MUHAMMAD NOMAN SHAHID

COWLAR ASSESSMENT PART 2

Image segmentation using lazy snapping via a custom K-MEANS algorithm

1. Introduction

The project aims to perform lazy snapping for separation of background and foreground. The data contains images to be segmented and images that have blue strokes for background and red strokes for foreground. There are 3 images. Two of those images have been two different details of strokes. The project aims at using the strokes to extract seeds from the actual images and use them to separate the foreground and background using a custom implementation of kmeans algorithm.

NOTE: You can run the code from both the notebook and the '*main.py*' file in the root directory. Please read the README.md files for further instructions and information on the difference between the two.

2. Project Structure

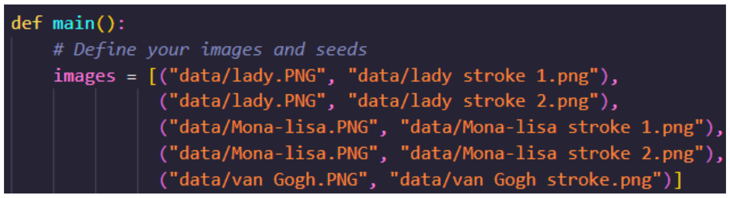
```
Lazy-snapping/
├── data/            # Raw data (not included in the git repository)
|   └── # all the images are here
|
├── notebooks/       # Jupyter notebooks
|   └── lazy_snap.ipynb
|
├── src/             # Source code
|   ├── __init__.py
|   ├── clustering.py
|   ├── display.py
|   ├── kmeans.py
|   ├── lazy_snapping.py
|   └── probability_proc.py
|
├── tests/           # Unit tests
|   ├── __init__.py
|   ├── test_clustering.py
|   ├── test_kmeans.py
|   └── test_probability_proc.py
```

```
├── .gitignore
├── README.md
├── requirements.txt
├── Report.pdf
└── main.py
```

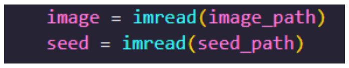
3. Data Processing

The data preprocessing stage involves several steps:

-Data Loading: The data is loaded using **cv2.imread**. Since we only need the imread function in the main file we only import that. It reads the data into an array with **BGR** values, so if an image has an alpha channel, it will be discarded and we will not have to handle any edge cases in our code (Like the **van Gogh stroke.png**) We create a list of 2-tuples each containing the image path as first element and stroke image path as the second. We loop over this list to read all the files one by one and apply the clustering in a nested loop.



```python
def main():
    # Define your images and seeds
    images = [("data/lady.PNG", "data/lady stroke 1.png"),
              ("data/lady.PNG", "data/lady stroke 2.png"),
              ("data/Mona-lisa.PNG", "data/Mona-lisa stroke 1.png"),
              ("data/Mona-lisa.PNG", "data/Mona-lisa stroke 2.png"),
              ("data/van Gogh.PNG", "data/van Gogh stroke.png")]
```



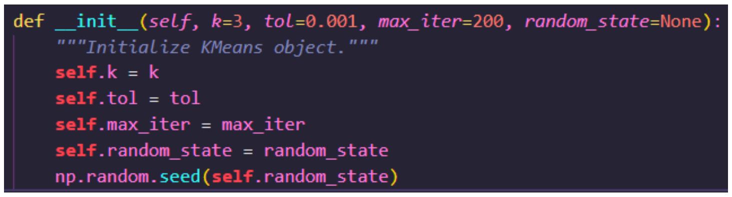
```python
    image = imread(image_path)
    seed = imread(seed_path)
```

Our next step is to separate the blue and black pixels from the stroke (seed) images, and we are not going to use any hardcoded approach for separating the specific colors. We will be clustering the stroke images so in order to do that we will first need to implement our actual K-means algorithm.

4. KMEANS

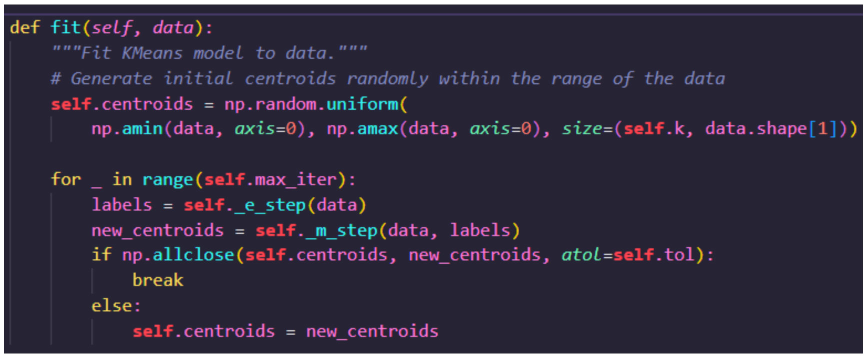
A custom Kmeans clustering algorithm is implemented.

