**Image Processing Project**

**Pencil quality evaluation**

**Student: Ilya Yaverbaum**

**Id: 324516673**

**Date: 25.09.2023**

**Table of content:**

Introduction………………………………………………………………………………………….3

Block diagram…………….………….…………………….………………………………………4

Chosen parameters………….…………………………………………………....…………...5

Feature examining………………………………………………………………………………15

Thresholds……………………………………………………………………………….…………22

Methodology………………….………………………………………………………………….23

Algorithm………………………….……………………………………………………………….25

Results and limitations……………….………………………………….……………………27

Validation and results……………………….……………………………………………….29

Future steps……………………………………………………………………………………….30

Appendices…………………………………………………………………………………………32

**Introduction**

In a world increasingly reliant on digital technology, the humble pencil remains a steadfast tool for expression and creativity. Whether used for sketching, note-taking, or technical drawing, the quality of a pencil's tip significantly influences the user's experience.

This project explores the intersection of traditional writing instruments and modern image processing techniques. Its primary goal is to develop an image processing system capable of evaluating pencil tip quality through the analysis of pencil tip images. By extracting key parameters, such as tip sharpness and shape, we aim to assess the overall condition of the pencil's tip.

Leveraging advancements in computer vision and image analysis, our approach offers a non-invasive and efficient method for assessing pencil tip quality using digital cameras or smartphone cameras. This technology has practical applications in quality control for manufacturers, aids educators in assessing pencil usability, and benefits artists and professionals in selecting the best writing or drawing instrument.

Throughout this project, we will delve into image processing methods, exploring various algorithms to quantify pencil tip quality accurately. We will also develop a user-friendly app for feedback on tip condition.

Our aim is to deliver a reliable system for evaluating pencil tip quality via image processing, bridging the gap between traditional writing instruments and modern technology, ultimately enhancing the writing, and drawing experience.

**Block diagram**

**Picture of pencil**

Load classifier

**Predict**

**Load classifier and predict**

Turn to grey scale

correction

Erosion/dilation/closing

**Enhance picture.**

Find corners/edges of tip

Calculate angle of tip

Calculate area of tip

Calculate sum of corners of tip

**Extruct features from image**

Notice that the same process was dedicated to train the classifier.

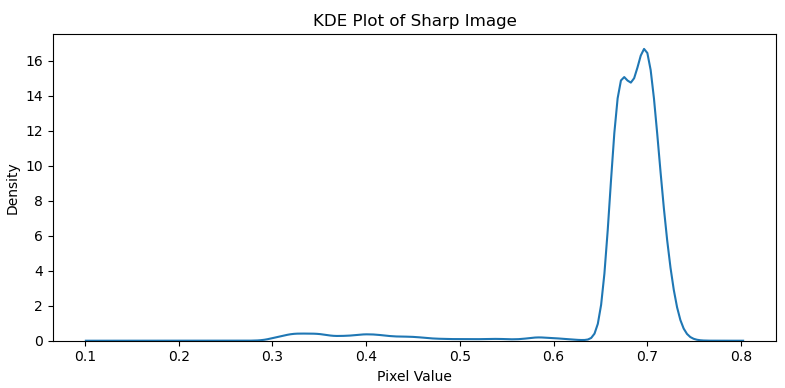
**Parameters:**

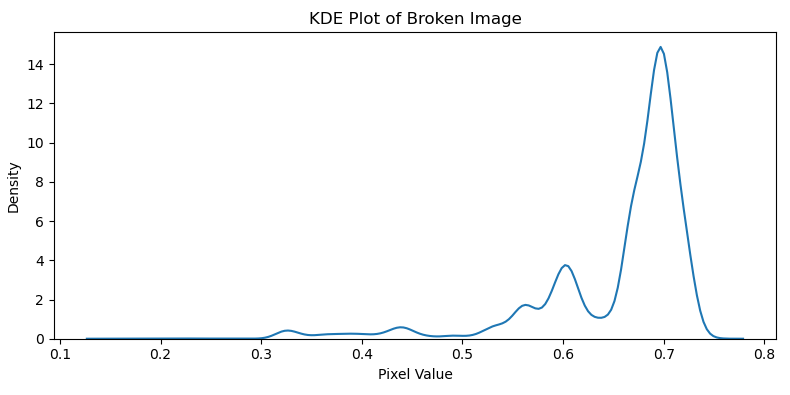
In process of development, I tried different approaches in order to extruct the most information saturated features, that can provide us with the decition making about the pencil condition.

Our goal is to get picture of pencil and by observing only the critical part of it - the tip , to determine pencils conditiom. In other words, we want to know if the pencil is sharp or broken.

For example, in the following picture, we have two pencils. One of them is sharp, and the other one is broken.  
The human eye can easyly detect the optimal and the worst cases. But how do we build the algorithm which can can such decisions?



Destribiotion of grey scale:



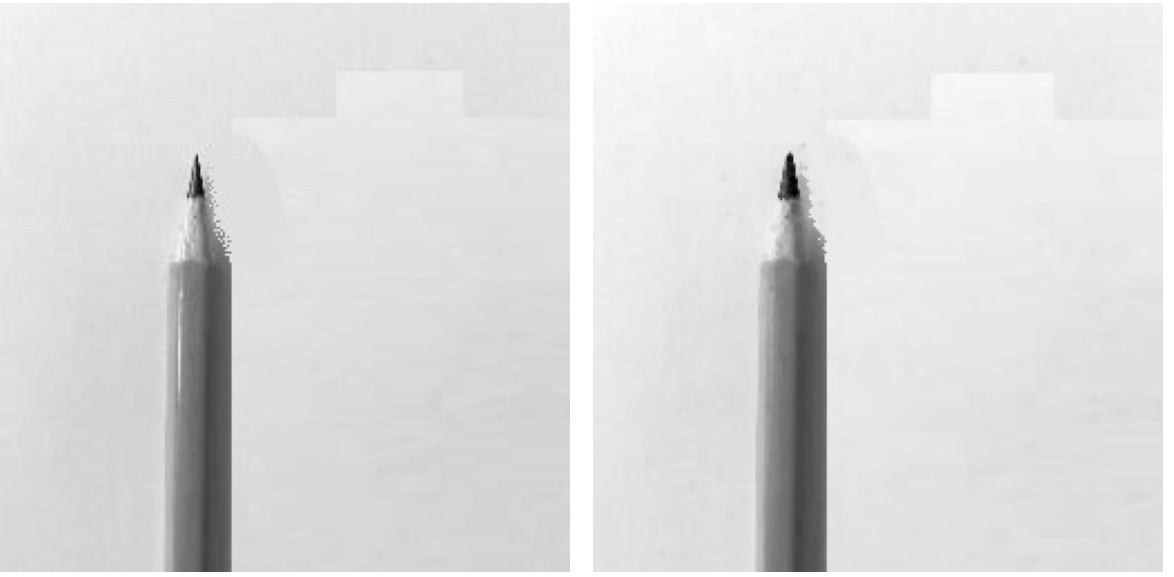
We may notice that most of the pixels are tend to the white color, while we can see small portion of the pixels concetrates in the darker area (close to 0).

**First step: Filters examining**  
We need to isolate the tip of the pencil in order to apply feature extruction.  
We want to use image processing methods and algorithms in order to fetch the target and study which features we may use.  
We will cover some basic mwthods and see how they affect our image, and how does it gets us closer to the wanted result.  
In the next examples the image already passed initial preprocessing : size adaptation, and grey scaling. We will compare sharp and broken instances of a pencil when applying different morphilogical operators.

* Notice that the left images are the original and right are filtered.

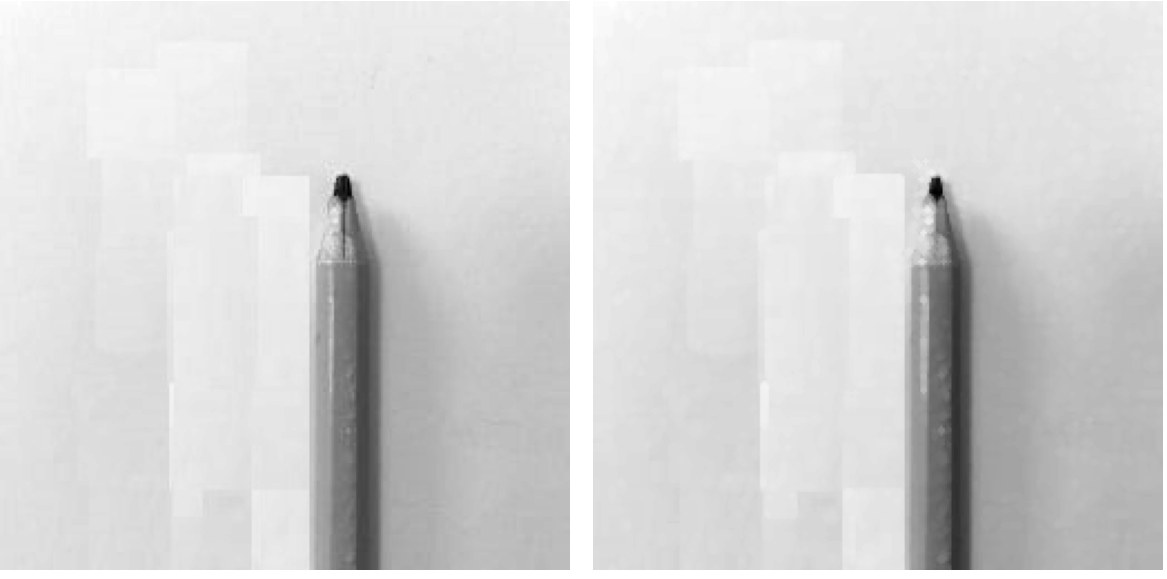
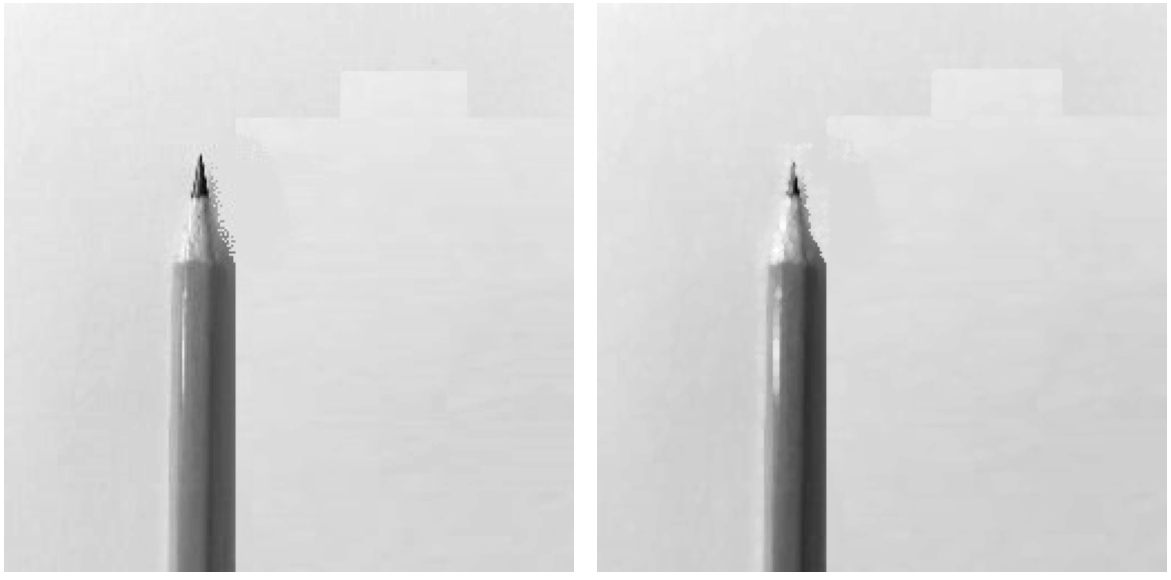
Erosion:

A close-up of a pencil

Description automatically generated

In the images above we see that Erosion shrinks bright regions and enlarges dark regions. It is necessary to apply this method to enhance the wanted area – the tip of the pencil which is supposed to be the darkest pixels in the picture.

