None:
        self.histogram_window.update_histogram(image)
        self.histogram_window.show()

def extract_text(self):
    """
    Run Tesseract OCR using the currently displayed image and display the text on the image.
    """

    # Get the index of the selected combo box item
    method_idx = self.method_combobox.currentIndex()

    if method_idx == 0:
        image = self.image
    else:
        image = self.best_image
```

```
        if image is not None:
            # Run Tesseract OCR on the image
            text = pytesseract.image_to_string(image)

            # Display the extracted text
            QMessageBox.information(self, "Text", "Extracted Text: \n\n" + text)

            # Calculate the text extraction accuracy if a clean image is provided
            if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
                clean_text = pytesseract.image_to_string(self.clean_image)
                accuracy = calculate_accuracy(text, clean_text)
                QMessageBox.information(self, "Clean Text", "Extracted Clean Text: \n\n" + clean_text)
                QMessageBox.information(self, "Accuracy", "Text Extraction Accuracy: " + str(accuracy) + "%")
        else:
            QMessageBox.warning(self, "Preprocessing Error", "Cannot extract text from a nonexistent preprocessed
image")

    def provide_clean_image(self):
        """
        Allows the user to provide a clean version of the image for text extraction accuracy calculations.
        """

        clean_image, clean_file_name = open_image_dialog()
        if clean_image is not None:
            self.clean_image = np.array(clean_image)
            self.clean_label.setText("Clean File: " + clean_file_name)
            if clean_image.shape[0] > max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, max_height))
            elif clean_image.shape[0] > max_height and clean_image.shape[1] <= max_width:
                clean_image = cv2.resize(clean_image, (clean_image.shape[1], max_height))
            elif clean_image.shape[0] <= max_height and clean_image.shape[1] > max_width:
                clean_image = cv2.resize(clean_image, (max_width, clean_image.shape[0]))

            clean_image_h, clean_image_w = clean_image.shape
            q_img = QImage(clean_image.data, clean_image_w, clean_image_h, clean_image_w,
QImage.Format_Indexed8)
            self.clean_image_label.setPixmap(QPixmap.fromImage(q_img))

            QMessageBox.information(self, "Success",
                                    "Valid clean image received! Press \"Extract Text\" for an accuracy calculation!")
        else:
            QMessageBox.warning(self, "Error", "Did not receive a valid clean image!")

    def prepare_to_save(self):
        """
        Obtain the currently displayed preprocessed binary image to prepare for saving to a directory.
        """

        if self.best_image is not None:
```

```
        save_image(self, self.best_image)
    else:
        QMessageBox.warning(self, "Error", "Please wait until the best image has been created before saving.")

def show_histogram(self):
    """
    Shows or hides the histogram with respect to the currently displayed image if the histogram button is pressed.
    """

    # Create the histogram window if it doesn't exist, otherwise remove it
    if self.histogram_window is None:
        if self.best_image is not None:
            self.histogram_window = HistogramWindow(self.best_image)
        else:
            self.histogram_window = HistogramWindow(self.image)
    else:
        self.histogram_window = None

    self.update_image()

def find_best_image(self):
    """
    Runs through a variety of preprocessing methods in parallel to determine the image the computer is able to read
    with the most confidence.

    If a clean image is provided by the user, compare the preprocessed binary image with the clean image and
    return the image with the highest accuracy.
    """

    self.best_image = None
    self.best_image_confidence = 0.0
    self.best_accuracy = 0.0

    clean_img_text = None
    if hasattr(self, 'clean_image') and isinstance(self.clean_image, np.ndarray):
        clean_img_text = pytesseract.image_to_string(self.clean_image)

