segmentation_no_supervisada_pytorch

June 15, 2024

1 Pruebas de segmentación con el Catálogo Morfológico de Nebulosas Planetarias del IAC (NO SUPERVISADO)

En este documento vamos a probar las técnicas comentadas por Diego Cantorna en el documento de astrogestem (disponible en la carpeta astrosegstem de este mismo repositorio) y vamos a añadir ciertas técnicas y mejoras. Todas las técnicas testeadas en este Jupyter Notebook son de aprendizaje no supervisado.

1.1 1. Carga del Dataset

Vamos a definir una clase, que tome como base la clase Dataset de Pytorch, para poder cargar todo nuestro conjunto de imágenes de uno o varios canales con su máscara correspondiente.

[ ]:
```
import os

print("Vamos a cambiar el directorio de trabajo")

# Indicamos la ruta del directorio de trabajo
route = os.getcwd()+ "/TFG/test/PNe_segmentation"
os.chdir(route)

current_directory = os.getcwd()
print(" El directorio actual es:", current_directory)

# Listamos el contenido del directorio
files = os.listdir(current_directory)
print(" Contenido del directorio actual:")
for file in files:
    print("\t",file)

# Listamos el contenido del directorio de las máscaras
# masks_directory = route+"TFG\\test\\PNe_segmentation\\masks"
# data_directory = route+"TFG\\test\\PNe_segmentation\\data"
## Ejecución en el CESGA Finisterrae III
masks_directory = route+"/masks"
data_directory = route+"/data"
```

[ ]:
```
import glob
from torch.utils.data import Dataset
import matplotlib.pyplot as plt
import torch
from torchvision import transforms
from torchvision.transforms.functional import InterpolationMode
import random as rd
import numpy as np
from astropy.io import fits

MinMaxNorm = lambda x: (x - np.min(x)) / (np.max(x) - np.min(x))

class NebulaeDataset(Dataset):

    def __init__(self, image_path, mask_path, dataframe, rsize = None,␣
 ↪transform = None):
        super().__init__()

        self.image_path = image_path  # Ruta a las imágenes
        self.mask_path = mask_path  # Ruta a las máscaras

        # Cargar los nombres de las imágenes y máscaras desde el dataframe
        self.data_dict = dataframe.set_index('name').to_dict(orient='index')

        # Filtrar las rutas de archivo según los nombres en el dataframe
        self.img_files = [os.path.join(self.image_path, files['h']) for files␣
 ↪in self.data_dict.values()]
        self.mask_files = [os.path.join(self.mask_path, files['mask']) for␣
 ↪files in self.data_dict.values()]
        self.names = list(self.data_dict.keys())  # Nombres de las imágenes y␣
 ↪máscaras

        self.rsize = rsize  # Size to use in default Resize transform
        self.transform = transform

    # Returns both the image and the mask
    def __getitem__(self, index):
        img_path = self.img_files[index]
        mask_path = self.mask_files[index]

        image = np.flip(fits.getdata(img_path, memmap=False).astype(np.
 ↪float32), axis=0)
        mask = plt.imread(mask_path)

        # Take only the first channel. CHANGE THIS IF WE ARE GOING TO WORK WITH␣
 ↪NUMEROUS CHANNELS
        if len(mask.shape) > 2:
```

```
            mask = mask[:,:,0]
        if len(image.shape) > 2:
            image = image[:,:,0]

        # Apply the defined transformations to both image and mask
        if self.transform is not None:
            seed = np.random.randint(2147483647) # make a seed with numpy␣
↪generator
            rd.seed(seed) # apply this seed to image transforms
            torch.manual_seed(seed)
            if type(self.transform) == tuple:
                image = self.transform[0](image)
            else:
                image = self.transform(image)
            rd.seed(seed) # apply the same seed to mask transforms
            torch.manual_seed(seed)
            if type(self.transform) == tuple:
                mask = self.transform[1](mask)
            else:
                mask = self.transform(mask)
        else:
            if self.rsize is not None:
                t = transforms.Compose([
                    MinMaxNorm,
                    transforms.ToTensor(),
                    transforms.Resize(self.rsize, interpolation=␣
↪InterpolationMode.NEAREST)
                    ])
            else:
                t = transforms.Compose([
                    MinMaxNorm,
                    transforms.ToTensor()
                    ])

            image = t(image)
            mask = t(mask)

        return image, mask.int()

    def __len__(self):
        return len(self.img_files)

    def different_shapes(self):
        shapes = set([tuple(self[i][0].permute(2,1,0).shape) for i in␣
↪range(len(self))])
        return list(shapes)
```

```
    def plot(self, index, plot_image = True, plot_mask = False):
        """
        Muestra una imagen y/o máscara aleatoria del lote.

        Parámetros:
        index (int): Índice del lote.
        plot_image (bool, opcional): Si es True, muestra la imagen. Por defecto␣
↪es True.
        plot_mask (bool, opcional): Si es True, muestra la máscara. Por defecto␣
↪es False.
        """
        image, mask = self[index]
        image = image.permute(1,2,0)
        mask = mask.permute(1,2,0)
        name = self.names[index]

        if plot_image:
            fig, ax = plt.subplots(1, 1, figsize=(5, 5))
            fig.suptitle(f"Canales de la nebulosa {name}", fontweight = 'bold',␣
↪fontsize = 14)
            ax.imshow(image, cmap = "gray")
            ax.set_title(f"Canal H")
            fig.show()
        if plot_mask:
            fig, ax = plt.subplots(1, 1, figsize=(5, 5))
            fig.suptitle(f"Máscara de la nebulosa {name}", fontweight = 'bold',␣
↪fontsize = 14)
            ax.imshow(mask, cmap = "gray")
            fig.show()

    def different_shapes(self):
        shapes = set([tuple(self[i][0].permute(2,1,0).shape) for i in␣
↪range(len(self))])
        return list(shapes)

    def bg_obj_proportions(self):
        proportions = []
        for i in range(len(self)):
            mask = self[i][1].numpy()
            bg = np.sum(mask == 0)
            obj = np.sum(mask == 1)
            proportions.append(obj/(bg+obj))
        return proportions
```

Ahora deberíamos de ser capaces de poder cargar todas las imágenes, como tensores de PyTorch, de nuestro csv como un Dataset

[ ]:
```
import pandas as pd
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df)
```

[ ]:
```
dataset.plot(40, plot_image = True, plot_mask = True)
```

1.2 2. Segmentación de las imágenes

Para la segmentación de las imágenes, vamos a probar diferentes algoritmos/técnicas propuestas por Diego Cantorna en el notebook de 'astrosegstem', pero para un mayor conjunto de datos para poder evaluarlas y verificar sus resultados.

1.2.1 Evaluación de resultados

Para evaluar el resultado de las técnicas de segmentación se pueden emplear distintas métricas.

Algunas de las métricas más utilizadas son la precisión, accuracy y recall, que junto al análisis de la matriz de confusión son las más utilizadas para cualquier problema de procesamiento de imágenes. A estas también se unen: - **Coeficiente de Dice** (Dice Similarity Coefficient): Mide la similitud entre la segmentación predicha y la segmentación de referencia. Valores más cercanos a 1 indican una mejor superposición. Para un problema de segmentación binaria como el nuestro, el F1-Score y el Dice son equivalentes. - **Índice de Jaccard** (Jaccard Index o Intersection over Union, IoU): Calcula la intersección entre la segmentación predicha y la segmentación de referencia dividida por su unión. También mide la superposición. las cuales son métricas básicas y esenciales en los problemas de segmentación.