- Initialization: The init method is defined with 4 instances i.e The value of k, the tolerance, max iterations and random state. We have to define random state just for the sake of consistency in our output. We will talk about it in the seed extraction part.



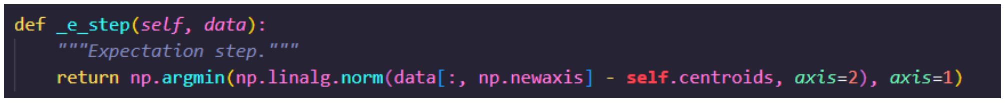
```python
def __init__(self, k=3, tol=0.001, max_iter=200, random_state=None):
    """Initialize KMeans object."""
    self.k = k
    self.tol = tol
    self.max_iter = max_iter
    self.random_state = random_state
    np.random.seed(self.random_state)
```

- Fitting the data: The **fit** method is called with the data to be clustered. This method initializes the centroids randomly within the range of the data. Then, it enters a loop for a maximum of **max_iter** iterations. In each iteration, it performs the E-step and M-step. If the centroids do not change significantly (as determined by the **tol** parameter), it breaks out of the loop.



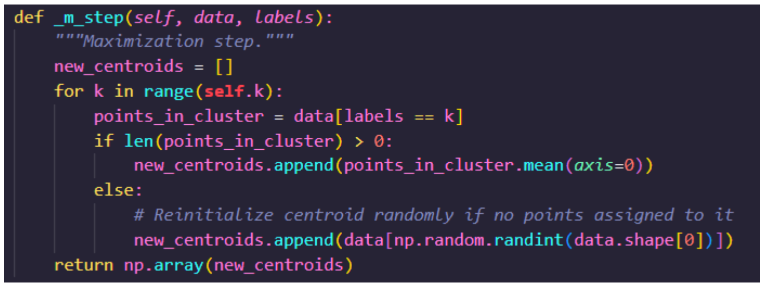
```python
def fit(self, data):
    """Fit KMeans model to data."""
    # Generate initial centroids randomly within the range of the data
    self.centroids = np.random.uniform(
        np.amin(data, axis=0), np.amax(data, axis=0), size=(self.k, data.shape[1]))

    for _ in range(self.max_iter):
        labels = self._e_step(data)
        new_centroids = self._m_step(data, labels)
        if np.allclose(self.centroids, new_centroids, atol=self.tol):
            break
        else:
            self.centroids = new_centroids
```

- E-step: The **_e_step** method (expectation) is called with the data. This method calculates the Euclidean distance from each data point to each centroid and assigns each data point to the cluster of the closest centroid. np.arming returns the index of the lowest value (distance in this case).



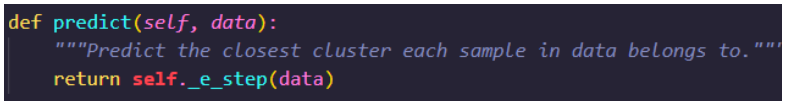
```python
def _e_step(self, data):
    """Expectation step."""
    return np.argmin(np.linalg.norm(data[:, np.newaxis] - self.centroids, axis=2), axis=1)
```

- M-step: The **_m_step** method (maximization) is called with the data and the labels (cluster assignments) from the E-step. This method calculates the new centroids as the mean of the data points in each cluster. If a cluster has no points, it reinitializes the centroid randomly.



```python
def _m_step(self, data, labels):
    """Maximization step."""
    new_centroids = []
    for k in range(self.k):
        points_in_cluster = data[labels == k]
        if len(points_in_cluster) > 0:
            new_centroids.append(points_in_cluster.mean(axis=0))
        else:
            # Reinitialize centroid randomly if no points assigned to it
            new_centroids.append(data[np.random.randint(data.shape[0])])
    return np.array(new_centroids)
```

- Predict: The **predict** method is called with new data to assign each data point to a cluster. This method simply calls the **_e_step** method with the new data.

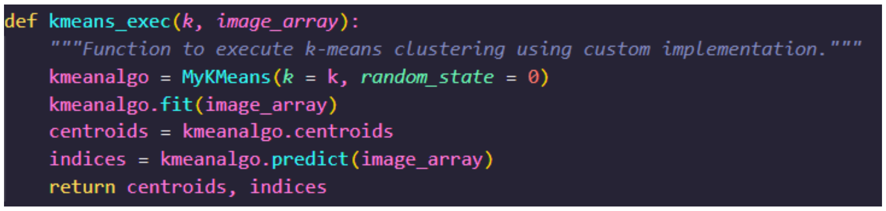


```python
def predict(self, data):
    """Predict the closest cluster each sample in data belongs to."""
    return self._e_step(data)
```

We use broadcasting techniques and vector operations to perform most of our computations to make things efficient.

