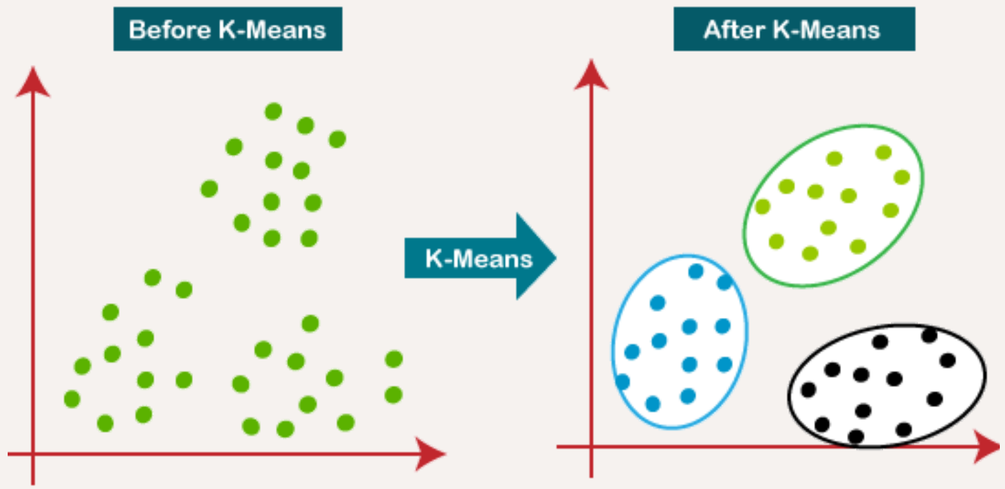
Image Segmentation

with K-means Clustering

Submitted To
Mr. Pankaj Dawadi

Image Segmentation

Image segmentation is a computer vision technique that involves dividing an image into multiple segments or regions with similar characteristics. The goal of image segmentation is to simplify or change the representation of an image into something more meaningful and easier to analyze.



k-means Clustering

K-means clustering is a machine learning algorithm used for grouping data points into K number of clusters based on their similarity. The algorithm works by iteratively assigning data points to the closest centroid and updating the centroids until convergence.

K-means clustering is an important algorithm which can be used for customer segmentation,Image Compression,Document Clustering and Anomaly Detection

```python
class KMeans():

    def __init__(self, K=5, max_iters=100, plot_steps=False):
        self.K = K
        self.max_iters = max_iters
        self.plot_steps = plot_steps

        # list of sample indices for each cluster
        self.clusters = [[] for _ in range(self.K)]
        # the centers (mean feature vector) for each cluster
        self.centroids = []
```

- **k**: No of clusters to create
- **max_iters**: maximum number of iterations to perform
- **plot_steps**: whether to plot the clustering process at each iteration
- **clusters:** a list of empty lists, where each inner list will store the indices of the samples assigned to each cluster
- **centroids**: a list of empty lists, where each inner list will store the mean feature vector for each cluster

```python
def predict(self, X):
    self.X = X
    self.n_samples, self.n_features = X.shape

    # initialize
    random_sample_idxs = np.random.choice(self.n_samples, self.K, replace=False)
    self.centroids = [self.X[idx] for idx in random_sample_idxs]

    # Optimize clusters
    for _ in range(self.max_iters):
        # Assign samples to closest centroids (create clusters)
        self.clusters = self._create_clusters(self.centroids)
        if self.plot_steps:
            self.plot()

        # Calculate new centroids from the clusters
        centroids_old = self.centroids
        self.centroids = self._get_centroids(self.clusters)

        # check if clusters have changed
        if self._is_converged(centroids_old, self.centroids):
            break

        if self.plot_steps:
            self.plot()

    # Classify samples as the index of their clusters
    return self._get_cluster_labels(self.clusters)
```