Vamos a utilizar todas estas métricas para evaluar nuestras técnicas de segmentación.

Vamos a dar mayor prioridad a la hora de evaluar al F1-Score (equivalente al Dice), al IoU y a la precisión, debido a que queremos extraer la silueta aunque no sea de una manera exacta. Métricas como el accuracy en este tipo de problemas no son muy representativas dado a que si el 90 por ciento de la imagen es fondo y nuestra técnica predice como máscara todo 0s (es decir, todo negro) nos va a devolver un 90 por ciento de accuracy pero realmente no nos estaría aportando ninguna información de valor. El recall sería una métrica más idónea si no nos quisiésemos saltar ningún píxel que tenemos que predecir como positivo, aunque diésemos algún falso positivo (métrica muy observada en segmentación de imagen médica por ejemplo).

1.2.2 Técnicas de agrupamiento de datos

Las técnicas de agrupamiento (clustering) tratan de encontrar una partición de un conjunto de datos de forma que los elementos de un mismo grupo sean más similares que los elementos de grupos distintos. Esto permite resumir un conjunto de datos, y puede facilitar algunos procesos de visualización o análisis posteriores con otras técnicas.

[ ]:
```
from sklearn.cluster import KMeans
from skimage import morphology
from skimage import exposure
from scipy import ndimage
import skfuzzy as fuzz
```

```
class ApplyKMeans:
    def __init__(self, concat = False, **kwargs):
        self.concat = concat
        self.kwargs = kwargs

    def __call__(self, im):
        im_orig = im.copy()
        if len(im.shape) == 3 and im.shape[2] > 1:
            im = im[:,:,-1]

        im_array = im.reshape(-1, 1)

        kmeans = KMeans(**self.kwargs).fit(im_array)  # Entrenar el modelo␣
↪K-Means

        # Obtener la imagen segmentada aplicando el algoritmo a cada píxel de␣
↪la imagen
        im_segm_array = kmeans.predict(im_array)

        # Reemplazar los índices de los clústeres por los centroides de los␣
↪clústeres
        im_segm_array = np.array([kmeans.cluster_centers_[i] for i in␣
↪im_segm_array])

        # Cambiar las dimensiones de los datos segmentados para que se␣
↪correspondan con la imagen inicial
        im_segm = im_segm_array.reshape(im.shape[0], im.shape[1], 1)

        if self.concat:
            if len(im_orig.shape) < 3:
                im_orig = np.expand_dims(im_orig, axis=2)

            return np.concatenate((im_orig, im_segm), axis=2)
        else:
            return im_segm

class ApplyFCM:
    def __init__(self, concat = False, **kwargs):
        self.concat = concat
        self.kwargs = kwargs

    def __call__(self, im):
        im_orig = im.copy()
        if len(im.shape) == 3 and im.shape[2] > 1:
            im = im[:,:,-1]

        im_array = im.reshape(1, -1)
```

```
        cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(data=im_array, **self.
 ↪kwargs)  # Aplicar el algoritmo FCM

        # Asociar a cada píxel el cluster para el que tiene una mayor␣
 ↪pertenencia
        clusters_array = np.argmax(u, axis=0)
        maximos = np.max(u, axis=0)

        # Reemplazar los índices de los clústeres por los centroides de los␣
 ↪clústeres
        im_segm_array = np.array([cntr[i] for i in clusters_array])

        # Cambiar las dimensiones de los datos segmentados para que se␣
 ↪correspondan con la imagen inicial
        im_segm = im_segm_array.reshape(im.shape[0], im.shape[1], 1)

        maximos = maximos.reshape(im.shape[0], im.shape[1], 1)
        im_segm = np.concatenate((maximos, im_segm), axis=2)

        if self.concat:
            if len(im_orig.shape) < 3:
                im_orig = np.expand_dims(im_orig, axis=2)

            return np.concatenate((im_orig, im_segm), axis=2)
        else:
            return im_segm

class ApplyMorphology:
    def __init__(self, operation = morphology.opening, concat = False,␣
 ↪**kwargs):
        self.concat = concat
        self.operation = operation
        self.kwargs = kwargs
        if operation == morphology.binary_opening or operation == morphology.
 ↪binary_closing:
            self.mode = "star_background"
        else:
            self.mode = "nebulae"

    def __call__(self, im):
        im_orig = im.copy()
        if len(im.shape) == 3 and im.shape[2] > 1:
            im = im[:,:,-1]

        if self.mode == "nebulae":
```

```
            im_filt = self.operation(im, **self.kwargs)
        else:
            im_preproc = np.copy(im)
            im_filt = ndimage.gaussian_filter(im, sigma=3)
            im_filt[im == 0] = 0

            im_zonas_claras_peq = im > (im_filt + np.std(im))

            im_zonas_claras_peq = self.operation(im_zonas_claras_peq, **self.
 ↪kwargs)

            im_preproc = (im_preproc - np.min(im_preproc))
            im_preproc[im_zonas_claras_peq] = 0

            im_filt = im_preproc
        if self.concat:
            if len(im_orig.shape) < 3:
                im_orig = np.expand_dims(im_orig, axis=2)

            im_filt = np.expand_dims(im_filt, axis=2)
            return np.concatenate((im_orig, im_filt), axis=2)
        else:
            return self.operation(im, **self.kwargs)

class ApplyIntensityTransformation:
    def __init__(self, transformation = exposure.rescale_intensity, concat =␣
 ↪False, **kwargs):
        self.transformation = transformation
        self.kwargs = kwargs
        self.concat = concat
        self.in_range = None
        self.kernel_size = None

        if "in_range" in self.kwargs:
            self.in_range = self.kwargs["in_range"]

        if "kernel_size" in self.kwargs:
            self.kernel_size = self.kwargs["kernel_size"]

    def __call__(self, im):
        im_orig = im.copy()
        if len(im.shape) == 3 and im.shape[2] > 1:
            im = im[:,:,-1]

        if self.in_range is not None:
            self.kwargs["in_range"] = (im.max() * self.in_range[0], im.max() *␣
 ↪self.in_range[1])
```

```
        # self.kwargs["in_range"] = (im.min(), im.max()) # Linea para realizar␣
 ↪un reescalado de la intensidad de la imagen lineal

        if self.kernel_size is not None:
            self.kwargs["kernel_size"] = im.shape[0] // self.kernel_size

        im_trans = self.transformation(im, **self.kwargs)
        if self.concat:
            if len(im_orig.shape) < 3:
                im_orig = np.expand_dims(im_orig, axis=2)

            im_trans = np.expand_dims(im_trans, axis=2)
            return np.concatenate((im_orig, im_trans), axis=2)
        else:
            return self.transformation(im, **self.kwargs)

class ApplyFilter:
    def __init__(self, filter = ndimage.gaussian_filter, concat = False,␣
 ↪**kwargs):
        self.filter = filter
        self.kwargs = kwargs
        self.concat = concat

    def __call__(self, im):
        im_orig = im.copy()
        if len(im.shape) == 3 and im.shape[2] > 1:
            im = im[:,:,-1]

        im_filt = self.filter(im, **self.kwargs)
        if self.concat:
            if len(im_orig.shape) < 3:
                im_orig = np.expand_dims(im_orig, axis=2)

            im_filt = np.expand_dims(im_filt, axis=2)
            return np.concatenate((im_orig, im_filt), axis=2)
        else:
            return self.filter(im, **self.kwargs)

class CustomPad():
    def __init__(self, target_size = (1056, 1536), fill = 0):
        self.target_size = target_size
        self.fill = fill

    def __call__(self, image):
        # Get the size of the input image
        width, height = image.shape[2], image.shape[1]
```

```
        # Compute the size of the padding
        pad_width = self.target_size[1] - width
        pad_height = self.target_size[0] - height

        # Compute the padding
        pad_left = pad_width // 2
        pad_right = pad_width - pad_left
        pad_top = pad_height // 2
        pad_bottom = pad_height - pad_top

        # Apply the padding
        return transforms.functional.pad(image, (pad_left, pad_top, pad_right,␣
 ↪pad_bottom), fill = self.fill)


def plot_all(image, mask, **kwargs):
        image = image.permute(1,2,0)
        mask = mask.permute(1,2,0)

        n_channels = image.shape[2]
        fig, ax = plt.subplots(1, n_channels + 1, figsize=(5 * n_channels, 5))
        # fig.suptitle(f"Canales de la nebulosa y máscara", fontweight =␣
 ↪'bold', fontsize = 14)
        for i in range(n_channels):
            ax[i].imshow(image[:,:,i]*255, **kwargs)
            ax[i].set_title(f"Canal {i}")

        ax[n_channels].imshow(mask, cmap = "gray")
        ax[n_channels].set_title(f"Máscara")
        fig.show()

def filter_cluster(image, min_background_percentage=0.90, mask_probs = None):
    """
    Filtra los clusters de una imagen binarizada para obtener el fondo.

    Parámetros:
    image (torch.Tensor): Imagen binarizada.
    min_background_percentage (float, opcional): Porcentaje mínimo de píxeles␣
 ↪de fondo. Por defecto es 0.90.

    Retorna:
    torch.Tensor: Imagen binarizada con el fondo.

    """

    # Sort unique cluster values in ascending order
```

```
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
↪tensor(1))

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage:
            break

        background = new_background

    if mask_probs is not None:
        background = background * mask_probs

    return background
```