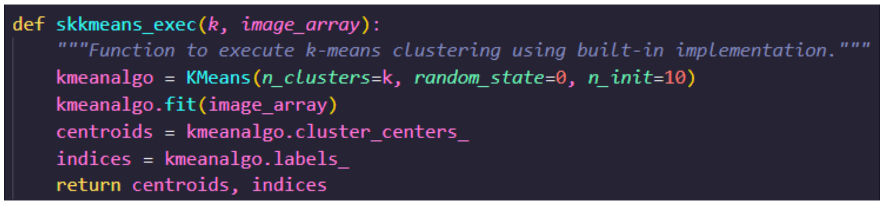
5. Execution flow

First, lets get the kmeans functions out of the way. We define two functions that take the k value and the image_array. They both perform clustering on the data and return the centroids as well their indices in the image (labels). The first one called **kmeans_exec,** calls our custom implementation of the function.



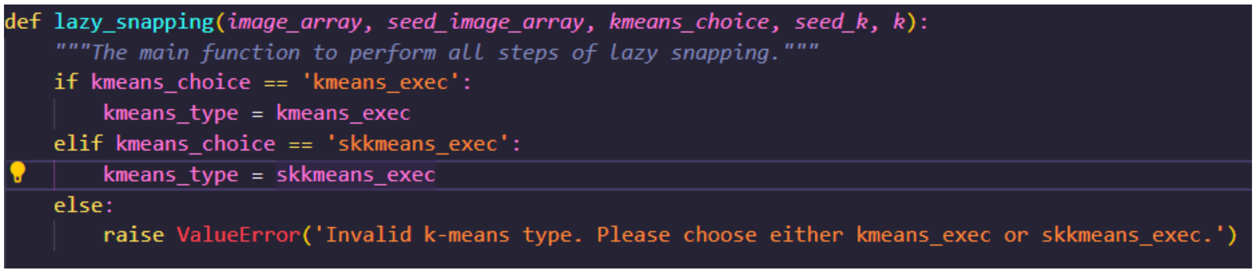
```
def kmeans_exec(k, image_array):
    """Function to execute k-means clustering using custom implementation."""
    kmeanalgo = MyKMeans(k = k, random_state = 0)
    kmeanalgo.fit(image_array)
    centroids = kmeanalgo.centroids
    indices = kmeanalgo.predict(image_array)
    return centroids, indices
```

The second one called **skkmeans_exec** calls the built-in sklearn.clustering kmeans function.



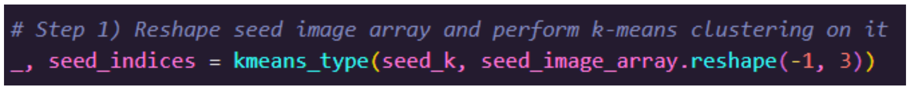
```
def skkmeans_exec(k, image_array):
    """Function to execute k-means clustering using built-in implementation."""
    kmeanalgo = KMeans(n_clusters=k, random_state=0, n_init=10)
    kmeanalgo.fit(image_array)
    centroids = kmeanalgo.cluster_centers_
    indices = kmeanalgo.labels_
    return centroids, indices
```

We perform all our functions inside of a function called **lazy_snapping.** This function is a helper function that calls all the other functions in the right order. The first thing we do here is check which kmeans function we want to use.



```
def lazy_snapping(image_array, seed_image_array, kmeans_choice, seed_k, k):
    """The main function to perform all steps of lazy snapping."""
    if kmeans_choice == 'kmeans_exec':
        kmeans_type = kmeans_exec
    elif kmeans_choice == 'skkmeans_exec':
        kmeans_type = skkmeans_exec
    else:
        raise ValueError('Invalid k-means type. Please choose either kmeans_exec or skkmeans_exec.')
```

- Clustering strokes: We reshape the **seed_image_array** and make it 2D. The first dimension is the features of the image and the second is the color channels. Then we pass it into the kmeans function. Seed_k is the number of clusters for our seed image. We choose **3** for this since we have blue pixels, red pixels and black pixels.



```
# Step 1) Reshape seed image array and perform k-means clustering on it
_, seed_indices = kmeans_type(seed_k, seed_image_array.reshape(-1, 3))
```

We do not need the centroids here for any further computations so we label it as an underscore.

- Extracting seeds: Then we reshape the seed_indices back to the shape of the image without the third dimension. We need to do this because we will use this array to filter (extract seeds from) our image_array.

```python
#reshape the image back to original shape without the third dimension
seed_indices = seed_indices.reshape(image_array.shape[:2])
```

After this we check the unique seed indices and their counts, then we delete the index with the highest count since it is the black pixels which we do not need.

```python
# Filter out the most frequent value (black pixels) in the seed image array
unique, counts = np.unique(seed_indices, return_counts=True)
unique = np.delete(unique, np.argmax(counts))
```

The unique array now only has two label values and they always come in the same order. The reason for that is the random_state that we have defined in our kmeans class. It does nothing useful for the quality of the segmentation but it helps that the resulting images are always in the same order so they are visually easy to compare.