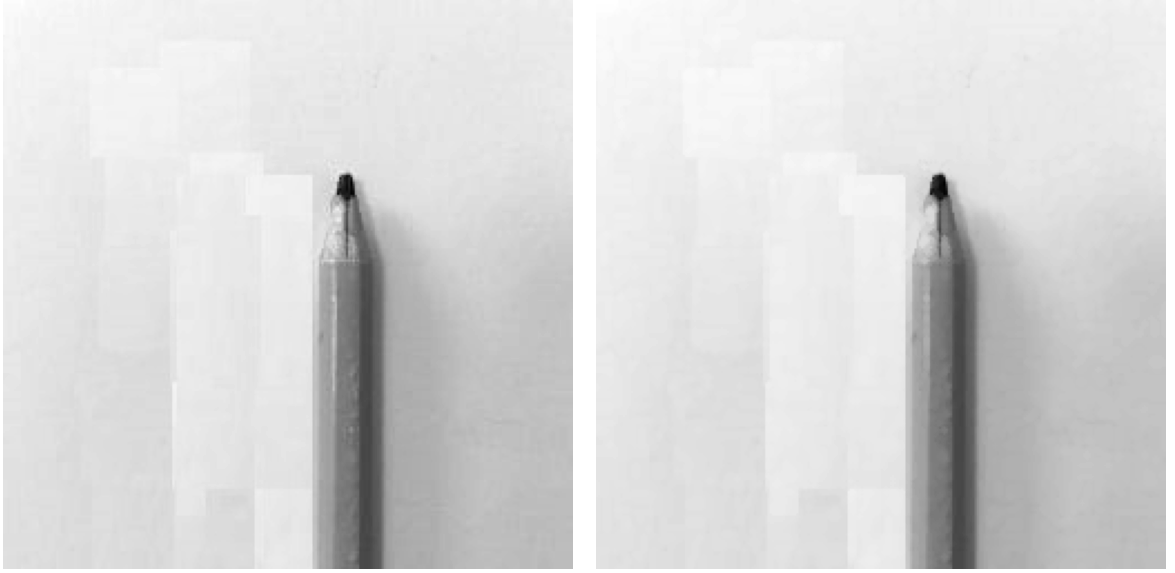
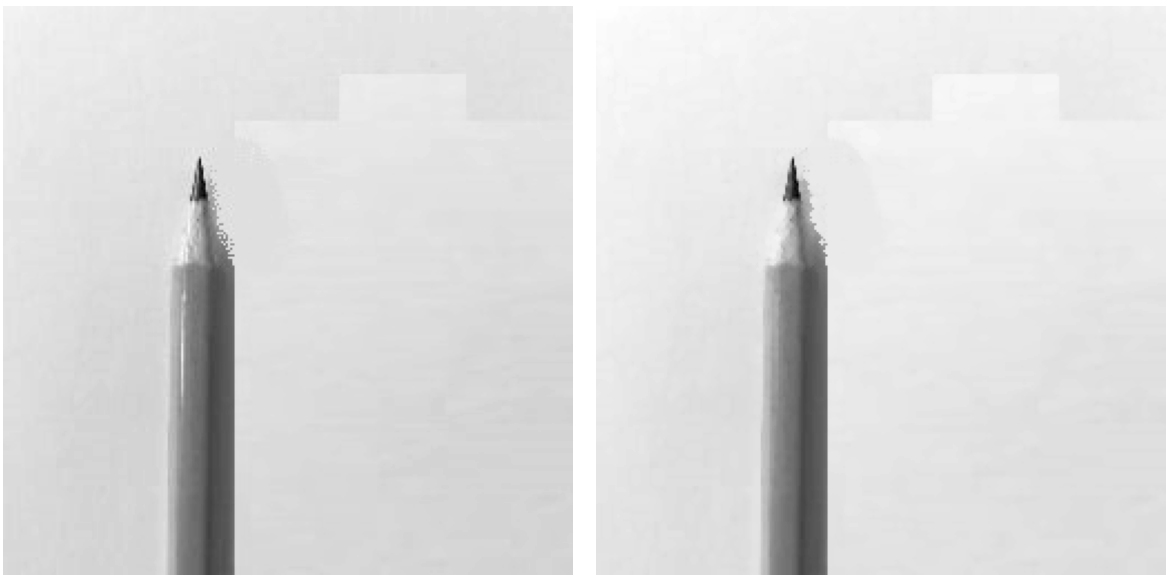
Dilation:



Morphological dilation sets the value of a pixel to the maximum over all pixel values within a local neighborhood centered about it. The values where the footprint is 1 define this neighborhood. Dilation enlarges bright regions and shrinks dark regions.

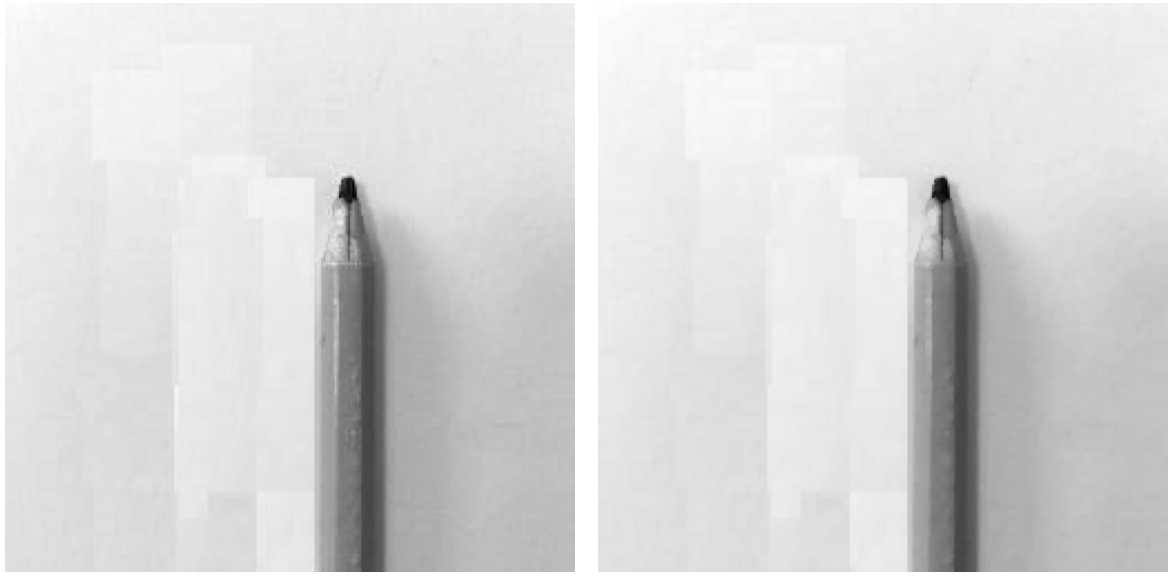
This method helps us to remove unwanted dark areas and pixels that are product of bad data compressing in the preprocessing stage.

Erosion + Dilation:

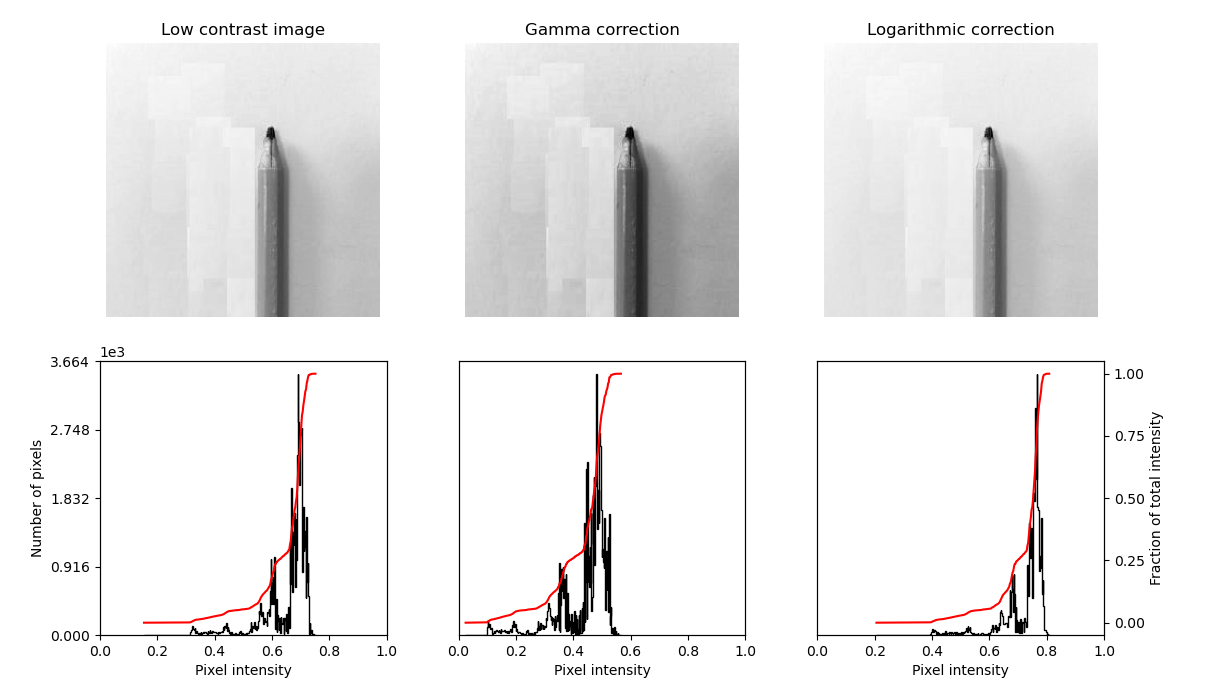
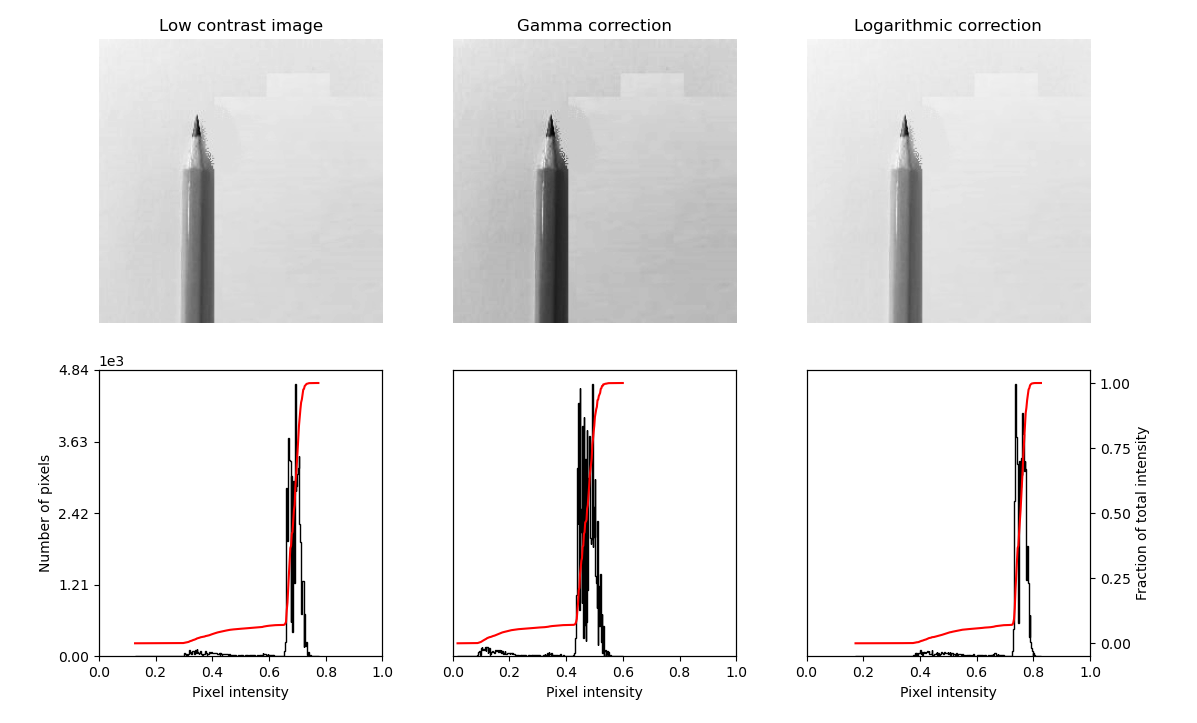


In the images above we combine the result of erosion and dilation sequencially. (Opening)  
We may notice that the combination of the two resulted in deleating luminance reflection from mirror like surfaces of the pencils.  
Also, smoothing the edges between dark and bright areas.

Opening:

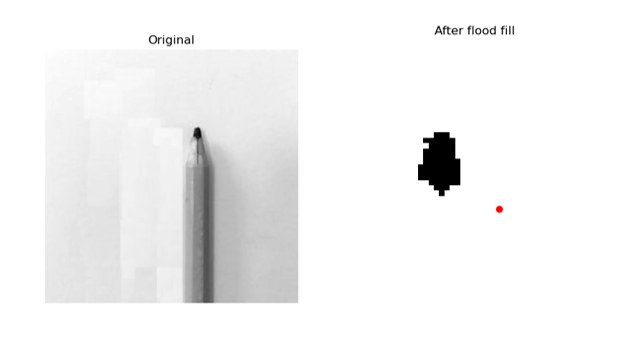
  
The morphological opening of an image is defined as an erosion followed by a dilation. Opening can remove small bright spots (i.e., “salt”) and connect small dark cracks. This tends to “open” up (dark) gaps between (bright) features.  
Also Closing method has been tested:  
The morphological closing of an image is defined as a dilation followed by an erosion. Closing can remove small dark spots (i.e. “pepper”) and connect small bright cracks. This tends to “close” up (dark) gaps between (bright) features.

Corections:

By experimenting with different types of image corections we studied the range of luminance that can be changed in the image.  
The motivation was to get the bect color range in order to distinguish between the black tip of a pencil and the rest of the image.

The logorithmic correction shows the best result in the seperation.

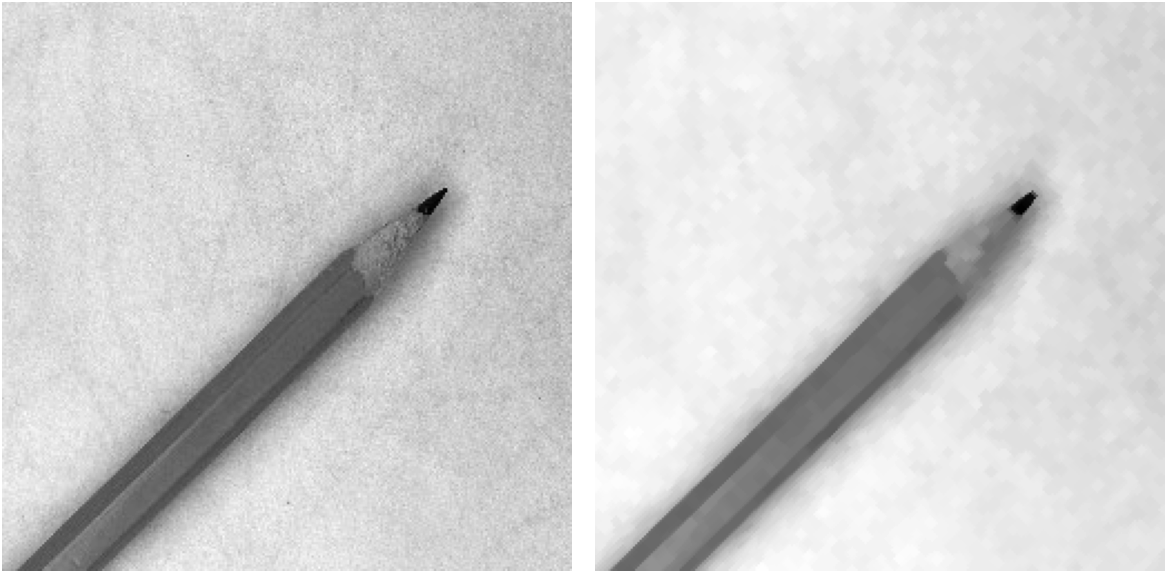
Flood fill:



Another method that has been examined is the Flood Fill.  
Starting at a specific seed\_point, connected points equal or within tolerance of the seed value are found, then set to new\_value.

There were more examining of different parameters in the spehre of morphological operations, for example:

Different type of interpolation by changing window shape, and size:

 A close-up of a pencil

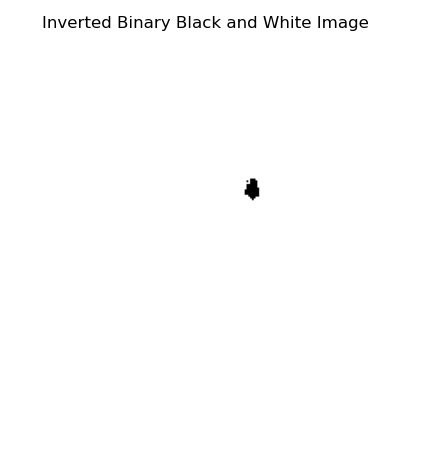
Description automatically generated

More aggressive types of footprints (shape and size) affected very much the image.  
on one hand we got better noise reduction as we enlarged the window size , but on the other hand, we deffected the tip of the pencil, which we cant afford to change for optimal results.

The chalange of manipulationg the image without distortion of the tip of the pencil pushed me to rely more on feature selection, than the preprocessing in order to receive better image.

Invertion method:

A white background with black text

Description automatically generated

**Second step: Feature examining**

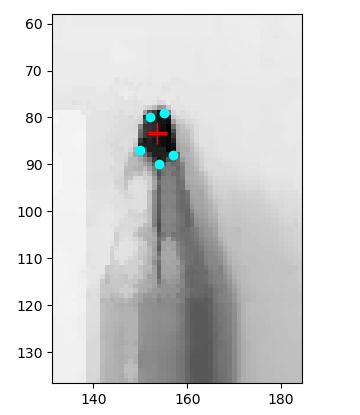
After diving into long research work, experimenting with different types of image processing methods and tools, we found the most fitting thresholds and methods that gives us sutisfiyng result.  
Many different methods has been tested withount good result, so we sticked to best preforming methods described above: Erosion, Dilation, Closing and Corection.

In this stage we will cover the most informational features that can be fetched from the image we just processed.

When observing the tip of the pencil the first thing that we expect to evaluate is the sharpnes of the tip which can be reduced to a 2D world as the Angle of the cross section of the tip (The profile of the tip).

This measurment requires us to transform the picture to geometric shape.

**Corners: corner\_peaks**

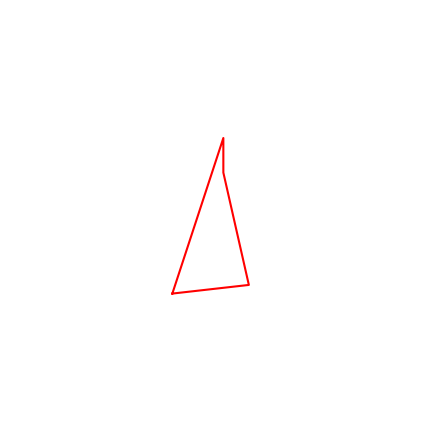
******:**

I have experimented with corner\_peak function and found it usefull.  
The function detects corner points using the Harris corner detector and determine the subpixel position of corners.  
Now we can work in geometrical realm, by turning the corners to a polygon.

****

A close-up of a graph

Description automatically generated



A red line drawing of a diamond

Description automatically generated

**First feature: Angle of the tip.**

In best case scenario the pencil is sharp, the tip has a shape of an isosceles triangle and the tip is the smallest angle in the shape.

Example of sharp tip:  
  
Example of broken tip:

**Second feature: Sum of angles**

In best case scenario the pencil is sharp, so the polygon that we get is a triangle (we receive only 3 corners).  
So the sum of the angles will always be 180 deg.

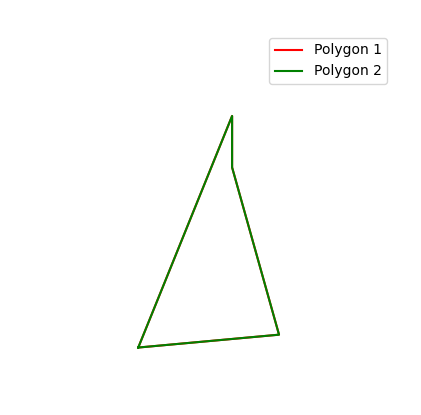
In less optimal cases the number of corners will be higher, which will get us bigger sum of angles.