1.2.3 2.1. K-Means

Vamos a comenzar por la técnica más básica (y en la que se basan la mayoría), K-Means.

Para aplicar el algoritmo se selecciona el número de grupos a utilizar y un prototipo (elemento representativo) de cada grupo. A continuación se realiza un proceso iterativo en el que se van asignando datos al grupo más próximo, se recalcula el prototipo de cada grupo, y se repite el proceso hasta que se estabiliza.

En este caso aplicaremos el algoritmo a los píxeles de imágenes de niveles de gris, por lo que el prototipo de cada grupo será el valor de un píxel. El número de grupos podemos establecerlo manualmente, realizando pruebas con distintos valores. Existen algoritmos más complejos que tratan de automatizar el proceso, pero es interesante familiarizarse inicialmente con las versiones más simples de los algoritmos, para centrar el estudio en los aspectos fundamentales.

Las pruebas con el K-Means van a ser realizadas con un modelo K-Means para cada imagen (para ambos canales) normalizando los datos entre 0 y 1 y dejandolos con sus valores reales (Demostrado que funciona exactamente igual, solo se realizará con los datos normalizados para que todas las imágenes se muevan en el mismo rango de valores).

[ ]:
```
from typing import Any
from sklearn.cluster import KMeans

transform_x = transforms.Compose([
                    MinMaxNorm,
```

```
                    ApplyKMeans(concat=False, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

# Prueba normalizando los datos entre 0 y 1
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

dataset.plot(40, plot_image = True, plot_mask = True)
```

[ ]:
```
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

# Prueba normalizando los datos entre 0 y 1
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

rd.seed(42)
random_indexs = rd.sample(range(len(dataset)), 3)
for index in random_indexs:
    plot_all(*dataset[index], cmap = "gray")
```

Vamos a realizar un filtrado de los cluster de la siguiente manera: 1. Tomamos el cluster con el valor de centroide más bajo y lo tomamos como fondo 1. Si el cluster considerado como fondo es demasiado pequeño, nos fijamos en el siguiente cluster con el centroide más bajo 2. Si el siguiente cluster con los valores de centroide más bajo es muy pequeño nos fijamos en el para considerarlo también como fondo. (Esta parte finalmente no le veo mucho sentido implementarlo, debido a que es practicamente lo mismo que variar el umbral general) 2. El resto que no se ha considerado como

fondo se considera como nebulosa consiguiendo una primera aproximación a la segmentación

[ ]:
```
# Celda para observar las proporciones de fondo máximas, mínimas y medias de␣
 ↪las imágenes para decidir el umbral de segmentación
nebulae_proportions = dataset.bg_obj_proportions()
print(f"Mean background proportion: {1-np.mean(nebulae_proportions):.4f}\nMax␣
 ↪background proportion: {1-min(nebulae_proportions):.4f}\nMin background␣
 ↪proportion: {1-max(nebulae_proportions):.4f}")
```

[ ]:
```
# Celda de código donde se realizan algunas pruebas para comprobar los␣
 ↪resultados de la segmentación
# Más adelante se implementa este código en una función que se puede llamar␣
 ↪desde el script principal

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[1]

    min_background_percentage = 0.925
    # min_add_background_percentage = 0.2

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
 ↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
 ↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
 ↪min_background_percentage and not solution:
```

```
            final_background = background.clone()
            solution = True
        #     if add_background.sum() / add_background.numel() >␣
↪min_add_background_percentage:
        #         break
        #     else:
        #         continue

        background = new_background

        cnt += 1

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

Ahora que tenemos una función definida que nos selecciona los primeros clusters convenientes como fondo, vamos a definir una metodología de evaluación y vamos a comprobar que tal funciona nuestra primera aproximación

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[1]

    output = filter_cluster(image_knn, min_background_percentage = 0.93).
↪expand_as(mask)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
↪# Precisión
```

```
    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes␣
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de K-Means

	mean	std
iou	0.301938	0.21419
f1	0.42291	0.256297
precision	0.505177	0.333949
accuracy	0.844327	0.128148
recall	0.587245	0.329879

(NOTA: A medida que aumento el número de clusters, a 9, 11, 13 y 15, obtengo mejores resultados, puede tener algún inconveniente?)

Vamos a aplicar ciertas mejoras como por ejemplo, aplicar operadores morfológicos para tratar de eliminar las estrellas que acaparan demasiada atención de la técnica de agrupamiento de datos, aplicar algún filtro sencillo con el que consigamos una mejor diferenciación entre los clústeres e intentar aplicar alguna técnica de umbralización con la que se escoja de mejor manera el mejor conjunto de clusters para segmentar la nebulosa.

2.1.1. AREA OPENING

[ ]:
```
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df)

imagen = dataset[40][0]
plt.imshow(imagen.permute(1,2,0).numpy(), cmap='gray')
```