After this we use these unique labels to filter the actual image array to extract the seeds.

```python
# Step 2) extract the foreground and background pixels from the image array
cluster1_pixels = image_array[seed_indices == unique[0]]
cluster2_pixels = image_array[seed_indices == unique[1]]
```

Then we perform kmeans on our “cluster” images. We try different k values here.

```python
# Step 3) Perform k-means clustering on both seed clusters separately
cluster1_centroids, cluster1_indices = kmeans_type(k, cluster1_pixels)
cluster2_centroids, cluster2_indices = kmeans_type(k, cluster2_pixels)
```

- Calculate the binary weight mask: Then we calculate a binary weight mask to separate the foreground and background. In order to do that we first define a **weight** function that calculates the relative frequency of each unique value in the index array.

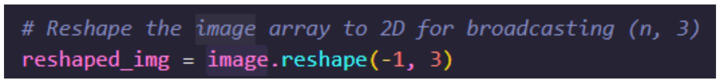
```python
def weight(index):
    """Calculate the weight of each pixel in the image."""
    _, counts = np.unique(index, return_counts=True)
    # return the relative frequency of each unique value in the index array
    return counts / len(index)
```

Then we define the **calc_bin_mask** function that takes indices and centroids of two clusters as well as the original image as parameters.

We call the weight function for both clusters’ indices inside of this function

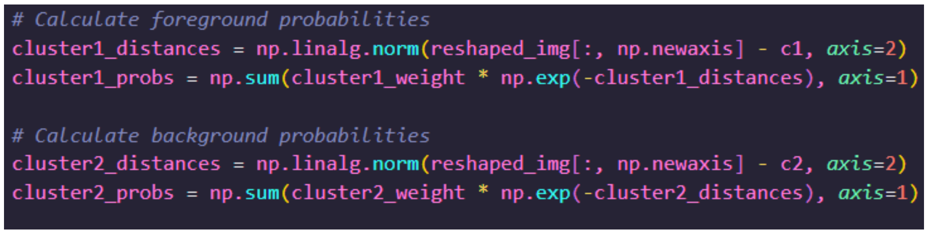
```python
def calc_bin_mask(image, c1, i1, c2, i2):
    """Calculate the probability of each pixel in the image belonging to foreground or background."""
    # Get the clusters' weights
    cluster1_weight = weight(i1)
    cluster2_weight = weight(i2)
```

We reshape our image to 2D so we can perform broadcasting in order to calculate the Euclidean distances. We add another axis in the step after this one for the purpose of broadcasting as well.



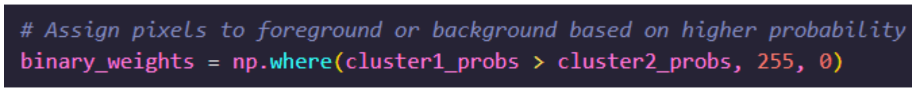
```
# Reshape the image array to 2D for broadcasting (n, 3)
reshaped_img = image.reshape(-1, 3)
```

We calculate both the foreground and background probabilities (likelihood) using a function of an exponent of negative of the distance. We add another axis in the distance step for the purpose of broadcasting.



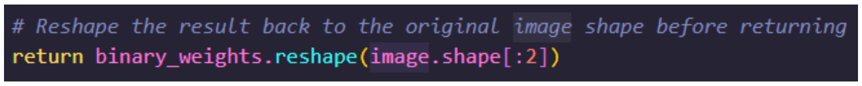
```
# Calculate foreground probabilities
cluster1_distances = np.linalg.norm(reshaped_img[:, np.newaxis] - c1, axis=2)
cluster1_probs = np.sum(cluster1_weight * np.exp(-cluster1_distances), axis=1)

# Calculate background probabilities
cluster2_distances = np.linalg.norm(reshaped_img[:, np.newaxis] - c2, axis=2)
cluster2_probs = np.sum(cluster2_weight * np.exp(-cluster2_distances), axis=1)
```

Then we assign the white pixels to the foreground or the background and black to the other. The reason we do not know which one we are assigning is because we used a clustering algorithm (kmeans) to separate the pixels. They could be any shade of blue or red so we go without trying to determine them as it will not affect the output.