    # Track the number of words from the original noisy image
    original_text = pytesseract.image_to_string(self.image)
    original_word_count = len(original_text.split())

    if original_word_count != 0:
        methods = [
            ("Binary Thresholding", {"start": 100, "end": 200 + 1, "bold": False}),                    # Core 1
            ("Adaptive Thresholding (Mean)", {"c_constants": list(range(0, 100 + 1)), "bold": False}),      # Core 2
            ("Adaptive Thresholding (Gaussian)", {"c_constants": list(range(0, 100 + 1)), "bold": False}),  # Core 3
            ("Median Filtering", {"kernel_sizes": list(range(3, 21 + 1, 2)), "bold": False}),              # Core 4
            ("Binary Thresholding 2", {"start": 100, "end": 200 + 1, "bold": True}),                    # Core 5
            ("Adaptive Thresholding 2 (Mean)", {"c_constants": list(range(0, 100 + 1)), "bold": True}),     # Core 6
```

```
            ("Adaptive Thresholding 2 (Gaussian)", {"c_constants": list(range(0, 100 + 1)), "bold": True}), # Core 7
            ("Median Filtering 2", {"kernel_sizes": list(range(3, 21 + 1, 2)), "bold": True}),             # Core 8
        ]
        num_cores = multiprocessing.cpu_count()

        if num_cores >= 8:
            # Run methods concurrently if the user has 8 or more cores
            with ThreadPoolExecutor(max_workers=num_cores) as executor:

                # Submit Otsu's method separately (Only 1 thread needed)
                otsu_future = executor.submit(self.find_best_otsu_image, [False, True], original_word_count,
                                              clean_img_text)

                # Submit other methods to the thread pool
                futures = [executor.submit(self.find_best_image_method, method, params, original_word_count,
                                           clean_img_text) for method, params in methods]

                # Wait for Otsu's method
                otsu_future.result()

                # Wait for all other methods to finish
                for future in futures:
                    future.result()
        else:
            # Run methods sequentially if the user has fewer than 8 cores
            self.find_best_otsu_image([False, True], original_word_count, clean_img_text)
            for method, params in methods:
                self.find_best_image_method(method, params, original_word_count, clean_img_text)
    else:
        QMessageBox.warning(self, "Error", "Could not find any text in your image")

def find_best_otsu_image(self, bold, original_word_count, clean_img_text):
    """
    Runs Otsu's method with and without the text "bolding" and checks if it is the most computer-recognizable
image.
    """

    ret, image = cv2.threshold(self.image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    self.calculate_doc_confidence(image, "Otsu's Thresholding", f"\nCalculated Threshold: {ret}, Bold Text: "
                                                                f"{bold[0]}", original_word_count, clean_img_text)
    image = dilate_image(image)
    self.calculate_doc_confidence(image, "Otsu's Thresholding", f"\nCalculated Threshold: {ret}, Bold Text: "
                                                                f"{bold[1]}", original_word_count, clean_img_text)

def find_best_image_method(self, method, params, original_word_count, clean_img_text):
    """
    Runs through ideal thresholding methods that are useful in removing noise from text documents by testing
    different parameter combinations (Binary Thresholding, Adaptive Thresholding, and Median Filtering). Check
both
```

```
    original text and text "bolding"

    :param method: The assigned preprocessing method that a core will try
    :param params: The set of parameters the core will try on the assigned preprocessing method
    :param original_word_count: The number of words extracted from the original image
    :param clean_img_text: Optional, The extracted text from the provided clean image
    :type original_word_count: int
    """

    method_name, method_params = method, params
    if method_name == "Binary Thresholding" or method_name == "Binary Thresholding 2":
        start, end, bold = method_params["start"], method_params["end"], method_params["bold"]
        for i in range(start, end, 1):
            _, image = cv2.threshold(self.image, i, 255, cv2.THRESH_BINARY)
            if bold:
                image = dilate_image(image)
            self.calculate_doc_confidence(image, "Binary Thresholding", f"\nThreshold: {i}, Bold Text: {bold}",
                                          original_word_count, clean_img_text)