```
imagen_opening = morphology.area_opening(imagen.permute(1,2,0)[:,:,0].numpy(),␣
 ↪area_threshold=500)
# imagen_opening = morphology.remove_small_objects(imagen.permute(1,2,0)[:,:,0].
 ↪numpy(), min_size=500)
plt.imshow(imagen_opening, cmap='gray')
```

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])


df = pd.read_csv("data_files_1c.csv")

# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False
```

```
    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.area_opening, concat␣
↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
```

```
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])


df = pd.read_csv("data_files_1c.csv")

# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.93).
 ↪expand_as(mask)

    ## Descomentar para hacer un preprocesado sencillo a las imágenes
    # output = morphology.binary_closing(output.permute(1,2,0).numpy()[:,:,0],␣
 ↪footprint=morphology.disk(5))
    # output = morphology.remove_small_objects(output, min_size=500)
    # output = torch.tensor(output).unsqueeze(0)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes␣
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
```

```
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break


df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con operadores morfológicos (opening en área)

	mean	std
iou	0.474498	0.251984
f1	0.600424	0.260653
precision	0.661401	0.323928
accuracy	0.865881	0.162479
recall	0.758522	0.246124

Como se puede observar, se consiguen mejores resultados aplicando operadores morfológicos que sin ellos. Parece ser que visualmente al procesar la imagen de la última manera que hemos hecho se concentran los valores de los píxeles en ciertos valores, por lo que vamos a imprimir el histograma de algunas imágenes para ver si esto es cierto y vamos a comprobar que no influya demasiado a la hora de realizar el KMeans.

[ ]:
```
# PARA EJECUTAR ESTA CELDA CORRECTAMENTE HACE FALTA HABER EJECUTADO EL␣
 ↪EXPERIMENTO ANTERIOR
for i in range(0, 5):
    im_op_morf = dataset[i][0].permute(1,2,0).numpy()[:,:,1]
    fig, axis = plt.subplots(1,2, figsize = (8,8))
    axis[0].imshow(im_op_morf, cmap='gray')
    axis[0].set_title("Imagen (op. morf.)")
    axis[1].hist(im_op_morf)
    axis[1].set_title("Histograma de la imagen (op. morf.)")
    fig.suptitle(f"Imagen {i}", fontsize=16, fontweight = 'bold')
    fig.show()
```

[ ]:
```
from skimage import exposure
for i in range(0, 5):
    im_op_morf = dataset[i][0].permute(1,2,0).numpy()[:,:,1]
    # Se podría hacer un reescalado de la intensidad de las imagenes del mínimo␣
 ↪al máximo (lineal, simplemente mover el histograma)
    # pero haciendo de esta forma conseguimos resaltar las partes de nebulosa␣
 ↪respecto de las de fondo
```

```
    image_reescaled = exposure.rescale_intensity(im_op_morf, in_range =␣
↪(im_op_morf.max()/5, im_op_morf.max()), out_range = (0, 1))

    # Probamos con un ajuste logarítmico de la intensidad
    # image_reescaled = exposure.adjust_log(im_op_morf, gain=1.2, inv=True)

    # También probamos con una ecualización del histograma para que las␣
↪intensidades estén más repartidas
    # image_reescaled = exposure.equalize_hist(im_op_morf)

    # Y por último, probamos con una ecualización adaptativa del histograma
    # image_reescaled = exposure.equalize_adapthist(im_op_morf, kernel_size =␣
↪im_op_morf.shape[0]//10)

    fig, axis = plt.subplots(1,2, figsize = (8,8))
    axis[0].imshow(image_reescaled, cmap='gray')
    axis[0].set_title("Imagen (op. morf.)")
    axis[1].hist(image_reescaled)
    axis[1].set_title("Histograma de la imagen (op. morf.)")
    fig.suptitle(f"Imagen {i}", fontsize=16, fontweight = 'bold')
    fig.show()
```

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.area_opening, concat␣
↪= True, area_threshold = 200, connectivity = 1),
                    # ApplyIntensityTransformation(transformation = exposure.
↪rescale_intensity, concat = True, in_range = (1/5, 1), out_range = (0, 1)),
                    # ApplyIntensityTransformation(transformation = exposure.
↪equalize_adapthist, concat = True, kernel_size = 5),
                    ApplyIntensityTransformation(transformation = exposure.
↪adjust_log, concat = True, gain = 1.5, inv = True),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
```

```
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
 ↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
 ↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
 ↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
```

```
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪rescale_intensity, concat = True, in_range = (1/5, 1), out_range = (0, 1)),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, kernel_size = 5),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪adjust_log, concat = True, gain = 1.5, inv = True),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.93).
 ↪expand_as(mask)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Índice de Jaccard
```

```
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con operadores morfológicos (opening en área) y adaptación del histograma

	mean	std
iou	0.462187	0.251906
f1	0.588433	0.261978
precision	0.653917	0.331209
accuracy	0.85844	0.166378
recall	0.753286	0.24564

Como se puede observar, los mejores resultados obtenidos son muy similares (iguales si se hace una adaptación del histograma lineal) a los resultados obtenidos sin la adaptación del histograma (después de hacer el operador morfológico), vamos a probar a continuación a hacer una adaptación del histograma a la imagen original y después hacer los operadores morfológicos y el KMeans.

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
```

```
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 7),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
```

```
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    # final_background = morphology.binary_closing(final_background,␣
↪footprint=morphology.disk(5))
    # final_background = morphology.remove_small_objects(final_background,␣
↪min_size=500)

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyIntensityTransformation(transformation = exposure.
↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 7),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
↪n_init=10, random_state=42),
                    transforms.ToTensor(),
```

```
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.93).
 ↪expand_as(mask)

    ## Descomentar para hacer un preprocesado sencillo a las imágenes
    # output = morphology.binary_closing(output.permute(1,2,0).numpy()[:,:,0],␣
 ↪footprint=morphology.disk(5))
    # output = morphology.remove_small_objects(output, min_size=500)
    # output = torch.tensor(output).unsqueeze(0)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)
```

```
    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes␣
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con adaptación del histograma y operadores morfológicos (opening en área)

	mean	std
iou	0.482382	0.243033
f1	0.610257	0.255888
precision	0.646103	0.303263
accuracy	0.911641	0.0766837
recall	0.747889	0.289217

Como podemos observar, los resultados practicamente iguales que los anteriores aunque, al ver las imágenes me hace sospechar que un filtro como por ejemplo Gaussiano después de la adaptación del histograma podría hacer mejorar los resultados considerablemente. Otra mejora que veo posible es la de incluir el operador morfológico de closing en el resultado final como postprocesado, aunque de este apartado podemos hablar más adelante (dejamos los resultados en la siguiente tabla haciendo una pequeña prueba con esta mejora)

	mean	std
iou	0.514091	0.257969
f1	0.635843	0.262296
precision	0.654544	0.309503
accuracy	0.903982	0.104944
recall	0.814793	0.249345

Como última prueba de este apartado vamos a probar a eliminar por completo gracias a los operadores morfológicos el fondo de estrellas de las imágenes, en vez de intentar reducir su visibilidad como estabamos haciendo hasta ahora (gracias a la función area_opening).

2.1.2. BINARY OPENING

[ ]:
```
import skimage.morphology as morphology
from scipy import ndimage
```

```
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df)

fig, ax = plt.subplots(1,2, figsize = (8,8))
imagen = dataset[1][0].permute(1,2,0).numpy()[:,:,0]
imagen_preproc = np.copy(imagen)

ax[0].imshow(imagen, cmap='gray')

imagen_filt = ndimage.gaussian_filter(imagen, sigma=3)
imagen_filt[imagen == 0] = 0

imagen_zonas_claras_peq = imagen > (imagen_filt + np.std(imagen))

imagen_zonas_claras_peq = morphology.binary_opening(imagen_zonas_claras_peq,␣
 ↪morphology.disk(2))

imagen_preproc = (imagen_preproc - np.min(imagen_preproc))
imagen_preproc[imagen_zonas_claras_peq] = 0