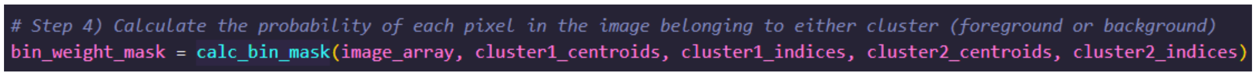
```
# Assign pixels to foreground or background based on higher probability
binary_weights = np.where(cluster1_probs > cluster2_probs, 255, 0)
```

We reshape the binary mask to the shape of the image and return it.



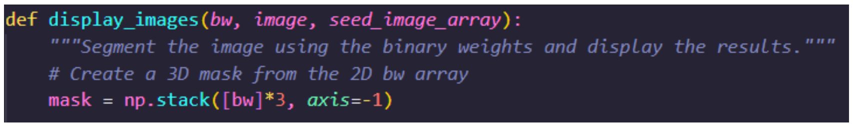
```
# Reshape the result back to the original image shape before returning
return binary_weights.reshape(image.shape[:2])
```

This is how we call it in the **lazy_snapping** function:



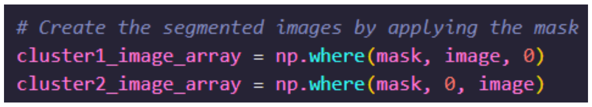
```
# Step 4) Calculate the probability of each pixel in the image belonging to either cluster (foreground or background)
bin_weight_mask = calc_bin_mask(image_array, cluster1_centroids, cluster1_indices, cluster2_centroids, cluster2_indices)
```

- Display: After this we call the display function to display the images. We first convert the binary mask we got in previous step from 2D to 3D since our image array is 3D.



```
def display_images(bw, image, seed_image_array):
    """Segment the image using the binary weights and display the results."""
    # Create a 3D mask from the 2D bw array
    mask = np.stack([bw]*3, axis=-1)
```

We create two image arrays by applying the mask. The first array is created by setting the white pixels to the original image pixels and black pixels to 0 (still black) and the second vice versa. These arrays represent the foreground and background (or vice versa since we never specified beforehand).



```
# Create the segmented images by applying the mask
cluster1_image_array = np.where(mask, image, 0)
cluster2_image_array = np.where(mask, 0, image)
```

After that we convert them back to RGB (They were in BGR since that is the default in cv2)

```python
# Convert the images to RGB
stroke_img = cvtColor(seed_image_array, COLOR_BGR2RGB)
og_image = cvtColor(image, COLOR_BGR2RGB)
cluster1_image = cvtColor(cluster1_image_array, COLOR_BGR2RGB)
cluster2_image = cvtColor(cluster2_image_array, COLOR_BGR2RGB)
```

Then we create 4 subplots and display the stroke image, og_image, cluster1 and cluster2 images, in that order on each row.

```python
# Create a figure with 4 subplots
fig, axs = plt.subplots(1, 4, figsize = (16, 4))

# Display each image on a separate subplot
for ax, image in zip(axs, [stroke_img, og_image, cluster1_image, cluster2_image]):
    ax.axis('off')
    ax.imshow(image)

plt.show()
```

Here is how we call it in the **lazy_snapping** function.

```python
# Step 4) Display the results
display_images(bin_weight_mask, image_array, seed_image_array)
```

Back in our main function we perform the clustering over different values of clusters (**k**) keeping the **k** value of stroke image clustering at 3.

```python
N_values = [2, 32, 64, 96, 128]

# Loop over your images and seeds
for image_path, seed_path in images:
    # Read the images
    image = imread(image_path)
    seed = imread(seed_path)

    # Perform the lazy snapping
    for N in N_values:
        print(f'Result with {N} clusters: ')
        lazy_snapping(image, seed, 'kmeans_exec', 3, N)
        lazy_snapping(image, seed, 'skkmeans_exec', 3, N)
```

6. Testing

We create basic unit tests for most of the functions. We check things like, shape of the data, expected output and type of the data being returned. They also check the expected output of the model. We will not go to explain all the tests in this report. They are very straightforward and easy to understand. We have not created any tests for the visualizations since those require the checking of plots manually as well.

We use **pytest** and **pytest-cov** for testing. The tests are located in the **'tests'** directory.

7. Results Analysis

Both our custom kmeans and the built-in kmeans (The one we labelled skkmeans) have the exact same quality of results. That is why we are not going to compare them in this report since there are too many iterations. We are also not going to show the results for some variations of the number of clusters for reasons mentioned later on. However, they are all present in the notebook if one is interested. We do not even need to run the notebook. We can just go to the last cell and see all the different variations of results.

Comparison:

Our custom kmeans has proven to be at least **10%** and at most **150%** percent faster than the built-in function. It is the highest with the lowest k values (number of clusters) and keeps on decreasing as the number of clusters go up.