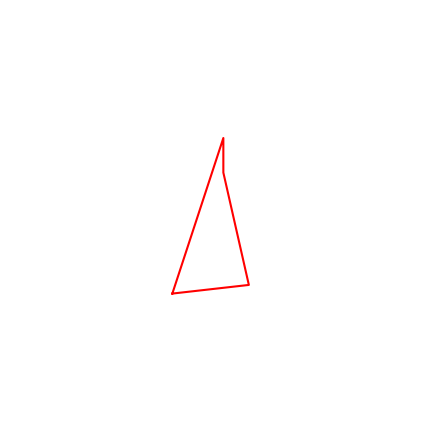
Example of sharp tip:

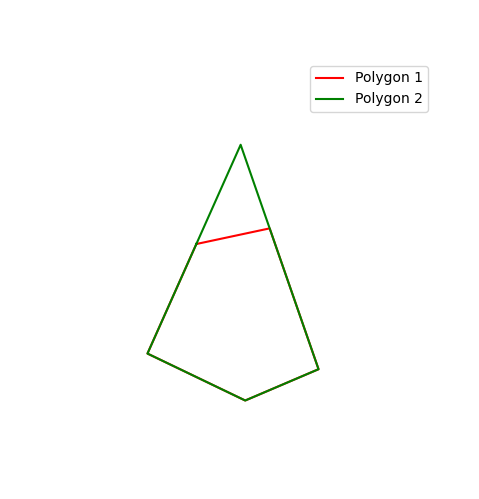
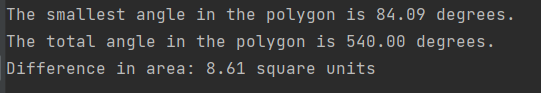
Example of broken tip:

**Third feature: Area difference**

In this feature we create “ideal” polygon by filtering the two longest edges of the polygon, which almost always represents the side edges of the tip. We calculate the point of intersaction of this two linnes. Using this new point of intersaction between the two lines we create a “Perfect” polygon in which the tip of the pencil is sharpest.  
By calculating the difference in the area of the original polygon and the “perfect” one, we are establishing the distance of the sharpness of the tipp from the hypothetical true maximum.

Example of sharp tip:

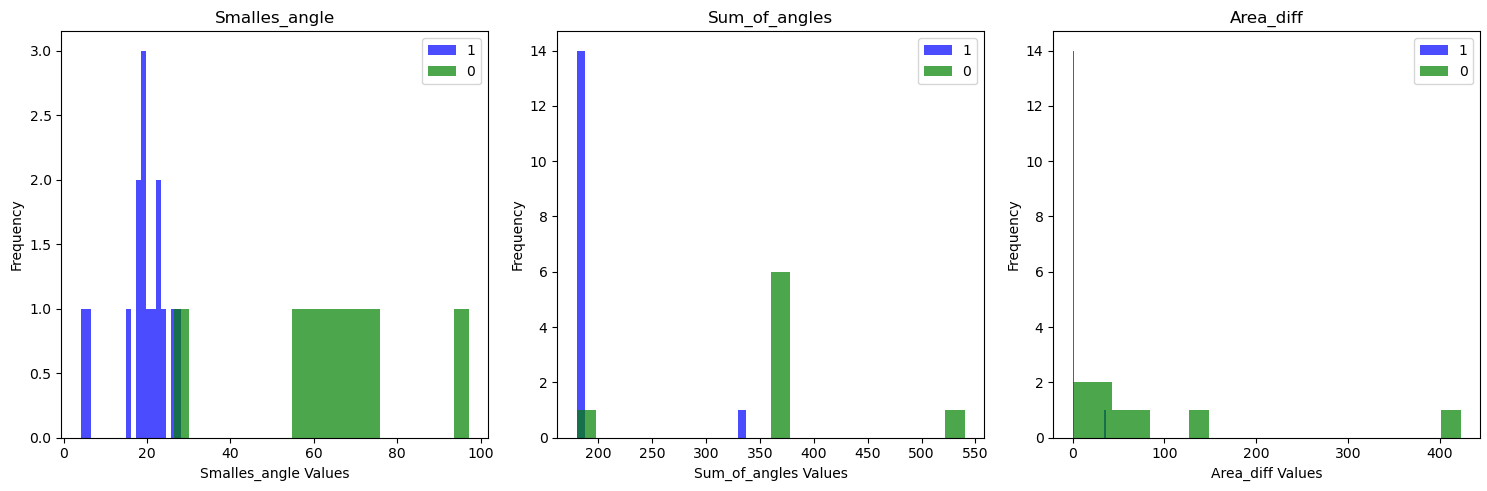


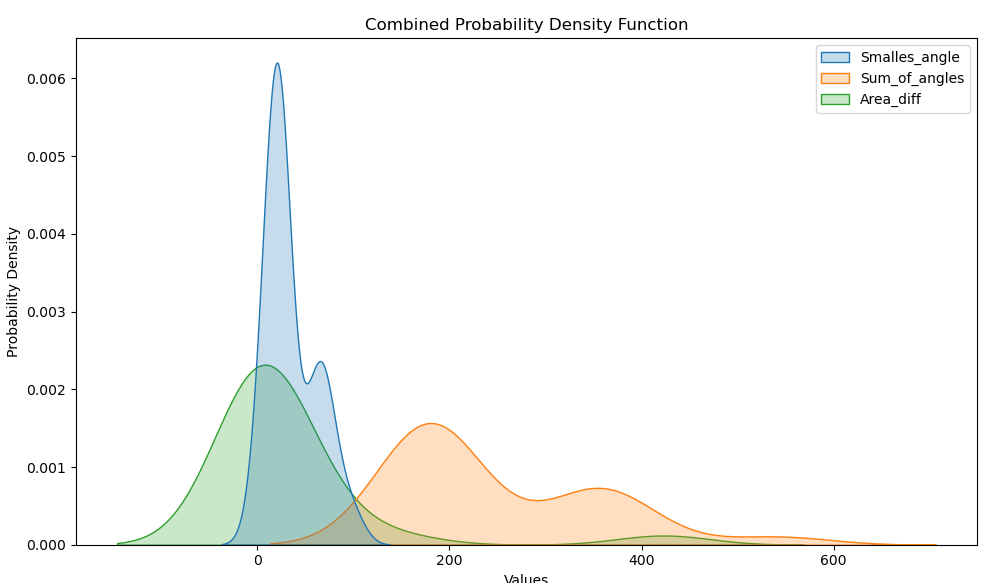
Example of broken tip:

A red line drawing of a diamond

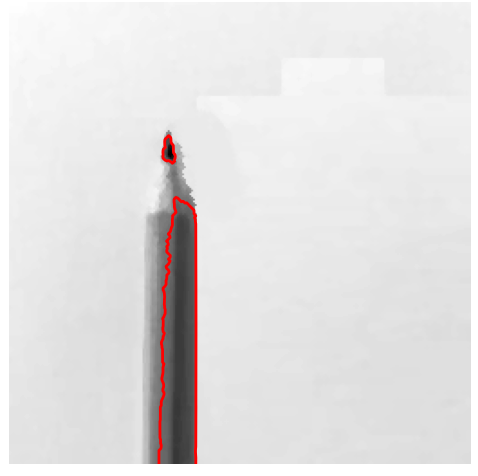
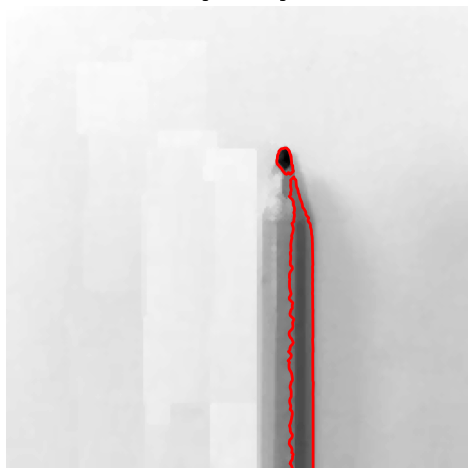
Description automatically generated

Insight on the feature distrebution:

Where 1 represents Sharp label, and 0 broken label.



There are more skimage library methods that we studied, but their implementation demanding more research time and deeper evaluation.  
We tried: **censure**, **morphological\_chan\_vese, find\_contours, sobel\_edge\_detector.**

Find\_contours: (sharp - left, broken - right)

Censure:

A close-up of a pen

Description automatically generated

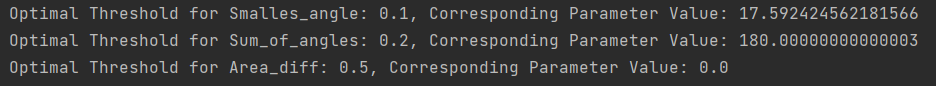
A black and white image of a tall tower

Description automatically generatedA close-up of a pencil

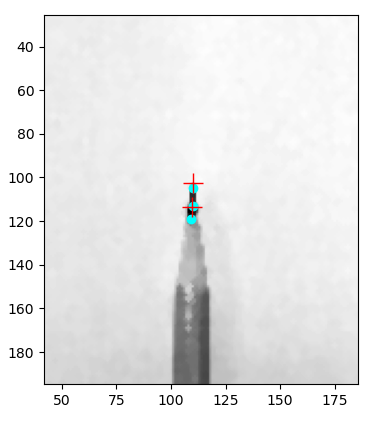
Description automatically generatedSobel\_edge\_detector:

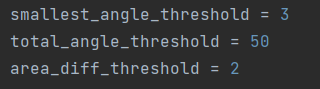
**Thresholds:**

The thresholds achieved by training a logistic regression model whith adaptation to print the optimal threshold values.

The output I received:

Example of bad thresholds:

The thresholds value changed. And the image evaluated by our own function, that check the featutres, and based on 2 of a 3 decition for classifying the image.



The output showed that the pencil is broke, althoug it is not true.

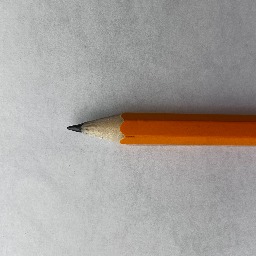
**Methodology**:

I built a logistic regression classifier to differentiate between sharp and broken pencils. Using a labeled dataset, I experimented with different thresholds to find the one that achieved the most accurate classification.

I used the images as versatile as posible, to not overfit some types of images.