# imagen_opening = morphology.remove_small_objects(imagen.permute(1,2,0)[:,:,0].
 ↪numpy(), min_size=500)
ax[1].imshow(imagen_preproc, cmap='gray')
```

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))
```

```
for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
 ↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
 ↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
 ↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

```
[ ]: import segmentation_models_pytorch as smp
     import pandas as pd

     results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

     transform_x = transforms.Compose([
                         MinMaxNorm,
                         ApplyMorphology(operation = morphology.binary_opening,␣
      ↪concat = True, footprint = morphology.disk(2)),
                         ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
      ↪n_init=10, random_state=42),
                         transforms.ToTensor(),
                         # CustomPad(target_size = (980, 980), fill = -1)
                         ])

     transform_y = transforms.Compose([
                         transforms.ToTensor(),
                         # CustomPad(target_size = (980, 980), fill = 0)
                         ])

     df = pd.read_csv("data_files_1c.csv")
     # Prueba normalizando los datos entre 0 y 1
     dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
      ↪(transform_x, transform_y))

     for i in range(len(dataset)):
         image, mask = dataset[i]
         image_knn = image[-1]

         output = filter_cluster(image_knn, min_background_percentage = 0.93).
      ↪expand_as(mask)

         ## Descomentar para hacer un preprocesado sencillo a las imágenes
         # output = morphology.binary_closing(output.permute(1,2,0).numpy()[:,:,0],␣
      ↪footprint=morphology.disk(5))
         # output = morphology.remove_small_objects(output, min_size=500)
         # output = torch.tensor(output).unsqueeze(0)

         tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

         iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
      ↪# Índice de Jaccard
         f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
      ↪# F1-Score
         accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
      ↪# Accuracy
```

```
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break


df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con operadores morfólogicos para eliminar el fondo de estrellas (opening binario)

	mean	std
iou	0.340304	0.202407
f1	0.474101	0.230723
precision	0.548508	0.328343
accuracy	0.860052	0.112781
recall	0.629009	0.295397

Como podemos comprobar, funciona algo mejor que solo utilizar KMeans aunque bastante parecido, como se puede observar en las imágenes observadas. Vamos a probar a unificar las dos técnicas de operadores morfológicos que hemos aplicado, primero eliminamos el fondo de estrellas y después intentamos visualizar lo menos posible los restos que queden de el (primero binary_opening y después area_opening)

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
```

```
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
 ↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()
```

```
        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
↪concat = True, footprint = morphology.disk(2)),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
↪(transform_x, transform_y))
```

```
for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.94).
 ↪expand_as(mask)

    ## Descomentar para hacer un preprocesado sencillo a las imágenes
    # output = morphology.binary_closing(output.permute(1,2,0).numpy()[:,:,0],␣
 ↪footprint=morphology.disk(5))
    # output = morphology.remove_small_objects(output, min_size=500)
    # output = torch.tensor(output).unsqueeze(0)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes␣
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break


df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con operadores morfológicos (ambas técnicas)

	mean	std
iou	0.472936	0.234044
f1	0.605284	0.24067
precision	0.67945	0.311074
accuracy	0.878765	0.142001
recall	0.726142	0.241789

Como se puede ver los resultados son muy parecidos a solo realizar la técnica de operadores morfológicos que NO elimina el fondo de estrellas por completo, aunque hemos tenido que subir un poco el umbral manual (porcentaje de píxeles de fondo) para obtener ese resultado, por lo que vamos a comprobar los resultados que da con una adaptación del histograma entre ambas técnicas de operadores morfológicos.

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 7),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]
```

```
    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    # final_background = morphology.binary_closing(final_background,␣
↪footprint=morphology.disk(5))
    # final_background = morphology.remove_small_objects(final_background,␣
↪min_size=500)

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()
```

```
    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 7),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.94).
 ↪expand_as(mask)

    ## Descomentar para hacer un preprocesado sencillo a las imágenes
    # output = morphology.binary_closing(output.permute(1,2,0).numpy()[:,:,0],␣
 ↪footprint=morphology.disk(5))
    # output = morphology.remove_small_objects(output, min_size=500)
    # output = torch.tensor(output).unsqueeze(0)
```

```
    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con Opening binario, adaptación del histograma y opening en área

	mean	std
iou	0.496861	0.230796
f1	0.62982	0.227609
precision	0.661146	0.282945
accuracy	0.916543	0.0739382
recall	0.769579	0.246387

Vamos a comprobar finalmente que tal funcionaría con el postprocesado sencillo que probamos anteriormente

Resultados de KMeans con Opening binario, adaptación del histograma y opening en

área + postprocesado

	mean	std
iou	0.534588	0.250919
f1	0.658135	0.243586
precision	0.656112	0.29426
accuracy	0.917111	0.0781187
recall	0.841828	0.223383

2.1.3. FILTRO GAUSSIANO

[ ]:
```
from skimage import exposure
import scipy.ndimage as ndimage

df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df)

for i in range(0, 5):
    image = dataset[i][0].permute(1,2,0).numpy()[:,:,0]

    # image_reescaled = exposure.equalize_hist(image, nbins = 640)

    # im_op_morf = morphology.area_opening(image_reescaled, area_threshold=200)

    im_op_morf = morphology.area_opening(image, area_threshold=200)

    imagen_filter = ndimage.gaussian_filter(im_op_morf, sigma = 4)


    # Se podría hacer un reescalado de la intensidad de las imagenes del mínimo␣
↪al máximo (lineal, simplemente mover el histograma)
    # pero haciendo de esta forma conseguimos resaltar las partes de nebulosa␣
↪respecto de las de fondo
    # image_reescaled = exposure.rescale_intensity(im_op_morf, in_range =␣
↪(im_op_morf.max()/5, im_op_morf.max()), out_range = (0, 1))

    # Probamos con un ajuste logarítmico de la intensidad
    # image_reescaled = exposure.adjust_log(im_op_morf, gain=1.2, inv=True)

    # También probamos con una ecualización del histograma para que las␣
↪intensidades estén más repartidas
    # image_reescaled = exposure.equalize_hist(im_op_morf, nbins = 640)

    # Y por último, probamos con una ecualización adaptativa del histograma
    # image_reescaled = exposure.equalize_adapthist(im_op_morf, kernel_size =␣
↪im_op_morf.shape[0]//10)
```

```
    fig, axis = plt.subplots(1,2, figsize = (8,8))
    axis[0].imshow(imagen_filter, cmap='gray')
    axis[0].set_title("Imagen (op. morf.)")
    axis[1].hist(imagen_filter)
    axis[1].set_title("Histograma de la imagen (op. morf.)")
    fig.suptitle(f"Imagen {i}", fontsize=16, fontweight = 'bold')
    fig.show()
```

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    # ApplyMorphology(operation = morphology.area_opening,␣
 ↪concat = True, area_threshold = 200, connectivity = 1),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 5),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyFilter(filter = ndimage.gaussian_filter, concat =␣
 ↪True, sigma = 5),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93
```