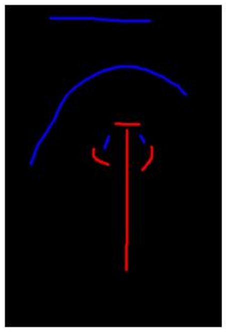
So, from 2 to 96 clusters we go from around **150%** faster down to around **10%.**

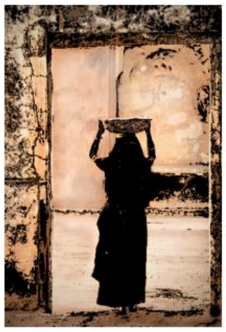
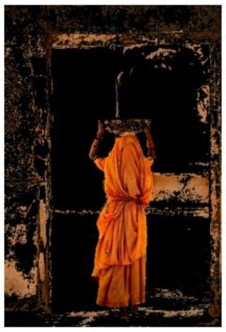
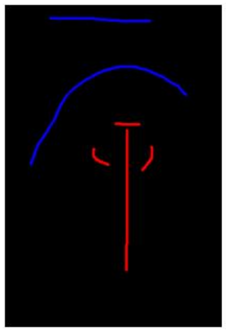
But with 128 clusters the built-in library is faster by a small amount. Not only that, the last van Gogh image had complex details and the built-in library ran fairly faster than our implementation. From this trend we find that our implementation is only better for simpler cases and less clusters.

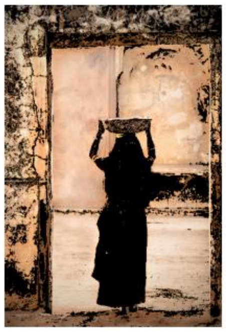
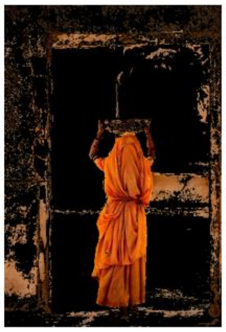
- For k=2:

The first image does not have many complex structures. It is easy to distinguish between foreground and background so the separation is fairly acceptable. The rest of the images show really bad results.

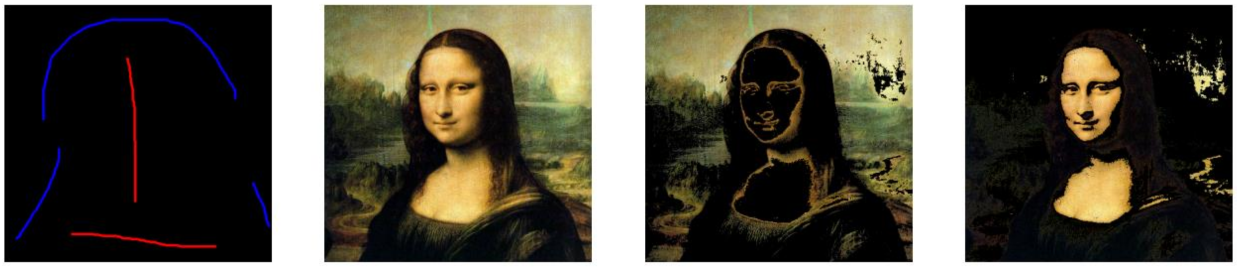
Not to mention the lady image shows no difference at all with these two different strokes.