Number of examples of labeled images that I used:

A pencil with a black tip

Description automatically generatedSharp pencils:

A close-up of a pencil

Description automatically generatedA pencil with a sharp tip

Description automatically generated

A pencil on a white surface

Description automatically generatedA close-up of a pencil

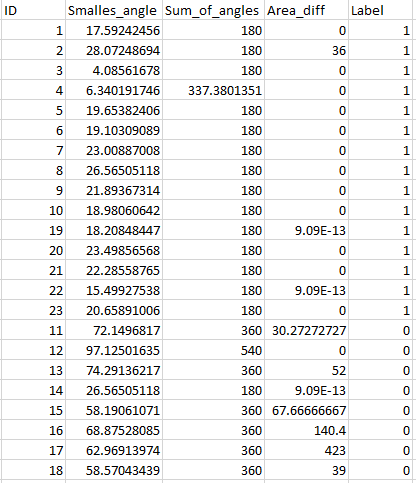
Description automatically generatedBroken pencils:

A pencil on a white surface

Description automatically generatedA pencil with a sharp tip

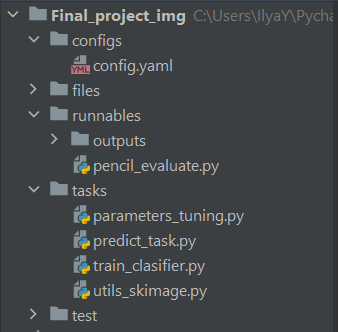
Description automatically generated

Then I built a DataFrame with different pencils, their extructed features, and the true labels.



The result stored as Pickle file, that will be used to predict the class of the input images.

**The Algorithm:**



The main task of the project is the **pencil\_evaluate** file.

pencil\_evaluate.

* Creates a Predict object. Using Cfg configuration yaml file.  
  The configuration file contains the path to the image we want to evaluate, and the path to the classifier.
* Runs Predict.run( ) method of Predict object.

predict\_task

- Initiates the Predict object

* Generates enhanced and preproccessed picture.
* Extracts features from the image.
* Evaluates the image as binary classification using trained classifier we described earlier.
* Has all the neccesary functions of skimage library, used to preform preprocessing on image.

utils\_skimage

* Has feature related methods.
* Using sklearn LogisticRegression classifier, we may fit and evaluate new classifiers based on new data.

train\_classifier

* The classifier studies labeled data and their features: 'Smallest\_angle', 'Sum\_of\_angles', 'Area\_diff'.

Load and Preprocess the Image

Apply Morphological Operators

Feature Extraction

Decision Making

Display Result

Libraries and infrastructure:

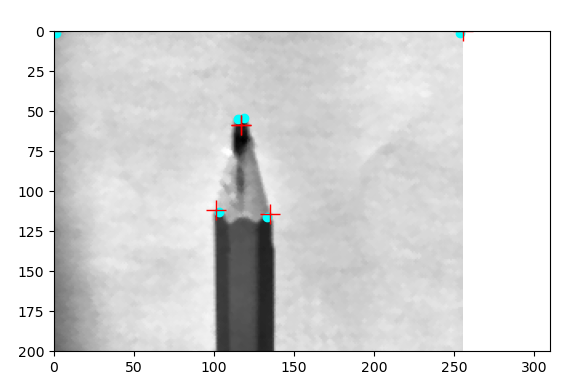
* Skimage: morphological operator, feature extraction and more.
* Sklearn: workinng with dataset. Train, test split. LogisticRegression and scores.
* Hydra: framework for the ability to dynamically create a hierarchical configuration.  
  This will be more usefull when the project will grow in dimentions, through the development.
* OOP. Using object-oriented programming, for more efficient and optimal development.

**Limitations:**

In the proccess of examinig different aspects, shading and angles of the object, I noticed that some changes don’t affect the pipeline of prediction, and some makes the prediction not feasible.

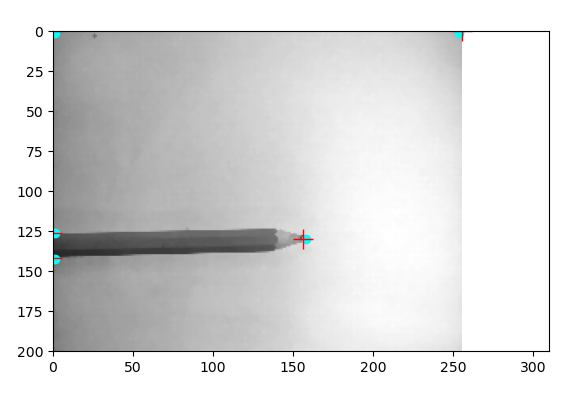
Examples:



* Bad lighting in image makes the tip of the pencil very difficult to marking because of all the other dark blocks on other areas of the pencil.  
    
  In some cases presence of shadow of the pencil is ruining the proccess.
* A graph of a pencil

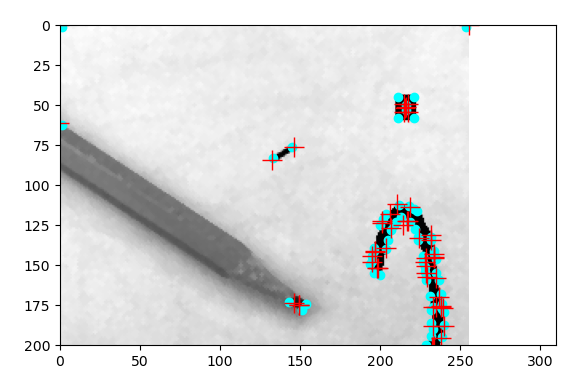
  Description automatically generatedBad quality of image:

Bad quality of image makes it hard to mark corners because of the very pixelized ‘image’.



* image taken from distance:

When the image is taking from a certain distance, the tip appears very small, and its is not possible to identify the polygon from it.



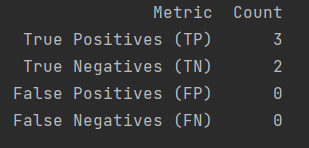
* Different objects on background  
  Here we simulated different types of interfeering in the background.  
  The presence of other objects making the tip observation not possible.

Standart enviroment for optimal result:

* The picture should be 256X256.
* The image must be taken from close distance with good quality camera.
* The background must be bright.
* The light must be strong, and to be directed such that there is no casting of a shadow, from the pencil.

Notice that there is no difference of the angle in which the pencil is shown.

**Validation and testing:**

After testing our thresholds on test data, conducted from selected mix of sharp and broken pencils, the results are:

We see that on this subset of data, the preproccessing, and the applyied filtering created a suitable condition for successful classifiyng.

**Future steps (theoretical in case of funding and human force):**

* Data validation:  
  Untill now, we had dependencies on good quality of image. And in case of bad interpretation of the filters, and failure, there is an output to the user, that indicates htat the image is bad.  
  But if the code isnt crushing and yet, we have image that doesn’t stands in the standarts of the input, the result wont be representing.  
  solution: we can built bad input image detector, based on other features, and thresholds.  
  Such input validation can be very usefull filter that comes right after the pre-processing.
* The result in the examples above only explains the flow of the task. The confusion matrix and the accuracy of the predictions can’t be proven with such small dataset.  
  The right step, is to create a big dataset, with many different images, and then deside the thresholds. The result should be more respresenting, as we chalenging our tool with a lot of different information.
* The parameters of the morphological filters had been chosen by my personal experimenting and trying different parameters, metrics, and thresholds.  
  In order to achieve better results, I would like to use the Optuna tool in python, in order to optimize the different parameters.  
  this tool gets range of examples (features and functions), and their results, and finds the most optimal set of parameters with this set of functions.
* There is a lot of different images proccessing tools out there. Only in the Skimage library I came across more than 20 object and feature detection algorithms.  
  I would like to invest more time in investigating more of them, adapting them to our ‘world’. Also, more time spending on the morphological operations would probably benefit our result.  
  Because of the time frames, I had only investigated few of them, but I am sure that I can achieve better results with more deep research.
* In the same topic, I would like to invest more time to read and research more articles, recent studies, and publications, to enrich my approach for this project.

**Appendices**

Config.yaml

image\_path: image\_path.jpg  
classifier\_path: classifier\_path.pkl

pencil\_evaluate

from omegaconf import DictConfig  
import hydra  
  
  
from tasks.predict\_task import Predict  
  
  
config\_path = r'C:\Users\IlyaY\PycharmProjects\Final\_project\_img\configs' # configuration path  
config\_name = r'config.yaml' # configuration name  
  
  
@hydra.main(version\_base=None, config\_path=config\_path, config\_name=config\_name)  
def my\_app(cfg: DictConfig):  
 predicter = Predict(cfg) # Initiate object of prediction  
 try:  
 result = predicter.run() # run method that classifies the pencil.  
 except:  
 print('Bad example') # in case that one of the processes crashed, print error.  
 result = 'Error'  
  
 print(result)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 my\_app() # run the app

predict\_task

from skimage import io, color  
from skimage.morphology import disk, diamond, square, ball # noqa  
import joblib  
import warnings  
from omegaconf import DictConfig  
  
  
from tasks.utils\_skimage import corrections, erosion\_, dilation\_, closing\_, find\_corner, create\_polygon, angle\_calc,\  
 calculate\_total\_polygon\_angle, create\_new\_polygon, show\_and\_compare\_polygons, show\_images  
  
  
# Suppress all warnings  
warnings.filterwarnings("ignore")  
  
  
class Predict:  
 def \_\_init\_\_(self, cfg: DictConfig):  
 self.cfg = cfg  
 self.image\_path = self.cfg.image\_path # get the image path from configuration  
 self.classifier\_path = self.cfg.classifier\_path # get the model path from configuration  
  
 self.smallest\_angle\_threshold = 17.6 # thresholds for manual classifying  
 self.total\_angle\_threshold = 180  
 self.area\_diff\_threshold = 0.1  
  
 def predict(self, smallest\_angle, total\_angle, area\_diff):  
  
 loaded\_classifier = joblib.load(self.classifier\_path) # Load the trained classifier from a file  
  
 new\_example = [[smallest\_angle, total\_angle, area\_diff]] # Example data for prediction (should have 3 features)  
  
 # Make predictions on the new example  
 predicted\_class = loaded\_classifier.predict(new\_example) # If it's a binary classification problem (0 or 1), predicted\_class will contain the predicted class label (0 or 1).  
 # predicted\_proba will contain the probability of belonging to class 1.  
 predicted\_proba = loaded\_classifier.predict\_proba(new\_example)  
  
 print(f"Predicted Class: {predicted\_class[0]}")  
  
 return predicted\_class[0]  
  
 def enhance\_picture(self):  
 # Load the image  
 image = io.imread(self.image\_path)  
 image = color.rgb2gray(image)  
  
 logarithmic\_corrected = corrections(image) # make correction for black enhancement  
 result\_image = erosion\_(logarithmic\_corrected) # get rid of noise and correct the edges  
 result\_image = dilation\_(result\_image)  
 result\_image = closing\_(result\_image)  
  
 return result\_image, image  
  
 def get\_features(self, result\_image):  
 pos = find\_corner(result\_image) # find the corners of our object - the tip of the pencil  
  
 # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] >= 20 and pair[1] >= 20] # sometimes the corners set to the picture corner, so we filter them  
 # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] <= 230 and pair[1] <= 230]  
  
 polygon = create\_polygon(pos) # create polygon based on the corners  
  
 smallest\_angle = angle\_calc(polygon) # calculate the feature  
 print(f"The smallest angle in the polygon is {smallest\_angle:.2f} degrees.")  
 total\_angle = calculate\_total\_polygon\_angle(polygon) # calculate the feature  
 print(f"The total angle in the polygon is {total\_angle:.2f} degrees.")  
 new\_polygon = create\_new\_polygon(polygon) # create perfect polygon for comparison  
 area\_diff = show\_and\_compare\_polygons(polygon, new\_polygon) # calculate the feature  
  
 return smallest\_angle, total\_angle, area\_diff  
  
 def prediction\_by\_threshold(self, smallest\_angle, total\_angle, area\_diff):  
 smallest\_angle\_threshold = 3  
 total\_angle\_threshold = 50  
 area\_diff\_threshold = 2  
  
 result = 0  
  
 if smallest\_angle < smallest\_angle\_threshold:  
 result = result + 1  
 if total\_angle < total\_angle\_threshold:  
 result = result + 1  
 if area\_diff < area\_diff\_threshold:  
 result = result + 1  
  
 if result >= 2:  
 return 1  
 else:  
 return 0  
  
 def run(self,):  
 result\_image, image = self.enhance\_picture() # preprocessing  
  
 smallest\_angle, total\_angle, area\_diff = self.get\_features(result\_image) # fetch the features  
  
 show\_images(image, result\_image)  
  
 prediction = self.predict(smallest\_angle, total\_angle, area\_diff) # return the prediction of the model on example  
 # prediction = self.prediction\_by\_threshold(smallest\_angle, total\_angle, area\_diff) # manual threshold  
  
 if prediction == 1: # translate the prediction to corresponding string  
 return 'sharp'  
 else:  
 return 'broken'

utils\_skimage

from skimage import io, color, feature, morphology, exposure, measure, filters  
import matplotlib.pyplot as plt  
import numpy as np  
from skimage.morphology import (erosion, dilation, opening, closing, # noqa  
 white\_tophat)  
from skimage.morphology import black\_tophat, skeletonize, convex\_hull\_image # noqa  
from skimage.morphology import disk, diamond, square, ball # noqa  
from skimage.segmentation import chan\_vese  
from skimage import feature  
from skimage import data, img\_as\_float  
from skimage import data  
from skimage import transform  
from skimage.feature import CENSURE  
from skimage.feature import corner\_harris, corner\_subpix, corner\_peaks  
from skimage.segmentation import flood, flood\_fill  
from skimage.segmentation import (morphological\_chan\_vese,  
 morphological\_geodesic\_active\_contour,  
 inverse\_gaussian\_gradient,  
 checkerboard\_level\_set)  
from skimage import exposure  
  
  
def show\_images(image, result\_image):  
  
 # Display both the original and final images side by side  
 plt.figure(figsize=(12, 6))  
  
 # Original image  
 plt.subplot(121)  
 plt.imshow(image, cmap=plt.cm.gray)  
 plt.title('Original Image')  
 plt.axis('off')  
  
 # Final image with only darkest pixels retained  
 plt.subplot(122)  
 plt.imshow(result\_image, cmap=plt.cm.gray)  
 plt.title('Darkest Pixels Retained, Others Turned White')  
 plt.axis('off')  
  
 plt.tight\_layout()  
 # plt.show()  
  
  
def erosion\_(image):  
 footprint = disk(1) # square etc...  
 eroded = erosion(image, footprint)  
 return eroded  
  
  
def dilation\_(image):  
 footprint = disk(1) # square etc...  
 dilated = dilation(image, footprint)  
 return dilated  
  
  
def opening\_(image):  
 footprint = disk(1) # square etc...  
 opened = opening(image, footprint)  
 return opened  
  
  
def closing\_(image):  
 footprint = disk(1) # square etc...  
 closed = closing(image, footprint)  
 return closed  
  
  
def black\_tophat\_(image):  
 footprint = diamond(3) # square etc...  
 b\_tophat = black\_tophat(image, footprint)  
 return b\_tophat  
  
  
def chan\_vese\_(image):  
 cv = chan\_vese(image, mu=0.5, lambda1=1, lambda2=1, tol=1e-3,  
 max\_num\_iter=30, dt=0.5, init\_level\_set="checkerboard",  
 extended\_output=True)  
  
 fig, axes = plt.subplots(2, 2, figsize=(8, 8))  
 ax = axes.flatten()  
  
 ax[0].imshow(image, cmap="gray")  
 ax[0].set\_axis\_off()  
 ax[0].set\_title("Original Image", fontsize=12)  
  
 ax[1].imshow(cv[0], cmap="gray")  
 ax[1].set\_axis\_off()  
 title = f'Chan-Vese segmentation - {len(cv[2])} iterations'  
 ax[1].set\_title(title, fontsize=12)  
  
 ax[2].imshow(cv[1], cmap="gray")  
 ax[2].set\_axis\_off()  
 ax[2].set\_title("Final Level Set", fontsize=12)  
  
 ax[3].plot(cv[2])  
 ax[3].set\_title("Evolution of energy over iterations", fontsize=12)  
  
 fig.tight\_layout()  
 # plt.show()  
  
  
def plot\_img\_and\_hist(image, axes, bins=256):  
 *"""Plot an image along with its histogram and cumulative histogram.  
  
 """* image = img\_as\_float(image)  
 ax\_img, ax\_hist = axes  
 ax\_cdf = ax\_hist.twinx()  
  
 # Display image  
 ax\_img.imshow(image, cmap=plt.cm.gray)  
 ax\_img.set\_axis\_off()  
  
 # Display histogram  
 ax\_hist.hist(image.ravel(), bins=bins, histtype='step', color='black')  
 ax\_hist.ticklabel\_format(axis='y', style='scientific', scilimits=(0, 0))  
 ax\_hist.set\_xlabel('Pixel intensity')  
 ax\_hist.set\_xlim(0, 1)  
 ax\_hist.set\_yticks([])  
  
 # Display cumulative distribution  
 img\_cdf, bins = exposure.cumulative\_distribution(image, bins)  
 ax\_cdf.plot(bins, img\_cdf, 'r')  
 ax\_cdf.set\_yticks([])  
  
 return ax\_img, ax\_hist, ax\_cdf  
  
  
def equalize(img):  
 # Adaptive Equalization  
 img\_adapteq = exposure.equalize\_adapthist(img, clip\_limit=0.03) # 0.03  
  
 return img\_adapteq  
  
def corrections(img):  
  
 gamma\_corrected = exposure.adjust\_gamma(img, 0.4) # Gamma  
 logarithmic\_corrected = exposure.adjust\_log(img, 1) # Logarithmic  
  
 return logarithmic\_corrected  
  
  
def canny\_(image):  
 edges = feature.canny(image, sigma=3)  
 return edges  
  
  
def sobel\_edge\_detector(image):  
 edge\_sobel = filters.roberts(image)  
 return edge\_sobel  
  
  
def find\_contour(image):  
 # Convert the image to grayscale if it's in color  
 if image.shape[-1] == 3:  
 gray\_image = color.rgb2gray(image)  
 else:  
 gray\_image = image  
  
 # Find contours in the grayscale image  
 contours = measure.find\_contours(gray\_image, 0.5) # Adjust the threshold as needed  
  
 # Display the original image  
 plt.figure(figsize=(8, 6))  
 plt.imshow(image, cmap='gray')  
 plt.title('Original Image')  
 plt.axis('off')  
  
 # Plot the detected contours  
 for contour in contours:  
 plt.plot(contour[:, 1], contour[:, 0], linewidth=2, c='r')  
  
 # plt.show()  
 return contours  
  
  
def store\_evolution\_in(lst):  
 *"""Returns a callback function to store the evolution of the level sets in  
 the given list.  
 """* def \_store(x):  
 lst.append(np.copy(x))  
  
 return \_store  
  
  
def morph\_acwe(image):  
 # Morphological ACWE  
  
 # Initial level set  
 init\_ls = checkerboard\_level\_set(image.shape, 6)  
 # List with intermediate results for plotting the evolution  
 evolution = []  
 callback = store\_evolution\_in(evolution)  
 ls = morphological\_chan\_vese(image, num\_iter=35, init\_level\_set=init\_ls,  
 smoothing=3, iter\_callback=callback)  
  
 fig, axes = plt.subplots(2, 2, figsize=(8, 8))  
 ax = axes.flatten()  
  
 ax[0].imshow(image, cmap="gray")  
 ax[0].set\_axis\_off()  
 ax[0].contour(ls, [0.5], colors='r')  
 ax[0].set\_title("Morphological ACWE segmentation", fontsize=12)  
  
 ax[1].imshow(ls, cmap="gray")  
 ax[1].set\_axis\_off()  
 contour = ax[1].contour(evolution[2], [0.5], colors='g')  
 contour.collections[0].set\_label("Iteration 2")  
 contour = ax[1].contour(evolution[7], [0.5], colors='y')  
 contour.collections[0].set\_label("Iteration 7")  
 contour = ax[1].contour(evolution[-1], [0.5], colors='r')  
 contour.collections[0].set\_label("Iteration 35")  
 ax[1].legend(loc="upper right")  
 title = "Morphological ACWE evolution"  
 ax[1].set\_title(title, fontsize=12)  
  
