# IMAGE PROCESSING
# 3.ASSIGNMENT REPORT
# IMAGE SEGMENTATION

**Prepared by: Sertaç Güler**
**School ID: 21986764**

# Problem Definition:

In image processing sometimes we need some parts to show greater influence. Because some parts may be more important to us than other parts. For example if you want to detect an object you wouldn't require background details. You just need the object. Here comes image segmentation into work. Image segmentation groups parts of pixels that go together. This means similar pixels will be grouped together and create some clusters. We want these clusters' between-class variance to be maximized so that it will look distinctive and intra-class variance to be minimized so that we will not include other pixels from other classes.

# Explanation of Code:

I needed to import CV2 to read and write images and process them as RGB images. I imported numpy to do numerical operations , matplotlib to organise and display image results and os library to handle file paths.
Then I defined a function that extracts pixel features. This function returns rgb features and rgb features with location information. It normalizes the image before extracting features.

```
def extractPixelFeatures(image):
    numRows, numCols, _ = image.shape
    rgbFeatures = np.zeros((numRows * numCols, 3))
    locationFeatures = np.zeros((numRows * numCols, 2))

    index = 0
    for row in range(numRows):
        for col in range(numCols):
            rgb = image[row, col] / 255.0
            location = np.array([row / numRows, col / numCols])
            rgbFeatures[index] = rgb
            locationFeatures[index] = location
            index += 1

    rgbLocationFeatures = np.hstack((rgbFeatures, locationFeatures))
    return rgbFeatures, rgbLocationFeatures
```

I created a function called KMeans. This function is used to sort similar parts of the image into groups, which we call 'clusters'. To do this, the function looks at the features of the image, like colors and positions of pixels, and decides how many

groups 'k' to make. The function then repeatedly tries to find the best center point for each group. It looks at every pixel and decides which group center it is closest to. This process keeps going until the group centers stop changing much, or it has tried a lot of times. At the end of this, the image's pixels are sorted into 'k' different groups, helping us see patterns in the image.

```
def KMeans(features, k):
    numSamples, numFeatures = features.shape
    centroids = features[np.random.choice(numSamples, k, replace=False)]

    for iteration in range(100):
        distances = np.zeros((k, numSamples))
        for i in range(k):
            for j in range(numSamples):
                distance = np.sqrt(np.sum((features[j] - centroids[i]) ** 2))
                distances[i, j] = distance

        closestCentroid = np.argmin(distances, axis=0)
        newCentroids = np.zeros((k, numFeatures))

        for i in range(k):
            clusterPoints = features[closestCentroid == i]
            if len(clusterPoints) > 0:
                newCentroids[i] = np.mean(clusterPoints, axis=0)
            else:
                newCentroids[i] = features[np.random.choice(numSamples, 1, replace=False)]

        if np.array_equal(centroids, newCentroids):
            break

        centroids = newCentroids

    return closestCentroid, centroids
```

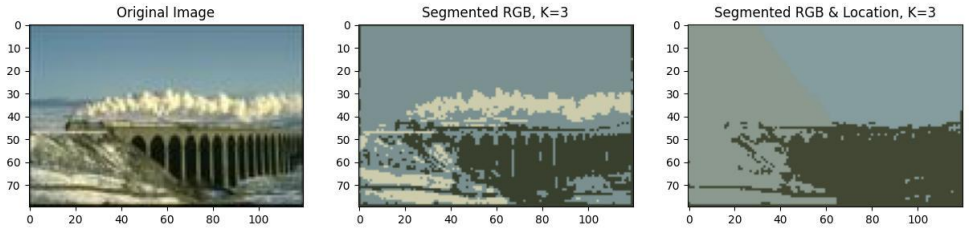
I have a function called segmentImage. It takes the image and breaks it down into different parts or 'segments' based on similar features, like colors. This is done using the 'k' groups we figured out in the custom K-means function. Here's how it works: For every pixel in the image, the function looks at which group it belongs to and changes its color to match the color of the center of that group. By doing this for all pixels, the original image gets split up into different color-coded segments. Each segment represents an area where the pixels are similar to each other. This helps us see different parts of the image more clearly, almost like looking at a map with different colored areas.