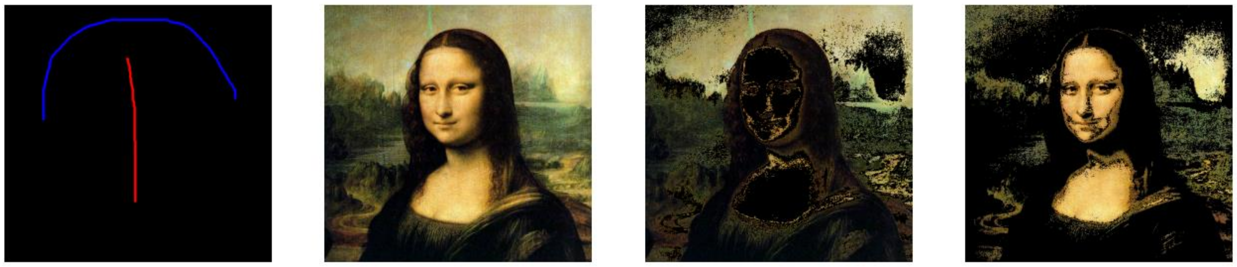


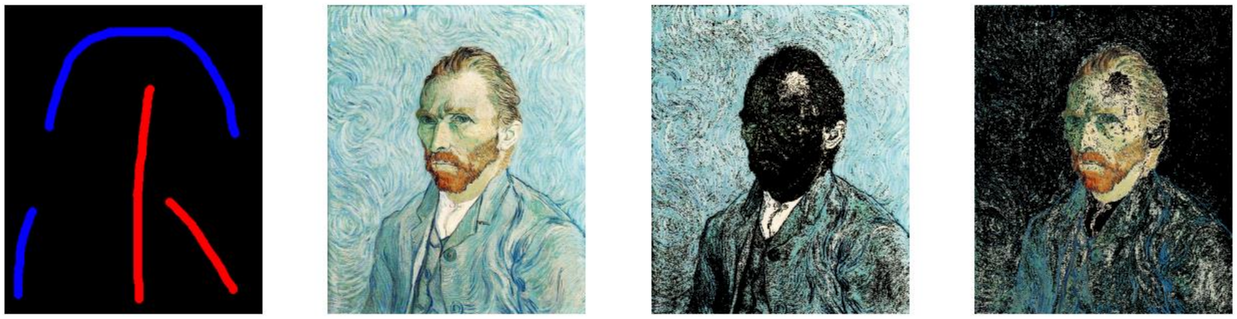



The mona lisa images show really bad result but we see that the first stroke image gives a better separation.



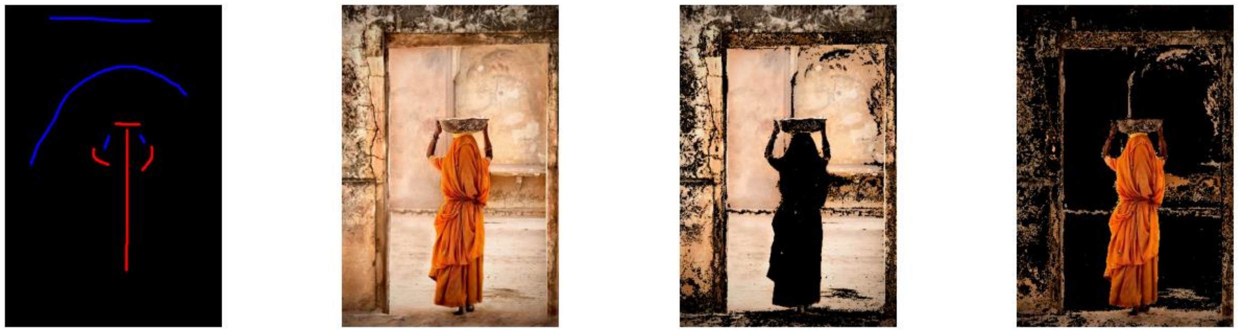


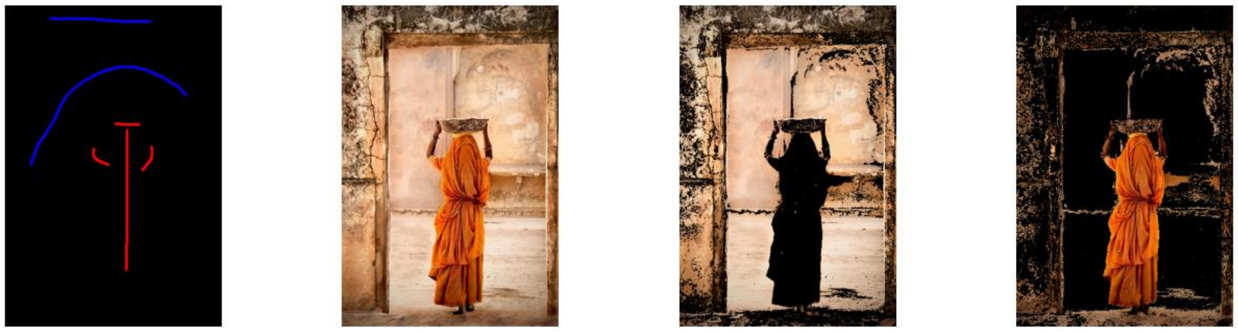
This one is better than the mona lisa but can be improved a lot.



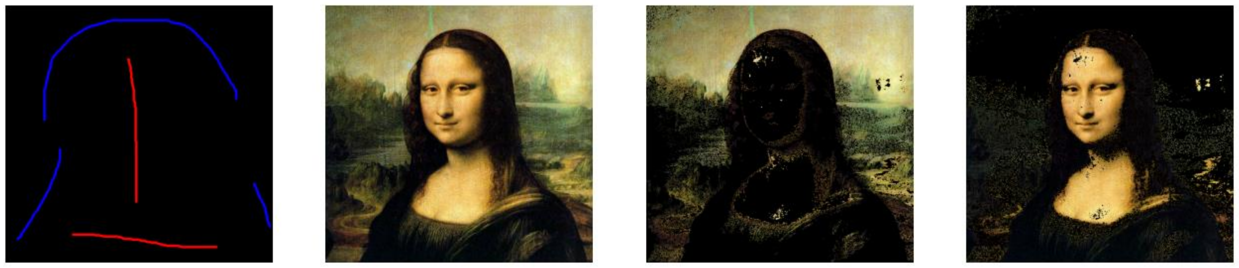
- For k=32:

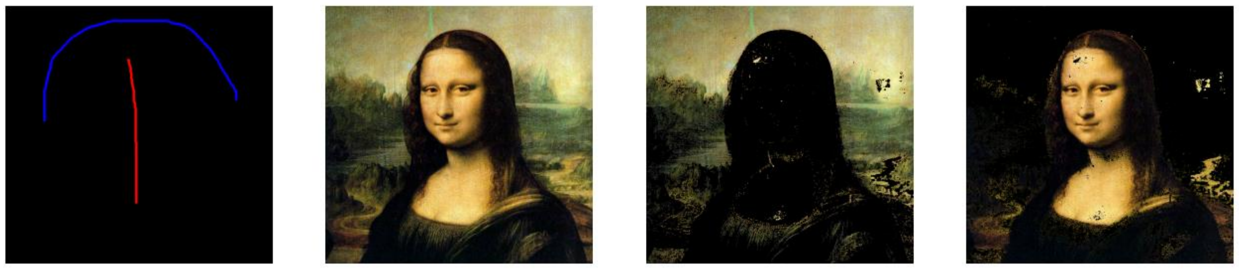
The results improved significantly with this number. There is very little difference in both stroke images. It can be seen if one looks carefully enough. The first one looks a bit better.



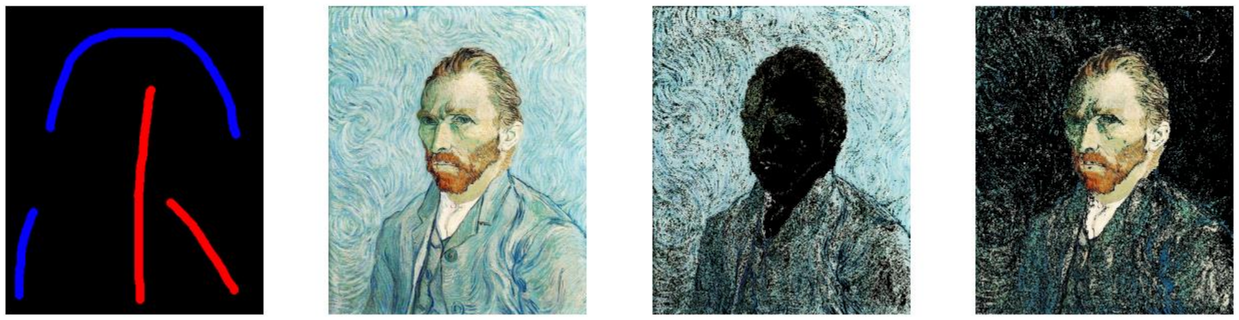


The results have improved significantly for mona lisa from 2 to 32 clusters. Also the first image with more defining strokes gives better results.





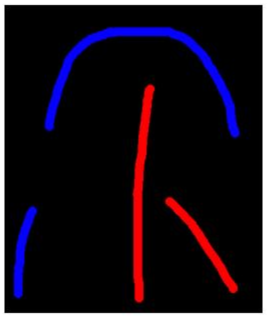
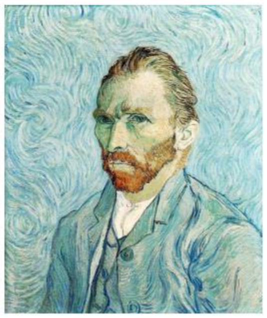
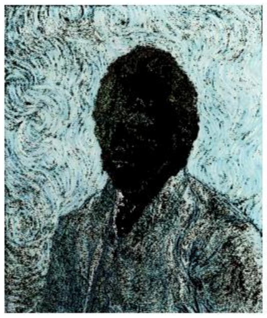
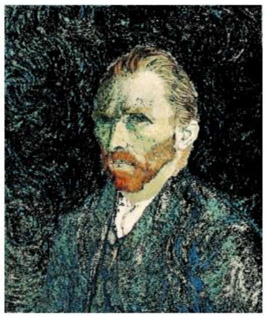
Better results as seen in the other images as well for van gogh.



- For k=64:

For this section we will only mention the van gogh image. The rest have insignificant improvements while the computation time has increased two-fold.

For k=96 and k=128, the results do not improve enough to be visually noticeable but the computation time increases a lot so they are not worth mentioning in this report (They are available in the notebook).

This is the only image we see some noticeable improvements in at this point.

8. Conclusion

The project successfully implements a custom KMEANS algorithm. Giving too less detail in the stroke image can lead to poor results. There should be enough strokes to distinguish any complexities in the image. The computation time of the custom Kmeans is better with smaller number of clusters and less complex images. However, the results for the custom kmeans and the built-in library function by scikit-learn have the exact same quality which is proof that the algorithm itself works correctly for the custom implementation.