    elif method_name == "Adaptive Thresholding (Mean)" or method_name == "Adaptive Thresholding 2
(Mean)":
        c_constants, bold = method_params["c_constants"], method_params["bold"]
        for c_constant in c_constants:
            image = cv2.adaptiveThreshold(self.image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY,
                                          255, c_constant)
            if bold:
                image = dilate_image(image)
            self.calculate_doc_confidence(image, "Adaptive Thresholding", f"\nParameters:
ADAPTIVE_THRESH_MEAN_C, "
                                                  f"Block Size: 255, C Constant: "
                                                  f"{c_constant}, Bold Text: {bold}",
                                                  original_word_count, clean_img_text)
    elif method_name == "Adaptive Thresholding (Gaussian)" or method_name == "Adaptive Thresholding 2
(Gaussian)":
        c_constants, bold = method_params["c_constants"], method_params["bold"]
        for c_constant in c_constants:
            image = cv2.adaptiveThreshold(self.image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY,
                                          255, c_constant)
            if bold:
                image = dilate_image(image)
            self.calculate_doc_confidence(image, "Adaptive Thresholding", f"\nParameters: "
                                                  f"ADAPTIVE_THRESH_GAUSSIAN_C, "
                                                  f"Block Size: 255, C Constant: "
                                                  f"{c_constant}, Bold Text: {bold}",
                                                  original_word_count, clean_img_text)
    elif method_name == "Median Filtering" or method_name == "Median Filtering 2":
        kernel_sizes, bold = method_params["kernel_sizes"], method_params["bold"]
```

```
        for kernel_size in kernel_sizes:
            image = cv2.medianBlur(self.image, kernel_size)
            if bold:
                image = dilate_image(image)
            self.calculate_doc_confidence(image, "Median Filtering",
                                          f"\nParameters: Kernel Size: {kernel_size}, Bold Text: "
                                          f"{bold}", original_word_count, clean_img_text)

    def calculate_doc_confidence(self, image, method, parameters, original_word_count, clean_img_text):
        """
        Runs the preprocessed image through Tesseract and obtain word confidence values for every word extracted
        from the image. Keeps track of the image with the highest confidence average.

        If there is a clean image, keep track of the image with the highest accuracy instead.

        :param image: The preprocessed image
        :param method: The preprocessing method utilized
        :param parameters: The parameters used for the method
        :param original_word_count: The number of words extracted from the original image
        :param clean_img_text: Optional, The extracted text from the provided clean image
        :type image: np.ndarray
        :type original_word_count: int
        """

        if clean_img_text is None:
            result = pytesseract.image_to_data(image, config=r'--oem 3 --psm 6', output_type=Output.DICT)
            confidences = result['conf']

            num_words = 0.0
            confidence_total = 0

            for word, confidence in zip(result['text'], confidences):
                if word == "" and confidence == -1:     # More common error (no word and no confidence)
                    continue
                elif word != "" and confidence == -1:   # Less common error (word, but no confidence)
                    num_words = num_words + 1.0
                else:
                    num_words = num_words + 1.0
                    confidence_total = confidence_total + confidence

            if num_words != 0.0 and pytesseract.image_to_string(image).strip() != "":

                retained_word_percentage = num_words / original_word_count

                # Ignore text documents that lost more than 10% of text
                if retained_word_percentage >= 0.90:
                    doc_confidence_percentage = (confidence_total / num_words)

                    with self.lock:
```

```
                if doc_confidence_percentage >= self.best_image_confidence:
                    self.best_image_confidence = doc_confidence_percentage
                    self.best_image = image.copy()  # Make a copy to avoid shared references
                    self.update_best_method_label(method, parameters)

        else:
            preprocess_text = pytesseract.image_to_string(image)
            accuracy = calculate_accuracy(preprocess_text, clean_img_text)
            with self.lock:
                if accuracy >= self.best_accuracy:
                    self.best_accuracy = accuracy
                    self.best_image = image.copy()
                    self.update_best_method_label(method, parameters)

    def update_best_method_label(self, method, parameters):
        """
        Change the label to show the user the preprocessing method and parameters used that produced the image
        that the computer believes is the most recognizable.

        If a clean image is provided, the label is based on the most accurate preprocessed image with respect to the
        clean image.