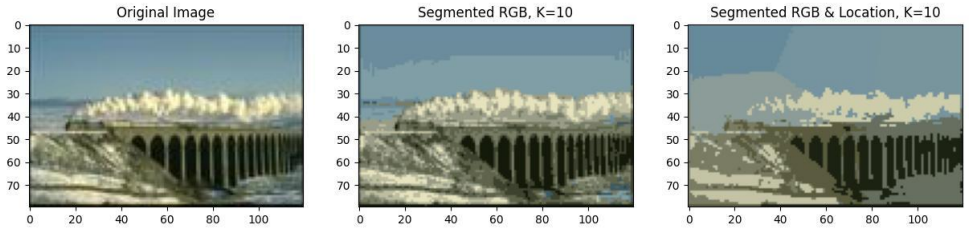
```python
def segmentImage(image, features, k):
    labels, centroids = KMeans(features, k)
    numRows, numCols, _ = image.shape
    segmentedImg = np.zeros_like(image)

    for i in range(numRows):
        for j in range(numCols):
            index = i * numCols + j
            label = labels[index]
            segmentedImg[i, j] = centroids[label][:3] * 255

    return segmentedImg
```

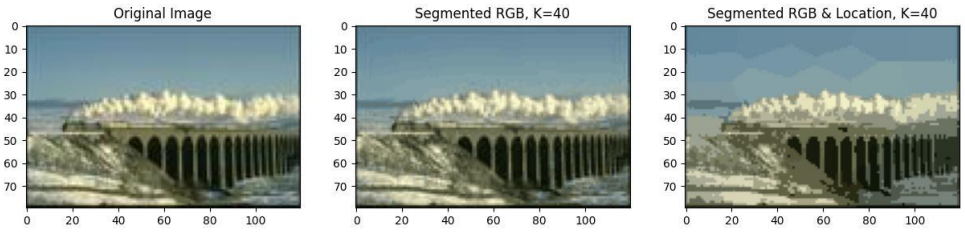
I couldn't create super pixel features which can be used in KMeans algorithm to get better segmentation results.

## RESULTS:

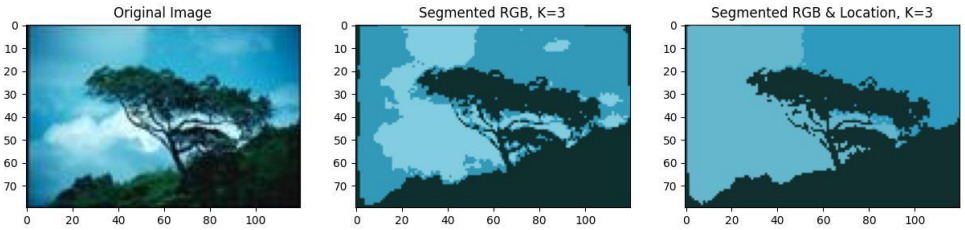
I tested my program using 5  different images from the Berkeley Segmentation Dataset. I continued to increase the number of clusters. I used 6 different cluster numbers namely 3, 7, 10, 15, 25, 40. I expected the resulting images to be more similar to the original image as the cluster size increases.
I will analyze the results only showing the clusters with 3, 10 and 40.