  
 # Morphological GAC  
 image = img\_as\_float(data.coins())  
 gimage = inverse\_gaussian\_gradient(image)  
  
 # Initial level set  
 init\_ls = np.zeros(image.shape, dtype=np.int8)  
 init\_ls[10:-10, 10:-10] = 1  
 # List with intermediate results for plotting the evolution  
 evolution = []  
 callback = store\_evolution\_in(evolution)  
 ls = morphological\_geodesic\_active\_contour(gimage, num\_iter=230,  
 init\_level\_set=init\_ls,  
 smoothing=1, balloon=-1,  
 threshold=0.69,  
 iter\_callback=callback)  
  
 ax[2].imshow(image, cmap="gray")  
 ax[2].set\_axis\_off()  
 ax[2].contour(ls, [0.5], colors='r')  
 ax[2].set\_title("Morphological GAC segmentation", fontsize=12)  
  
 ax[3].imshow(ls, cmap="gray")  
 ax[3].set\_axis\_off()  
 contour = ax[3].contour(evolution[0], [0.5], colors='g')  
 contour.collections[0].set\_label("Iteration 0")  
 contour = ax[3].contour(evolution[100], [0.5], colors='y')  
 contour.collections[0].set\_label("Iteration 100")  
 contour = ax[3].contour(evolution[-1], [0.5], colors='r')  
 contour.collections[0].set\_label("Iteration 230")  
 ax[3].legend(loc="upper right")  
 title = "Morphological GAC evolution"  
 ax[3].set\_title(title, fontsize=12)  
  
 fig.tight\_layout()  
 # plt.show()  
  
  
def find\_corner(image):  
 coords = corner\_peaks(corner\_harris(image), min\_distance=1, threshold\_rel=0.01) # min\_distance=5, threshold\_rel=0.02  
 coords\_subpix = corner\_subpix(image, coords, window\_size=13) # window\_size= 13  
  
 fig, ax = plt.subplots()  
 ax.imshow(image, cmap=plt.cm.gray)  
 ax.plot(coords[:, 1], coords[:, 0], color='cyan', marker='o',  
 linestyle='None', markersize=6)  
 ax.plot(coords\_subpix[:, 1], coords\_subpix[:, 0], '+r', markersize=15)  
 ax.axis((0, 310, 200, 0))  
 plt.show()  
  
 return coords  
  
  
def censure(image):  
 tform = transform.AffineTransform(scale=(1.5, 1.5), rotation=0.1,  
 translation=(150, -200))  
 img\_warp = transform.warp(image, tform)  
  
 detector = CENSURE()  
  
 fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))  
  
 detector.detect(image)  
  
 ax[0].imshow(image, cmap=plt.cm.gray)  
 ax[0].scatter(detector.keypoints[:, 1], detector.keypoints[:, 0],  
 2 \*\* detector.scales, facecolors='none', edgecolors='r')  
 ax[0].set\_title("Original Image")  
  
 detector.detect(img\_warp)  
  
 ax[1].imshow(img\_warp, cmap=plt.cm.gray)  
 ax[1].scatter(detector.keypoints[:, 1], detector.keypoints[:, 0],  
 2 \*\* detector.scales, facecolors='none', edgecolors='r')  
 ax[1].set\_title('Transformed Image')  
  
 for a in ax:  
 a.axis('off')  
  
 plt.tight\_layout()  
 # plt.show()  
  
 return detector.keypoints  
  
  
def flood\_fill\_(image, pos):  
 pairs\_array = np.array(pos)  
  
 # Calculate the mean pair  
 mean\_pair = np.mean(pairs\_array, axis=0)  
 mean\_pair = np.round(mean\_pair).astype(int)  
  
 # light\_coat = flood\_fill(image, tuple(pos[0]), 255, tolerance=0.4)  
 # light\_coat = flood\_fill(image, tuple(pos[0]), 255, tolerance=0.1)  
 light\_coat = flood\_fill(image, tuple(mean\_pair), 255, tolerance=0.1)  
 fig, ax = plt.subplots(ncols=2, figsize=(10, 5))  
  
 ax[0].imshow(image, cmap=plt.cm.gray)  
 ax[0].set\_title('Original')  
 ax[0].axis('off')  
  
 ax[1].imshow(light\_coat, cmap=plt.cm.gray)  
 ax[1].plot(tuple(mean\_pair)[0], tuple(mean\_pair)[1], 'ro') # seed point  
 ax[1].set\_title('After flood fill')  
 ax[1].axis('off')  
  
 # plt.show()  
  
  
def angle\_calc(polygon):  
 *"""  
 Calculate the smallest angle in a polygon.  
  
 Parameters:  
 polygon (list of tuples): List of (x, y) coordinates representing the polygon vertices.  
  
 Returns:  
 float: The smallest angle in degrees.  
 """* if len(polygon) < 3:  
 return None # A polygon must have at least 3 vertices  
  
 smallest\_angle = float('inf')  
  
 for i in range(len(polygon)):  
 # Get three consecutive vertices  
 prev\_vertex = polygon[i - 1]  
 current\_vertex = polygon[i]  
 next\_vertex = polygon[(i + 1) % len(polygon)] # Wrap around for the last vertex  
  
 # Calculate vectors from the current vertex to the previous and next vertices  
 vector1 = np.array(prev\_vertex) - np.array(current\_vertex)  
 vector2 = np.array(next\_vertex) - np.array(current\_vertex)  
  
 # Calculate the angle between the two vectors using the dot product  
 angle\_rad = np.arccos(np.dot(vector1, vector2) / (np.linalg.norm(vector1) \* np.linalg.norm(vector2)))  
 angle\_deg = np.degrees(angle\_rad)  
  
 # Update the smallest angle if necessary  
 if angle\_deg < smallest\_angle:  
 smallest\_angle = angle\_deg  
  
 return smallest\_angle  
  
  
def calculate\_total\_polygon\_angle(polygon):  
 *"""  
 Calculate the total angle in degrees of a polygon.  
  
 Parameters:  
 polygon (list of tuples): List of (x, y) coordinates representing the polygon vertices.  
  
 Returns:  
 float: The total angle in degrees.  
 """* if len(polygon) < 3:  
 return None # A polygon must have at least 3 vertices  
  
 total\_angle = 0.0  
  
 for i in range(len(polygon)):  
 # Get three consecutive vertices  
 prev\_vertex = polygon[i - 1]  
 current\_vertex = polygon[i]  
 next\_vertex = polygon[(i + 1) % len(polygon)] # Wrap around for the last vertex  
  
 # Calculate vectors from the current vertex to the previous and next vertices  
 vector1 = np.array(prev\_vertex) - np.array(current\_vertex)  
 vector2 = np.array(next\_vertex) - np.array(current\_vertex)  
  
 # Calculate the angle between the two vectors using the dot product  
 angle\_rad = np.arccos(np.dot(vector1, vector2) / (np.linalg.norm(vector1) \* np.linalg.norm(vector2)))  
 angle\_deg = np.degrees(angle\_rad)  
  
 # Add the angle to the total  
 total\_angle += angle\_deg  
  
 return total\_angle  
  
  
def create\_polygon(pos):  
 # Calculate the centroid of the coordinates  
 centroid = np.mean(pos, axis=0)  
  
 # Sort the coordinates based on their polar angles relative to the centroid  
 sorted\_coordinates = sorted(pos,  
 key=lambda coord: np.arctan2(coord[1] - centroid[1], coord[0] - centroid[0]))  
  
 # Extract x and y coordinates from the sorted list  
 x\_coords, y\_coords = zip(\*sorted\_coordinates)  
  
 # Rotate the polygon 90 degrees to the right  
 rotated\_x\_coords = [centroid[0] - (y - centroid[1]) for x, y in sorted\_coordinates]  
 rotated\_y\_coords = [centroid[1] + (x - centroid[0]) for x, y in sorted\_coordinates]  
  
 # Create a 256x256 canvas  
 fig, ax = plt.subplots(figsize=(5, 5))  
 ax.set\_xlim(0, 256)  
 ax.set\_ylim(0, 256)  
  
 # Plot the rotated coordinates as a polygon  
 plt.plot(rotated\_x\_coords + [rotated\_x\_coords[0]], rotated\_y\_coords + [rotated\_y\_coords[0]], 'r')  
  
 # Display the plot  
 plt.gca().invert\_yaxis() # Invert the y-axis to match image coordinates  
 plt.gca().invert\_xaxis() # Invert the y-axis to match image coordinates  
 plt.axis('off') # Turn off axis labels and ticks  
 # plt.show()  
  
 rotated\_polygon = [(x, y) for x, y in zip(rotated\_x\_coords, rotated\_y\_coords)]  
 return rotated\_polygon  
  
  
def create\_triangle(pos):  
 # Calculate the centroid of the coordinates  
  
 centroid = np.mean(pos, axis=0)  
  
 # Rotate the original coordinates 90 degrees to the right  
 pos = [(centroid[0] + (y - centroid[1]), centroid[1] - (x - centroid[0])) for x, y in pos]  
  
 # Sort the coordinates based on their polar angles relative to the centroid  
 sorted\_coordinates = sorted(pos,  
 key=lambda coord: np.arctan2(coord[1] - centroid[1], coord[0] - centroid[0]))  
  
 # Extract x and y coordinates from the sorted list  
 x\_coords, y\_coords = zip(\*sorted\_coordinates)  
  
 # Add the first coordinate at the end to close the polygon  
 x\_coords = list(x\_coords) + [x\_coords[0]]  
 y\_coords = list(y\_coords) + [y\_coords[0]]  
  
  
 # Calculate the lengths of edges  
 edge\_lengths = [np.sqrt((x2 - x1) \*\* 2 + (y2 - y1) \*\* 2) for (x1, y1), (x2, y2) in  
 zip(sorted\_coordinates, sorted\_coordinates[1:] + [sorted\_coordinates[0]])]  
  
 # Find the indices of the two longest edges  
 indices\_of\_longest\_edges = np.argsort(edge\_lengths)[-2:]  
  
 # Extract the coordinates of the two longest edges  
 longest\_edge1 = [sorted\_coordinates[indices\_of\_longest\_edges[0]],  
 sorted\_coordinates[(indices\_of\_longest\_edges[0] + 1) % len(pos)]]  
 longest\_edge2 = [sorted\_coordinates[indices\_of\_longest\_edges[1]],  
 sorted\_coordinates[(indices\_of\_longest\_edges[1] + 1) % len(pos)]]  
  
 # Create a 256x256 canvas  
 fig, ax = plt.subplots(figsize=(5, 5))  
 ax.set\_xlim(0, 256)  
 ax.set\_ylim(0, 256)  
  
 # Plot the sorted coordinates as a polygon  
 plt.plot(x\_coords, y\_coords, 'r')  
  
 # Plot the two longest edges as a triangle  
 triangle\_x = [longest\_edge1[0][0], longest\_edge1[1][0], longest\_edge2[1][0]]  
 triangle\_y = [longest\_edge1[0][1], longest\_edge1[1][1], longest\_edge2[1][1]]  
 plt.fill(triangle\_x, triangle\_y, 'b', alpha=0.5, label='Triangle')  
  
 # Display the legend  
 plt.legend()  
  
 # Display the plot  
 # plt.gca().invert\_yaxis() # Invert the y-axis to match image coordinates  
 plt.gca().invert\_xaxis() # Invert the y-axis to match image coordinates  
 plt.axis('on') # Turn off axis labels and ticks  
 # plt.show()  
  
  
def calculate\_intersection(edge1, edge2):  
 *"""  
 Calculate the intersection point of two infinitely extended line segments.  
  
 Parameters:  
 edge1 (tuple): Tuple of two points representing the first edge.  
 edge2 (tuple): Tuple of two points representing the second edge.  
  
 Returns:  
 tuple or None: The intersection point as a tuple (x, y), or None if the lines are parallel.  
 """* x1, y1 = edge1[0]  
 x2, y2 = edge1[1]  
 x3, y3 = edge2[0]  
 x4, y4 = edge2[1]  
  
 # Calculate determinants  
 det = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)  
  
 if det == 0:  
 return None # Lines are parallel, no intersection  
  
 # Calculate intersection point  
 px = ((x1 \* y2 - y1 \* x2) \* (x3 - x4) - (x1 - x2) \* (x3 \* y4 - y3 \* x4)) / det  
 py = ((x1 \* y2 - y1 \* x2) \* (y3 - y4) - (y1 - y2) \* (x3 \* y4 - y3 \* x4)) / det  
  
 return px, py  
  
  
def create\_new\_polygon(polygon):  
 *"""  
 Add an intersection point to the two longest edges of a polygon (infinite length) and plot it.  
  