        :param method: The preprocessing method utilized
        :param parameters: The parameters used for the method
        """

        label_text = f"Best Preprocessing Method: {method} {parameters}"
        self.method_label.setText(label_text)

    def provide_help(self):
        """
        Display help information for the "Get Best Image" program.
        """

        help_text = """How to get "The Best Image":

                Step 1: Pass in your noisy image with "Open Image."
                Step 2: Choose "Get Best Image" from the drop-down menu. The program will calculate a binary image
that the computer is able to recognize with the highest confidence based on extracted text (excluding calculated
images with no recognizable text). This process will take some time, especially if your image is large. Once
completed, you will be able to see the resulting binary image.
                Step 3: Click on "Extract Text" to perform text extraction.
                Step 4: Save the preprocessed binary image if needed.

                Additional Features:

                If you pass in a clean version of your noisy image with "Provide Clean Image" and then press "Extract
Text," you can get the extracted text from the noisy and clean counterparts. You will also get a text extraction
accuracy from 0%-100%, which represents how well the extracted text from the noisy image resembles the extracted
```

text from the clean image. The best image will also be with respect to the given clean image rather than what the computer believes is the most recognizable image.

Press "Show/Hide Histogram" to get or remove the histogram of the currently displayed image.

Note:

"The Best Image" in terms of this application means the most computer-recognizable binary image, not necessarily the most human-recognizable image. Your image will be tested on popular preprocessing techniques that are most likely to enhance text documents, including Otsu's, binary, and adaptive thresholding, along with median filtering. It is almost certain the calculated image will not be the best it can possibly be because there are many other complex preprocessing techniques that are either available or under development that may work better than the ones available on this application. THERE IS NO ONE SIZE FITS ALL SOLUTION TO IMAGE ENHANCEMENT!

BIG Note:

Your image will be compressed in the application if it is over 512x512. However, saving the binary image will be based on the original dimensions of the given image. The same applies to the extracted text, accuracy calculations, and histograms.
"""

QMessageBox.information(self, "Help", help_text)

src.util.histogram.py:

```
import cv2
import matplotlib.pyplot as plt
import os
from PySide6.QtGui import QPixmap
from PySide6.QtWidgets import (
    QWidget,
    QVBoxLayout,
    QLabel,
    QPushButton,
    QFileDialog,
    QMessageBox
)


class HistogramWindow(QWidget):
    """
    A widget for displaying and saving the histogram of an image.

    This class provides a GUI for displaying the histogram of an image and allowing the user to save it
    as an image file.

    Attributes:
        image (np.ndarray): The image data for which the histogram is to be displayed.
```

```
    histogram_label (QLabel): A label for displaying the histogram.
"""

def __init__(self, image):
    """
    Initializes the HistogramWindow with the given image.

    :param image: The image data for which the histogram is to be displayed.
    :type image: np.ndarray
    """

    super().__init__()

    self.image = image
    self.histogram_label = QLabel("Histogram")

    # Allow the user to save the current histogram being displayed
    save_histogram_btn = QPushButton("Save Histogram")
    save_histogram_btn.clicked.connect(self.save_histogram)

    layout = QVBoxLayout()
    layout.addWidget(self.histogram_label)
    layout.addWidget(save_histogram_btn)

    self.calculate_histogram()

    self.setLayout(layout)

def update_histogram(self, new_image):
    """
    Updates the displayed histogram with respect to a new image.

    :param new_image: The new image data for which the histogram needs to be based on.
    :type new_image: np.ndarray
    """

    self.image = new_image
    self.calculate_histogram()

def save_histogram(self):
    """
    Saves the displayed histogram as an image file.
    """

    self.calculate_histogram()

    # Open a file dialog to allow the user to select a save location and filename (set to .png as default)
    file_path, _ = QFileDialog.getSaveFileName(self, "Save Histogram", "",
                                               "Images (*.png *.tiff *.jpeg *.jpg *.bmp)")
```

```
        # Check if the user entered a directory and did not cancel the operation
        if file_path != "":
            # Save the histogram to the requested file path
            if self.histogram_label.pixmap().save(file_path):
                QMessageBox.information(self, "Success", "Histogram saved successfully!")
            else:
                QMessageBox.warning(self, "Error", "Failed to save histogram!")
        else:
            QMessageBox.warning(self, "Error", "Saving operation cancelled!")