```
    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1

    # final_background = morphology.binary_closing(final_background,␣
↪footprint=morphology.disk(5))
    # final_background = morphology.remove_small_objects(final_background,␣
↪min_size=500)

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

```
[ ]: import segmentation_models_pytorch as smp
     import pandas as pd

     results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

     transform_x = transforms.Compose([
                         MinMaxNorm,
                         ApplyMorphology(operation = morphology.binary_opening,␣
      ↪concat = True, footprint = morphology.disk(2)),
                         # ApplyMorphology(operation = morphology.area_opening,␣
      ↪concat = True, area_threshold = 200, connectivity = 1),
                         ApplyIntensityTransformation(transformation = exposure.
      ↪equalize_hist, concat = True, nbins = 640),
                         # ApplyIntensityTransformation(transformation = exposure.
      ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 5),
                         ApplyMorphology(operation = morphology.area_opening, concat␣
      ↪= True, area_threshold = 200, connectivity = 1),
                         ApplyFilter(filter = ndimage.gaussian_filter, concat =␣
      ↪True, sigma = 5),
                         ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
      ↪n_init=10, random_state=42),
                         transforms.ToTensor(),
                         # CustomPad(target_size = (980, 980), fill = -1)
                         ])

     transform_y = transforms.Compose([
                         transforms.ToTensor(),
                         # CustomPad(target_size = (980, 980), fill = 0)
                         ])

     df = pd.read_csv("data_files_1c.csv")
     # Prueba normalizando los datos entre 0 y 1
     dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
      ↪(transform_x, transform_y))

     for i in range(len(dataset)):
         image, mask = dataset[i]
         image_knn = image[-1]

         output = filter_cluster(image_knn, min_background_percentage = 0.94).
      ↪expand_as(mask)

         ## Descomentar para hacer un preprocesado sencillo a las imágenes
         # output = output.permute(1,2,0).numpy()[:,:,0]
         # output = morphology.binary_closing(output, footprint=morphology.disk(5))
         # output = morphology.remove_small_objects(output, min_size=500)
```

```
    # output = torch.tensor(output).unsqueeze(0)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)

    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de KMeans con opening en área y Filtro Gaussiano (sigma 5)

	mean	std
iou	0.533148	0.247184
f1	0.65724	0.245682
precision	0.732878	0.29252
accuracy	0.901664	0.115164
recall	0.758274	0.244278

Resultados de KMeans con adaptación del histograma, opening en área y filtro gaussiano

	mean	std
iou	0.51504	0.252026
f1	0.638368	0.258574
precision	0.694062	0.30799
accuracy	0.918851	0.0819133
recall	0.751193	0.286227

Resultados de KMeans con opening binario, opening en área y Filtro Gaussiano (sigma 5)

	mean	std
iou	0.50307	0.227348
f1	0.636952	0.22226
precision	0.701959	0.301808
accuracy	0.903936	0.106539
recall	0.763739	0.249194

Resultados de KMeans con opening binario, adaptación del histograma, opening en área y Filtro Gaussiano (sigma 5)

	mean	std
iou	0.537252	0.246508
f1	0.663492	0.228315
precision	0.719191	0.298549
accuracy	0.925891	0.073396
recall	0.776446	0.243754

Resultados de KMeans con opening binario, adaptación del histograma, opening en área y Filtro Gaussiano (sigma 5) + postprocesado

	mean	std
iou	0.544275	0.244476
f1	0.670401	0.224853
precision	0.728293	0.294329
accuracy	0.927701	0.0734514
recall	0.777289	0.24409

2.1.4. UMBRALIZACIÓN (para evitar tener que elegir un umbral en el porcentaje de píxeles de fondo) (IGNORAR DE MOMENTO, NECESITA REFLEXIÓN Y DESARROLLO - 14/06/24)

Necesitamos algún método, como por ejemplo un test estadístico, que nos ayude a que la división de clusters en fondo y nebulosa sea más personalizada para cada imagen de lo que es un umbral de porcentaje de píxeles de fondo.

Algunas ideas que se me han ocurrido han sido: - Aplicar algún método, como Otsu, donde se considera cada cluster como una clase y se busca la división de los datos (a través de un umbral) que minimize la varianza intra-clase.

- Intentar modelar los clúster como una Mixtura de Gaussianas (GMM, por sus siglas en inglés Gaussian Mixture Model) de dos Gaussianas (o más, pero por simplicidad y para empezar solo con 2)

(NOTA: Aunque utilicemos otros métodos para separar los clústers en fondo y nebulosa, podemos seguir utilizando el umbral del porcentaje manual para descartar los primeros clústeres que van a ser la mayoría de veces solo fondo)

[ ]:
```
df = pd.read_csv("data_files_1c.csv")
dataset = NebulaeDataset(data_directory, masks_directory, df)

for i in range(0, 5):
    imagen = dataset[i][0].permute(1,2,0).numpy()[:,:,0]
    imagen = exposure.equalize_hist(imagen, nbins = 1024)
    fig, axis = plt.subplots(1,2, figsize = (8,8))
    axis[0].imshow(imagen, cmap='gray')
    axis[0].set_title("Imagen (op. morf.)")
    axis[1].hist(imagen)
    axis[1].set_title("Histograma de la imagen (op. morf.)")
    fig.suptitle(f"Imagen {i}", fontsize=16, fontweight = 'bold')
    fig.show()
```

Método de Otsu (simple) Vamos a comenzar aplicando al KMeans una umbralización de Otsu, tanto de manera general como local, para comprobar si consigue separar los clusters del KMeans en fondo y nebulosa

[ ]:
```
transform_x = transforms.Compose([
                    MinMaxNorm,
                    # ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    # ApplyMorphology(operation = morphology.area_opening,␣
 ↪concat = True, area_threshold = 200, connectivity = 1),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 5),
                    # ApplyMorphology(operation = morphology.area_opening,␣
 ↪concat = True, area_threshold = 200, connectivity = 1),
                    # ApplyFilter(filter = ndimage.gaussian_filter, concat =␣
 ↪True, sigma = 5),
                    ApplyKMeans(concat=True, n_clusters=7, max_iter=10,␣
 ↪n_init=10, random_state=42),
                    transforms.ToTensor(),
```

```
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

rd.seed(42)
random_indexs = rd.sample(range(len(dataset)), 3)
for index in random_indexs:
    plot_all(*dataset[index], cmap = "gray")
```

[ ]:
```
from skimage.filters import threshold_otsu
for i in range(0,5):
    image_original, mask = dataset[i]
    image_knn = image_original[-1]

    image = filter_cluster(image_knn, min_background_percentage = 0.8).
 ↪expand_as(mask).permute(1,2,0).numpy()[:,:,0]

    image = image * image_knn.numpy()
    # Calculamos el histograma de image y lo recortamos desde el minimo␣
 ↪distinto de 0 al maximo
    image_thr = image[image != 0]
    # image_thr = threshold_local(image_thr)[0]

    image_thr = threshold_otsu(image_thr)

    fig, axis = plt.subplots(1,2, figsize = (12,8))

    axis[0].imshow(image, cmap='gray')
    axis[0].set_title("Imagen en escala de grises")
    axis[1].hist(image)
    axis[1].set_title("Histograma de la imagen")

    axis[1].axvline(image_thr, color='r')

    fig.suptitle(f"Imagen {i}", fontsize=16, fontweight = 'bold')
    fig.show()

    plt.figure()
```

```
    plt.imshow(image > image_thr, cmap='gray')
```

[ ]:
```
for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93
    # min_add_background_percentage = 0.2