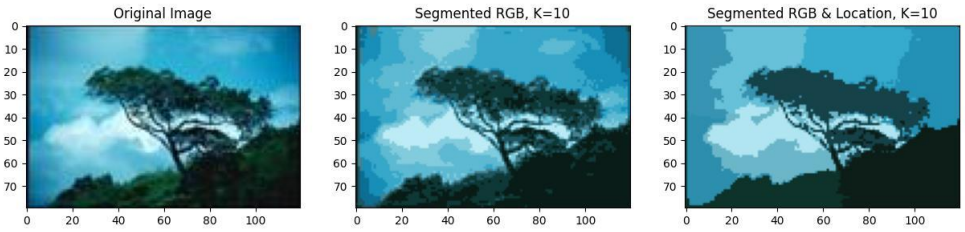


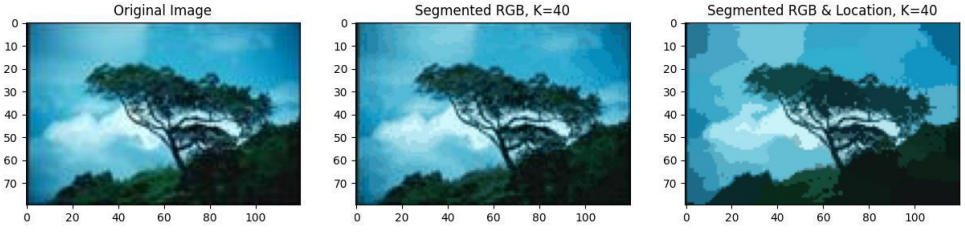





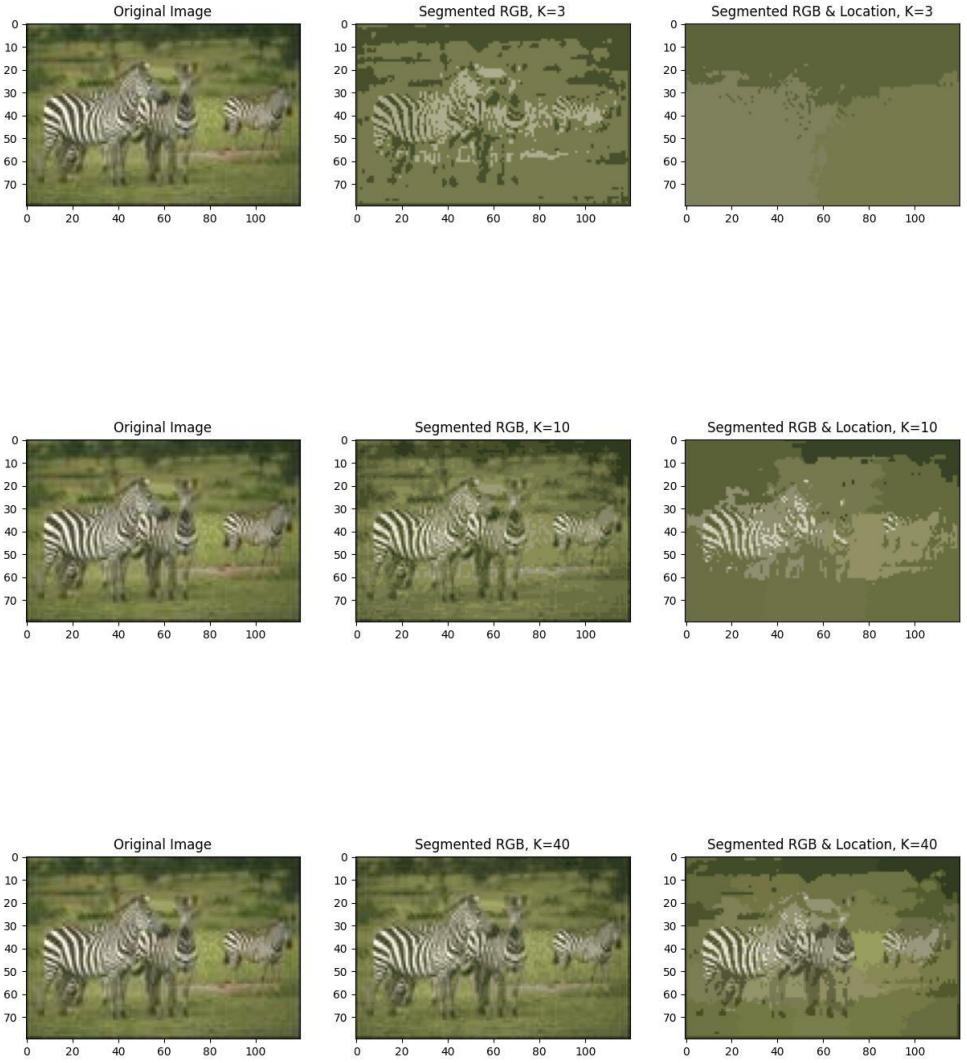

For the first image I have a train driving on the bridge. RGB image with location feature(cluster = 3) gave a very unsatisfactory result that does not even explain bridge or train. Whereas only the RGB feature image is very satisfactory.As we increase the number of clusters we converge to original image faster. However for some reason RGB image with location features makes a sharper segmentation which prevents it from converging to original image. The reason could be adding



location features may increase inter-class variance and minimize the intra-class variance. For other images I have similar results. For high detail images if we choose the cluster number as 3 we don't have enough class numbers to represent the image fully.




Note : You can find the full cluster size experiment results and other image experiment results in the image output folder.