    def calculate_histogram(self):
        """
        Calculates and displays the histogram of the current image.
        """

        hist = cv2.calcHist([self.image], [0], None, [256], [0, 256])
        hist /= hist.sum()

        # Clear the previous histogram data
        plt.clf()

        # Plot the histogram
        plt.plot(hist)
        plt.xlabel('Pixel Value')
        plt.ylabel('Normalized Frequency')
        plt.title("Histogram")
        plt.xlim(0 - 2, 255 + 2)
        plt.ylim(0 - 0.01, 1 + 0.01)

        # Save the histogram to a temporary file
        plt.savefig('histogram.png')

        # Set the pixmap to the histogram label
        pixmap = QPixmap('histogram.png')

        # Set the pixmap to the histogram label
        self.histogram_label.setPixmap(pixmap)

        # Remove the temporary file
        os.remove('histogram.png')
```

src.util.math.py:

```
import numpy as np


def round_up_to_odd(number: float) -> int:
```

```
    """
    Rounds up a given floating point number to the nearest odd integer. If the floating point number
    is already an odd integer, it returns the same number cast as an int.

    :param number: The numerical value to be rounded to the nearest odd integer.
    :type number: float

    :return: The rounded-up odd integer.
    :rtype: int
    """

    return int(np.ceil(number) // 2 * 2 + 1)
```

src.util.save.py:

```
import cv2
from PySide6.QtWidgets import (QFileDialog, QMessageBox)


def save_image(self, preprocessed_binary_img):
    """
    Opens a file dialog to allow the user to select a directory, valid image format, and file name. If the user confirms
    the save operation, the preprocessed binary image is saved to the chosen filepath, converted to the requested
    image format (or a default format if the given format is not valid), and given the chosen filename.

    :param self: The instance of the main window or widget.
    :param preprocessed_binary_img: The preprocessed binary image to be saved.
    :type preprocessed_binary_img: np.ndarray
    """

    # Open a file dialog to allow the user to select a save location, image format, and file name
    # Note: The image format is set to .png by default
    file_path, _ = QFileDialog.getSaveFileName(self, "Save Image", "", "Images (*.png *.tiff *.jpeg *.jpg *.bmp)")

    # Check if the user entered a directory and did not cancel the operation
    if file_path != "":
        # Use OpenCV to write the image to the chosen file path
        cv2.imwrite(file_path, preprocessed_binary_img)
        QMessageBox.information(self, "Success", "Image saved successfully!")
    else:
        QMessageBox.warning(self, "Error", "Saving operation cancelled or failed!")
```

src.run.py:

```
import sys

from PySide6.QtWidgets import (
    QApplication,
```

```
    QVBoxLayout,
    QMainWindow,
    QListWidget,
    QListWidgetItem,
    QPushButton,
    QWidget,
    QHBoxLayout,
    QMessageBox
)

from src.thresholding.adaptive_thresholding import AdaptiveThresholding
from src.thresholding.binary_thresholding import BinaryThresholding
from src.thresholding.otsus_thresholding import OtsusThresholding
from src.filtering.median_filtering import MedianFiltering
from src.filtering.gaussian_blur import GaussianBlur
from src.filtering.bilateral_filtering import BilateralFiltering
from src.morphology.opening_closing import OpeningClosing
from src.morphology.closing_opening import ClosingOpening
from src.util.get_best_image import GetBestImage

# Dictionary that holds the program names as keys, and class definitions as values
programs = {
    "Otsu's Thresholding": OtsusThresholding,
    "Binary Thresholding": BinaryThresholding,
    "Adaptive Thresholding": AdaptiveThresholding,
    "Opening-Closing Morphology": OpeningClosing,
    "Closing-Opening Morphology": ClosingOpening,
    "Median Filtering": MedianFiltering,
    "Gaussian Blur Filtering": GaussianBlur,
    "Bilateral Filtering": BilateralFiltering,
    "Get Best Image": GetBestImage
}


class ProgramSelector(QMainWindow):
    """
    A GUI application for selecting and running the available image preprocessing programs.