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True
        #     if add_background.sum() / add_background.numel() >␣
↪min_add_background_percentage:
        #         break
        #     else:
        #         continue

        background = new_background
```

```
        cnt += 1

    # final_background = morphology.binary_closing(final_background,␣
↪footprint=morphology.disk(5))
    # final_background = morphology.remove_small_objects(final_background,␣
↪min_size=500)

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_knn = image[-1]

    output = filter_cluster(image_knn, min_background_percentage = 0.94).
↪expand_as(mask)
    output = output.permute(1,2,0).numpy()[:,:,0]
    output = morphology.binary_closing(output, footprint=morphology.disk(5))
    output = morphology.remove_small_objects(output, min_size=500)
    output = torch.tensor(output).unsqueeze(0)
    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
```

```
    results["recall"].append(recall)

    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

1.2.4 Fuzzy C-Means (FCM)

Vamos a continuar probando una de las variantes del algoritmo de KMeans. Este algoritmo se diferencia del anterior en que, en vez de aportar un cluster al que pertenece cada cluster, aporta un nivel de pertenencia entre 0 y 1.

[ ]:
```
# CELDA PARA MOSTRAR LA SEGMENTACIÓN PASO A PASO Y LAS TRANSFORMACIONES␣
 ↪APLICADAS (Ejecutar si se quiere observar el proceso paso a paso)
threshold = None
# threshold = 0.25

transform_x = transforms.Compose([
                    MinMaxNorm,
                    ApplyMorphology(operation = morphology.binary_opening,␣
 ↪concat = True, footprint = morphology.disk(2)),
                    # ApplyMorphology(operation = morphology.area_opening,␣
 ↪concat = True, area_threshold = 100, connectivity = 1),
                    ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
 ↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 4),
                    ApplyMorphology(operation = morphology.area_opening, concat␣
 ↪= True, area_threshold = 200, connectivity = 1),
                    ApplyFilter(filter = ndimage.gaussian_filter, concat =␣
 ↪True, sigma = 9),
                    ApplyFCM(concat=True, c=7, m=2, error = 0.005, maxiter=15,␣
 ↪seed=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])

transform_y = transforms.Compose([
                    transforms.ToTensor(),
```

```
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(0,5):
    image_original, mask = dataset[i]
    image = image_original[-1]

    min_background_percentage = 0.93

    # Sort unique cluster values in ascending order
    unique_values = image.unique(sorted=True)

    background = torch.where(image == unique_values[0], torch.tensor(0), torch.
 ↪tensor(1))

    fig, ax = plt.subplots(2, (len(unique_values)-1)//2 +␣
 ↪(len(unique_values)-1)%2, figsize=(5 * (len(unique_values)-1)//2, 5 * 2))
    cnt = 0
    solution = False

    # Mientras que el porcentaje de píxeles de fondo sea menor que el␣
 ↪porcentaje mínimo, seguimos añadiendo clusters al fondo
    for cluster_value in unique_values[1:]:
        j, i = divmod(cnt, 2)
        ax[i, j].set_title(f"Background percentage: {(1 - background.sum() /␣
 ↪background.numel())*100:.2f}%", fontsize = 9)
        ax[i, j].imshow(background, cmap = "gray")
        fig.show()

        add_background = torch.where(image == cluster_value, torch.tensor(0),␣
 ↪torch.tensor(1))
        new_background = background * add_background

        if (1 - new_background.sum() / new_background.numel()) >␣
 ↪min_background_percentage and not solution:
            final_background = background.clone()
            solution = True

        background = new_background

        cnt += 1
```

```
    if threshold is not None:
        maxs = image_original[-2]
        probs_mask = maxs > threshold
        final_background = final_background * probs_mask
    # final_background = morphology.binary_closing(final_background,␣
↪footprint=morphology.disk(2))
    # final_background = morphology.remove_small_objects(final_background,␣
↪min_size=500)

    plt.figure()
    plt.imshow(final_background, cmap = "gray")
    plt.title(f"Segmentation")
    plt.show()

    plot_all(image_original, mask, cmap = "gray")
```

[ ]:
```
import segmentation_models_pytorch as smp
import pandas as pd

results = {"iou": [], "f1": [], "precision": [], "accuracy": [], "recall": []}

threshold = None
# threshold = 0.25

mask_probs = None

transform_x = transforms.Compose([
                    MinMaxNorm,
                    # ApplyMorphology(operation = morphology.binary_opening,␣
↪concat = True, footprint = morphology.disk(2)),
                    # ApplyMorphology(operation = morphology.area_opening,␣
↪concat = True, area_threshold = 100, connectivity = 1),
                    # ApplyIntensityTransformation(transformation = exposure.
↪equalize_hist, concat = True, nbins = 640),
                    # ApplyIntensityTransformation(transformation = exposure.
↪equalize_adapthist, concat = True, nbins = 640, kernel_size = 4),
                    # ApplyMorphology(operation = morphology.area_opening,␣
↪concat = True, area_threshold = 200, connectivity = 1),
                    # ApplyFilter(filter = ndimage.gaussian_filter, concat =␣
↪True, sigma = 9),
                    ApplyFCM(concat=True, c=7, m=2, error = 0.005, maxiter=15,␣
↪seed=42),
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = -1)
                    ])
```

```
transform_y = transforms.Compose([
                    transforms.ToTensor(),
                    # CustomPad(target_size = (980, 980), fill = 0)
                    ])

df = pd.read_csv("data_files_1c.csv")
# Prueba normalizando los datos entre 0 y 1
dataset = NebulaeDataset(data_directory, masks_directory, df, transform =␣
 ↪(transform_x, transform_y))

for i in range(len(dataset)):
    image, mask = dataset[i]
    image_clusters = image[-1]

    if threshold is not None:
        mask_probs = image[-2]
        mask_probs = mask_probs > threshold

    output = filter_cluster(image_clusters, min_background_percentage = 0.93,␣
 ↪mask_probs = mask_probs).expand_as(mask)

    ## Descomentar para hacer un preprocesado sencillo a las imágenes
    # output = output.permute(1,2,0).numpy()[:,:,0]
    # output = morphology.binary_closing(output, footprint=morphology.disk(2))
    # output = morphology.remove_small_objects(output, min_size=500)
    # output = torch.tensor(output).unsqueeze(0)

    tp, fp, fn, tn = smp.metrics.get_stats(output, mask, mode='binary')

    iou_score = smp.metrics.iou_score(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Índice de Jaccard
    f1_score = smp.metrics.f1_score(tp, fp, fn, tn, reduction="micro")         ␣
 ↪# F1-Score
    accuracy = smp.metrics.accuracy(tp, fp, fn, tn, reduction="macro")         ␣
 ↪# Accuracy
    recall = smp.metrics.recall(tp, fp, fn, tn, reduction="micro-imagewise")   ␣
 ↪# Sensibilidad
    precision = smp.metrics.precision(tp, fp, fn, tn, reduction="micro")       ␣
 ↪# Precisión

    results["iou"].append(iou_score)
    results["f1"].append(f1_score)
    results["precision"].append(precision)
    results["accuracy"].append(accuracy)
    results["recall"].append(recall)
```

```
    ## Si descomentamos las siguientes líneas, se mostrarán varias imágenes␣
 ↪segmentadas al igual que en la anterior celda de código
    # plot_all(image, mask, cmap = "gray")
    # plt.figure()
    # plt.imshow(output[0], cmap = "gray")
    # plt.title(f"Segmentation")
    # plt.show()
    # if i == 5:
    #     break

df = pd.DataFrame(results)
print(df.astype(float).describe().loc[['mean', 'std']].transpose().
 ↪to_markdown())
```