 Parameters:  
 polygon (list of tuples): List of (x, y) coordinates representing the polygon vertices.  
  
 Returns:  
 list of tuples: A new polygon with the intersection point.  
 """* if len(polygon) < 3:  
 return None # A polygon must have at least 3 vertices  
  
 # Calculate edge lengths and store them with their corresponding vertices  
 edges = []  
 for i in range(len(polygon)):  
 x1, y1 = polygon[i]  
 x2, y2 = polygon[(i + 1) % len(polygon)] # Wrap around for the last vertex  
 edge\_length = np.sqrt((x2 - x1) \*\* 2 + (y2 - y1) \*\* 2)  
 edges.append(((x1, y1), (x2, y2), edge\_length))  
  
 # Sort the edges by length in descending order  
 edges.sort(key=lambda edge: -edge[2])  
  
 # Extract the two longest edges  
 longest\_edge1 = edges[0]  
 longest\_edge2 = edges[1]  
  
 # Calculate the intersection point (infinite length)  
 intersection = calculate\_intersection(longest\_edge1, longest\_edge2)  
  
 if intersection is None:  
 return None # Lines are parallel, no intersection point  
  
 # Find the index of the first longest edge in the original polygon  
 index\_longest\_edge1 = polygon.index(longest\_edge1[0])  
  
 # Create a new polygon by adding the intersection point after the first longest edge  
 new\_polygon = polygon[:index\_longest\_edge1 + 1] + [intersection] + polygon[index\_longest\_edge1 + 2:]  
  
 # Plot the new polygon  
 x\_coords, y\_coords = zip(\*new\_polygon)  
  
 # Create a 256x256 canvas  
 fig, ax = plt.subplots(figsize=(5, 5))  
 ax.set\_xlim(0, 256)  
 ax.set\_ylim(0, 256)  
  
 # Plot the new polygon  
 plt.plot(x\_coords + (x\_coords[0],), y\_coords + (y\_coords[0],), 'b', label='New Polygon')  
  
 # Display the legend  
 plt.legend()  
  
 # Display the plot  
 plt.gca().invert\_yaxis() # Invert the y-axis to match image coordinates  
 plt.axis('off') # Turn off axis labels and ticks  
 # plt.show()  
  
 return new\_polygon  
  
  
def calculate\_polygon\_area(polygon):  
 *"""  
 Calculate the area of a polygon using the shoelace formula.  
  
 Parameters:  
 polygon (list of tuples): List of (x, y) coordinates representing the polygon vertices.  
  
 Returns:  
 float: The area of the polygon.  
 """* if len(polygon) < 3:  
 return 0.0 # A polygon with less than 3 vertices has zero area  
  
 x\_coords, y\_coords = zip(\*polygon)  
 x\_coords = list(x\_coords)  
 y\_coords = list(y\_coords)  
  
 x\_coords.append(x\_coords[0])  
 y\_coords.append(y\_coords[0])  
  
 area = 0.0  
 for i in range(len(polygon)):  
 area += x\_coords[i] \* y\_coords[i + 1] - x\_coords[i + 1] \* y\_coords[i]  
  
 area = 0.5 \* abs(area)  
 return area  
  
  
def show\_and\_compare\_polygons(poly1, poly2):  
 *"""  
 Display two polygons on a 256x256 canvas and print the difference in area between them.  
  
 Parameters:  
 poly1 (list of tuples): List of (x, y) coordinates representing the first polygon vertices.  
 poly2 (list of tuples): List of (x, y) coordinates representing the second polygon vertices.  
 """* # Create a 256x256 canvas  
 fig, ax = plt.subplots(figsize=(5, 5))  
 ax.set\_xlim(0, 256)  
 ax.set\_ylim(0, 256)  
  
 # Plot the first polygon  
 x\_coords1, y\_coords1 = zip(\*poly1)  
 x\_coords1 += (x\_coords1[0],)  
 y\_coords1 += (y\_coords1[0],)  
 plt.plot(x\_coords1, y\_coords1, 'r', label='Polygon 1')  
  
 # Plot the second polygon  
 x\_coords2, y\_coords2 = zip(\*poly2)  
 x\_coords2 += (x\_coords2[0],)  
 y\_coords2 += (y\_coords2[0],)  
 plt.plot(x\_coords2, y\_coords2, 'g', label='Polygon 2')  
  
 # Display the legend  
 plt.legend()  
  
 # Calculate and print the difference in area between the two polygons  
 area1 = calculate\_polygon\_area(poly1)  
 area2 = calculate\_polygon\_area(poly2)  
 area\_difference = abs(area1 - area2)  
 print(f"Difference in area: {area\_difference:.2f} square units")  
  
 # Display the plot  
 plt.gca().invert\_yaxis() # Invert the y-axis to match image coordinates  
 plt.gca().invert\_xaxis() # Invert the y-axis to match image coordinates  
 plt.axis('off') # Turn off axis labels and ticks  
 # plt.show()  
  
 return area\_difference  
  
  
def return\_params(image\_path):  
 # Load the image  
 image = io.imread(image\_path)  
 image = color.rgb2gray(image)  
  
 # image = equalize(image)  
 logarithmic\_corrected = corrections(image)  
 result\_image = erosion\_(logarithmic\_corrected)  
 result\_image = dilation\_(result\_image)  
 # result\_image = opening\_(result\_image)  
 result\_image = closing\_(result\_image)  
  
 pos = find\_corner(result\_image) # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] >= 20 and pair[1] >= 20]  
  
 # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] <= 230 and pair[1] <= 230] polygon = create\_polygon(pos)  
  
 smallest\_angle = angle\_calc(polygon)  
 print(f"The smallest angle in the polygon is {smallest\_angle:.2f} degrees.")  
 total\_angle = calculate\_total\_polygon\_angle(polygon)  
 print(f"The total angle in the polygon is {total\_angle:.2f} degrees.")  
 # create\_triangle(polygon)  
 new\_polygon = create\_new\_polygon(polygon)  
 area\_diff = show\_and\_compare\_polygons(polygon, new\_polygon)  
  
 # show\_images(image, result\_image)  
  
 return smallest\_angle, total\_angle, area\_diff  
  
  
def main(image\_path):  
 # Load the image  
 image = io.imread(image\_path)  
 image = color.rgb2gray(image)  
  
 # image = equalize(image)  
 logarithmic\_corrected = corrections(image)  
  
 result\_image = erosion\_(logarithmic\_corrected)  
 result\_image = dilation\_(result\_image)  
 # result\_image = opening\_(result\_image)  
 result\_image = closing\_(result\_image)  
  
 pos = find\_corner(result\_image) # *todo: continue* # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] >= 20 and pair[1] >= 20]  
  
 # Filter pairs based on the threshold  
 pos = [pair for pair in pos if pair[0] <= 230 and pair[1] <= 230] # *todo : work with polygon* polygon = create\_polygon(pos)  
  
 smallest\_angle = angle\_calc(polygon)  
 print(f"The smallest angle in the polygon is {smallest\_angle:.2f} degrees.")  
 total\_angle = calculate\_total\_polygon\_angle(polygon)  
 print(f"The total angle in the polygon is {total\_angle:.2f} degrees.")  
 # create\_triangle(polygon)  
 new\_polygon = create\_new\_polygon(polygon)  
 area\_diff = show\_and\_compare\_polygons(polygon, new\_polygon)  
  
 show\_images(image, result\_image)  
  
 return smallest\_angle, total\_angle, area\_diff

train\_classifier

import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import LogisticRegression  
from sklearn.metrics import f1\_score  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.metrics import confusion\_matrix  
  
  
def create\_hist(df):  
 # Create a figure  
 plt.figure(figsize=(10, 6))  
  
 # Combine all features into a single DataFrame  
 data\_to\_plot = df[['Smalles\_angle', 'Sum\_of\_angles', 'Area\_diff']]  
  
 # Create a KDE plot for all features combined  
 sns.kdeplot(data=data\_to\_plot, shade=True)  
  
 # Set labels and title  
 plt.xlabel('Values')  
 plt.ylabel('Probability Density')  
 plt.title('Combined Probability Density Function')  
  
 plt.tight\_layout()  
 plt.show()  
  
  
# Load your DataFrame  
path = r'C:\Users\IlyaY\Desktop\לימודים\תשפג\ק\עיבוד תמונה\Lapis\all\data.csv'  
df = pd.read\_csv(path) # Replace 'your\_data.csv' with your DataFrame file  
  
  
# Extract columns  
X = df[['Smalles\_angle', 'Sum\_of\_angles', 'Area\_diff']] # Replace with your parameter columns  
y = df['Label'] # Replace with your true label column  
  
# Split your data into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# Train a classifier (e.g., Logistic Regression) with multiple features  
classifier = LogisticRegression()  
classifier.fit(X\_train, y\_train)  
  
# Generate predicted probabilities on the test set with multiple features  
y\_probs = classifier.predict\_proba(X\_test)[:, 1] # Assuming class 1 is the positive class  
  
# Calculate the F1-score for various threshold values  
thresholds = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] # You can adjust these thresholds  
f1\_scores = []  
  
for threshold in thresholds:  
 y\_pred = (y\_probs >= threshold).astype(int)  
 f1 = f1\_score(y\_test, y\_pred)  
 f1\_scores.append(f1)  
  
# Find the threshold that maximizes the F1-score for the multi-feature model  
optimal\_threshold = thresholds[f1\_scores.index(max(f1\_scores))]  
  
# Save the trained multi-feature classifier to a file  
# joblib.dump(classifier, r"C:\Users\IlyaY\Desktop\לימודים\תשפג\ק\עיבוד תמונה\Lapis\all\trained\_classifier.pkl")  
  
# Convert predicted probabilities to binary predictions using the threshold  
y\_pred = (y\_probs >= optimal\_threshold).astype(int)  
  
# Calculate confusion matrix  
conf\_matrix = confusion\_matrix(y\_test, y\_pred)  
  
# Extract TP, TN, FP, FN from the confusion matrix  
TP = conf\_matrix[1, 1]  
TN = conf\_matrix[0, 0]  
FP = conf\_matrix[0, 1]  
FN = conf\_matrix[1, 0]  
  
# Create a DataFrame to display the results as a table  
result\_df = pd.DataFrame({'Metric': ['True Positives (TP)', 'True Negatives (TN)', 'False Positives (FP)', 'False Negatives (FN)'],  
 'Count': [TP, TN, FP, FN]})  
  
# Display the DataFrame as a table in Jupyter Notebook  
print(result\_df)