    This class represents the main window of the application. It allows the user to select
    an image preprocessing program from a list and execute it by clicking a start button.

    Attributes:
        instances (list): A list to keep track of instances of the selected image preprocessing programs.
    """

    instances = []

    def __init__(self):
        """
```

```
    Initializes the main window of the application.
    """

    # Need to initialize the parent class (QMainWindow) to show the main window
    super().__init__()

    self.setWindowTitle("OpenCV with Python")

    layout = QVBoxLayout()
    self.program_list = QListWidget()

    # Iterate through each key in the programs dictionary to display all available thresholding programs
    # to the user as a list widget
    for program in programs:
        self.program_list.addItem(QListWidgetItem(program))

    start_button = QPushButton("Start")
    start_button.clicked.connect(self.start)

    help_button = QPushButton("Start Where?")
    help_button.clicked.connect(self.give_help)

    button_layout = QHBoxLayout()
    button_layout.addWidget(start_button)
    button_layout.addWidget(help_button)

    layout.addWidget(self.program_list)
    layout.addLayout(button_layout)

    # Set the central widget to ensure it contains the start button and thresholding programs
    central_widget = QWidget()
    central_widget.setLayout(layout)
    self.setCentralWidget(central_widget)

def start(self):
    """
    Start the requested image preprocessing program.
    """

    try:
        # Get the text from the requested image preprocessing program to use as a key
        # to the programs dictionary and obtain an instance of its corresponding class
        item = self.program_list.selectedItems()[0]
        instance = programs.get(item.text())()

        # Save the instance for later use
        self.instances.append(instance)

        # Show the instance (can do this thanks to initializing the parent class)
```

```
            instance.show()
        except IndexError:
            # Reaches here if the user clicks the start and no image preprocessing program is selected.
            # In that case, do nothing
            return

    def give_help(self):
        """
        Display general help information
        """

        help_text = """To begin, choose any one of the nine subprograms and press the "Start" button. Here is a brief
description of each subprogram:

            Otsu's Thresholding: Automatically calculates an optimal threshold value that separates the foreground
from the background. Overall helpful, but probably not for "coffee-stain" or "salt-and-pepper" noise.

            Binary Thresholding: The user can manually input a threshold value rather than automatically. May be
helpful if Otsu's thresholding worsens the noise.

            Adaptive Thresholding: Divides the image into regions and calculates thresholds for each region. Works
generally well, including on "coffee-stain" noise.

            Opening-Closing Morphology: Dilates, then erodes specified image regions.

            Closing-Opening Morphology: Erodes, then dilates specified image regions.

            Median Filtering: Calculates the median value for every pixel based on a specified size of neighboring
pixels.

            Gaussian Blur Filtering: Similar to "Median Filtering," except it uses weighted sums instead of medians.

            Bilateral Filtering: Goes through every pixel and considers a specified size of neighboring pixels and their
intensity differences so that only nearby and similar pixels are blurred. Best for reducing "salt-and-pepper" noise.

            Get Best Image: Utilizes some of the above preprocessing methods to determine an ideal
computer-readable image. Use this if you are trying to remove as much noise from a text document as possible.

            Acceptable file formats include .tiff, .png, .jpeg, .jpg, and .bmp.
            """

        QMessageBox.information(self, "Help", help_text)


# Check that run.py is executed directly, otherwise do not run the application
if __name__ == "__main__":
    # Initialize a QApplication and ProgramSelector class, show the subprogram
    # options, and keep allowing for input until the user closes out.
    app = QApplication(sys.argv)
```