Resultados de FCM (simple) c=7

	mean	std
iou	0.375927	0.219531
f1	0.512536	0.2191
precision	0.560385	0.282301
accuracy	0.883111	0.0877457
recall	0.670768	0.29144

Resultados de FCM (threshold = 0.25) c=7

	mean	std
iou	0.372125	0.226079
f1	0.505774	0.229752
precision	0.566774	0.294991
accuracy	0.884306	0.0873961
recall	0.625326	0.28131

Resultados de FCM (simple) c=9

	mean	std
iou	0.352356	0.213078
f1	0.487316	0.22107
precision	0.530062	0.288398
accuracy	0.86052	0.110891
recall	0.68024	0.297449

Resultados de FCM (threshold = 0.25) c=9

	mean	std
iou	0.329139	0.2173
f1	0.458464	0.232549
precision	0.530828	0.313011
accuracy	0.860277	0.110107
recall	0.5836	0.284304

Resultados de FCM (simple) c=5

	mean	std
iou	0.346018	0.206797
f1	0.480247	0.227996
precision	0.544016	0.307694
accuracy	0.861736	0.107387
recall	0.647818	0.299709

Resultados de FCM (threshold = 0.25) c=5

	mean	std
iou	0.347282	0.208122
f1	0.481238	0.229512
precision	0.547664	0.309927
accuracy	0.862562	0.107312
recall	0.640019	0.294752

Se puede considerar esta solución como un borrador de fondo de estrellas, ya que al aplicar un umbral de pertenencia en las imágenes siempre descarta las estrellas del fondo, vamos a comparar que tal funciona respecto nuestro borrador de fondo de estrellas con operadores morfológicos.

Resultados de FCM (simple) con operador binario

	mean	std
iou	0.363835	0.209597
f1	0.500761	0.21994
precision	0.534574	0.295538
accuracy	0.871749	0.0907226
recall	0.689259	0.262586

Resultados de FCM (threshold = 0.25) con operador binario

	mean	std
iou	0.351067	0.212059

	mean	std
f1	0.484921	0.228886
precision	0.534494	0.308186
accuracy	0.871943	0.0904017
recall	0.618003	0.256574

Resultados de FCM (simple) con operador en área

	mean	std
iou	0.455727	0.217641
f1	0.595032	0.214648
precision	0.644525	0.302007
accuracy	0.901368	0.082002
recall	0.749689	0.248432

Resultados de FCM (threshold = 0.25) con operador en área

	mean	std
iou	0.441428	0.231196
f1	0.575318	0.239802
precision	0.638409	0.319608
accuracy	0.900503	0.0823982
recall	0.675645	0.272046

Resultados de FCM (simple) con ambos operadores morfológicos

	mean	std
iou	0.434979	0.216478
f1	0.57457	0.216249
precision	0.630748	0.315386
accuracy	0.899437	0.0741227
recall	0.739725	0.250791

Resultados de FCM (threshold = 0.25) con ambos operadores morfológicos

	mean	std
iou	0.416596	0.224087
f1	0.551497	0.239701
precision	0.620918	0.331917
accuracy	0.898226	0.0734305
recall	0.654533	0.285514

Resultados de FCM (simple) con adaptación del histograma

	mean	std
iou	0.344041	0.194668
f1	0.480948	0.220531
precision	0.442844	0.266004
accuracy	0.856461	0.0772394
recall	0.762065	0.266871

Resultados de FCM (threshold = 0.25) con adaptación del histograma

	mean	std
iou	0.441428	0.231196
f1	0.575318	0.239802
precision	0.638409	0.319608
accuracy	0.900503	0.0823982
recall	0.675645	0.272046

Resultados de FCM (simple) con apertura en área y adaptación del histograma

	mean	std
iou	0.377132	0.19476
f1	0.518873	0.209905
precision	0.473287	0.292068
accuracy	0.866084	0.0595233
recall	0.85641	0.204314

Resultados de FCM (threshold = 0.45 por que menos no habia cambio) con apertura en área y adaptación del histograma

	mean	std
iou	0.366558	0.193885
f1	0.507718	0.208752
precision	0.467925	0.292669
accuracy	0.864792	0.059048
recall	0.832137	0.214636

Resultados de FCM (simple) con apertura binaria, adaptación del histograma y opening en área

	mean	std
iou	0.439533	0.198234
f1	0.583274	0.204723
precision	0.577765	0.283441
accuracy	0.901501	0.0655886
recall	0.797216	0.234162

Resultados de FCM (threshold = 0.25) con apertura binaria, adaptación del histograma y opening en área

	mean	std
iou	0.436168	0.200064
f1	0.579458	0.206447
precision	0.575386	0.285284
accuracy	0.9012	0.065409
recall	0.789347	0.234885

Resultados de FCM (simple) con apertura binaria y filtro gaussiano

	mean	std
iou	0.449819	0.231084
f1	0.58527	0.228024
precision	0.632938	0.292495
accuracy	0.900511	0.0832081
recall	0.737613	0.275606

Resultados de FCM (threshold = 0.25) con apertura binaria y filtro gaussiano

	mean	std
iou	0.442555	0.235778
f1	0.575977	0.237933
precision	0.636301	0.301989
accuracy	0.900828	0.0832787
recall	0.683621	0.296212

Resultados de FCM (simple) con apertura en área y filtro gaussiano

	mean	std
iou	0.492137	0.23423
f1	0.625384	0.225294
precision	0.678883	0.302424
accuracy	0.909404	0.0813395

	mean	std
recall	0.771316	0.252718

Resultados de FCM (threshold = 0.25) con apertura en área y filtro gaussiano

	mean	std
iou	0.480827	0.24135
f1	0.611565	0.240507
precision	0.674853	0.313602
accuracy	0.908992	0.0808518
recall	0.713841	0.280797

Resultados de FCM (simple) con apertura binaria, apertura en área y filtro gaussiano

	mean	std
iou	0.486966	0.242643
f1	0.618106	0.233243
precision	0.675286	0.317727
accuracy	0.909146	0.0742083
recall	0.774254	0.252611

Resultados de FCM (threshold = 0.25) con apertura binaria, apertura en área y filtro gaussiano

	mean	std
iou	0.477186	0.250463
f1	0.605349	0.248521
precision	0.670173	0.327936
accuracy	0.908301	0.0745995
recall	0.707859	0.279038

Resultados de FCM (simple) con apertura binaria, adaptación del histograma y filtro gaussiano

	mean	std
iou	0.504703	0.21734
f1	0.640941	0.213398
precision	0.648524	0.292695
accuracy	0.918512	0.0639917
recall	0.804142	0.225611

Resultados de FCM (threshold = 0.25) con apertura binaria, adaptación del histograma y filtro gaussiano

	mean	std
iou	0.495991	0.226867
f1	0.62954	0.228345
precision	0.641458	0.302026
accuracy	0.91726	0.0646175
recall	0.773269	0.242168