```
ex = ProgramSelector()
ex.show()
sys.exit(app.exec())
```

References

Balamurugan, E., Sengottuvelan, P., & Sangeetha, K. (2013, November). *An Empirical Evaluation of Salt and Pepper Noise Removal for Document Images using Median Filter*. research.ijcaonline.org. https://research.ijcaonline.org/volume82/number4/pxc3892139.pdf

Brownlee, J. (2022, September 12). *Number of CPUs in Python*. SuperFastPython.com. https://superfastpython.com/number-of-cpus-python/#:~:text=We%20can%20get%20the%20count%20of%20the%20number%20of%20CPUs,will%20return%20the%20value%20None.

Chinnasarn, K. (n.d.). *Removing Salt-and-Pepper Noise in Text/Graphics Images*. *Faculty of Science, Burapa University*. angsila.cs.buu.ac.th. https://angsila.cs.buu.ac.th/~krisana/cv/paper/Apccas98.pdf

Coman, I. (2024, February 8). *Image Analysis - Part 1: Binary Image Analysis* [PowerPoint *slides]. College of Liberal Arts and Sciences, SUNY Oswego. https://mylearning.suny.edu/d2l/le/content/1083058/viewContent/33296009/View*

Coman, I. (2024, February 15). *Texture* [PowerPoint slides]. College of Liberal Arts and Sciences, SUNY Oswego. https://mylearning.suny.edu/d2l/le/content/1083058/viewContent/33414578/View

Cukierski, W. (2015). *Denoising Dirty Documents*. Kaggle.com. https://kaggle.com/competitions/denoising-dirty-documents

Espaa-Boquera,S., Pastor-Pellicer,J., Castro-Bleda,M.J., and Zamora-Martinez,F.. (2015). NoisyOffice. UCI Machine Learning Repository. https://doi.org/10.24432/C5G31N.

Huamán, A. (n.d.). *Histogram Equalization*. Docs.OpenCV.org.

https://docs.opencv.org/3.4/d4/d1b/tutorial_histogram_equalization.html

Hoffstaetter, S. et al., (2022, August 16). *pytesseract 0.3.10*. PyPI.org. https://pypi.org/project/pytesseract/

Jayanetti, P. (2020, October 16). *Remove Salt and Pepper noise with Median Filtering*. Medium.com. https://medium.com/analytics-vidhya/remove-salt-and-pepper-noise-with-median-filtering-b739614fe9db

Kumar, A. (2019, August 22). *Denoising Documents with background noise.* Medium.com. https://medium.com/@amardeepkumar_25731/denoising-documents-with-background-noise-f449e1fd92d2

North Code. (2020, July 13). *Enhance a Document Scan using Python and OpenCV*. YouTube.com. https://www.youtube.com/watch?v=tYF3EBkvYO0&t=1s

Nurfikri, F. (2022, November 1). *How to Build Optical Character Recognition (OCR) in Python*. *BuiltIn.com. https://builtin.com/data-science/python-ocr*

Prasad, S. (2020, December 10). *Python for Character Recognition – Tesseract*. Topcoder.com. *https://www.topcoder.com/thrive/articles/python-for-character-recognition-tesseract*

Python Software Foundation. (2024, April 16). threading - Thread-based parallelism. docs.python.org. https://docs.python.org/3/library/threading.html#lock-objects

The Qt Company. (2024). *QFileDialog Class*. doc.qt.io. https://doc.qt.io/qt-6/qfiledialog.html

Rosebrock, A. (2021, April 28). *OpenCV Image Histograms (cv2.calcHist)*. PyImageSearch.com. https://pyimagesearch.com/2021/04/28/opencv-image-histograms-cv2-calchist/

Sanchhaya Education Private Limited. (2023, February 15). *Image Enhancement Techniques* using OpenCV – Python. GeeksforGeeks.org. https://www.geeksforgeeks.org/image-enhancement-techniques-using-opencv-python/

Schuck, J. (2020a, July 12). *Enhance a Document Scan using Python and OpenCV*. Medium.com. https://medium.com/analytics-vidhya/enhance-a-document-scan-using-python-and-opencv-9934a0c2da3d

Schuck, J. (2020b, July 12). *OpenCV-with-Python-Series/LICENSE*. GitHub.com. https://github.com/joschuck/OpenCV-with-Python-Series/blob/master/LICENSE

zaidkhan15. (2024, January 31). *Introduction to Levenshtein distance*. GeeksforGeeks.org. https://www.geeksforgeeks.org/introduction-to-levenshtein-distance/