- method to predict which cluster each sample in a dataset belongs to.
- The predict method takes in a dataset X and returns a list of cluster labels, where each label corresponds to the index of the cluster
- iterates for a maximum of max_iters times to optimize the clusters.
- In each iteration, it first assigns each sample in X to the closest centroid to create the clusters.
- Then, calculates the mean feature vector for each cluster to update the centroids.
- Stops if the centroid has not changed

```python
def _get_cluster_labels(self, clusters):
    # each sample will get the label of the cluster it was assigned to
    labels = np.empty(self.n_samples)

    for cluster_idx, cluster in enumerate(clusters):
        for sample_index in cluster:
            labels[sample_index] = cluster_idx
    return labels

def _create_clusters(self, centroids):
    # Assign the samples to the closest centroids to create clusters
    clusters = [[] for _ in range(self.K)]
    for idx, sample in enumerate(self.X):
        centroid_idx = self._closest_centroid(sample, centroids)
        clusters[centroid_idx].append(idx)
    return clusters
```

- takes the clusters as input and assigns labels to each data point based on the cluster they belong to
- then loops through each cluster and assigns the cluster index to each data point in the cluster.

- takes the centroids as input and assigns each data point to the closest centroid to create clusters
- calculates the closest centroid and then appends the index of the data point to the corresponding cluster

```python
def _closest_centroid(self, sample, centroids):
    # distance of the current sample to each centroid
    distances = [euclidean_distance(sample, point) for point in centroids]
    closest_index = np.argmin(distances)
    return closest_index

def _get_centroids(self, clusters):
    # assign mean value of clusters to centroids
    centroids = np.zeros((self.K, self.n_features))
    for cluster_idx, cluster in enumerate(clusters):
        cluster_mean = np.mean(self.X[cluster], axis=0)
        centroids[cluster_idx] = cluster_mean
    return centroids
```

- takes a single data point sample and a list of centroids and returns the index of the closest centroid to the data point

- takes the clusters as input and computes the mean of each cluster to get the new centroids.
- loops through each cluster, calculates the mean of the data points in the cluster using np.mean, and assigns the resulting mean vector to the corresponding centroid in the centroids array

```python
    def _is_converged(self, centroids_old, centroids):
        # distances between each old and new centroids, fol all centroids
        distances = [euclidean_distance(centroids_old[i], centroids[i]) for i in range(self.K)]
        return sum(distances) == 0

    def plot(self):
        fig, ax = plt.subplots(figsize=(12, 8))

        for i, index in enumerate(self.clusters):
            point = self.X[index].T
            ax.scatter(*point)

        for point in self.centroids:
            ax.scatter(*point, marker="x", color='black', linewidth=2)

        plt.show()
    def cent(self):
        return self.centroids
```

✓ 0.1s

- takes two sets of centroids as input, centroids_old and centroids, and checks if they are the same

- plot() method creates a scatter plot of the data points and the centroids at each iteration of the algorithm, to help visualize the clustering process.
- cent() method simply returns the final centroids of the algorithm, which can be useful for further analysis or prediction tasks

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```
✓ 0.1s

```
pixel_values = image.reshape((-1, 3))
pixel_values = np.float32(pixel_values)
```
✓ 0.0s

```
k = KMeans(K=3, max_iters=100)
y_pred = k.predict(pixel_values)
k.cent()
```
✓ 11m 33.1s

- BGR to RGB color model
- The reshape() function is used to reshape the image into a two-dimensional array with one row per pixel and three columns corresponding to the Red, Green, and Blue (RGB) color channels.
- K-means clustering in practice.

- The y_pred array contains the predicted cluster assignments for each pixel in the pixel_values array
- The np.unique(y_pred) function returns the unique values in the y_pred array, sorted in ascending order.
- The flatten() method converts this 2D array to a 1D array by flattening the rows of the array into a single row.
- The segmented_image array is created by taking the labels array, and flattening it to a 1D array
- the resulting flattened segmented_image array is reshaped back to the shape of the input image to obtain the final segmented image.

```
masked_image = np.copy(image)
```
[22] ✓ 0.0s

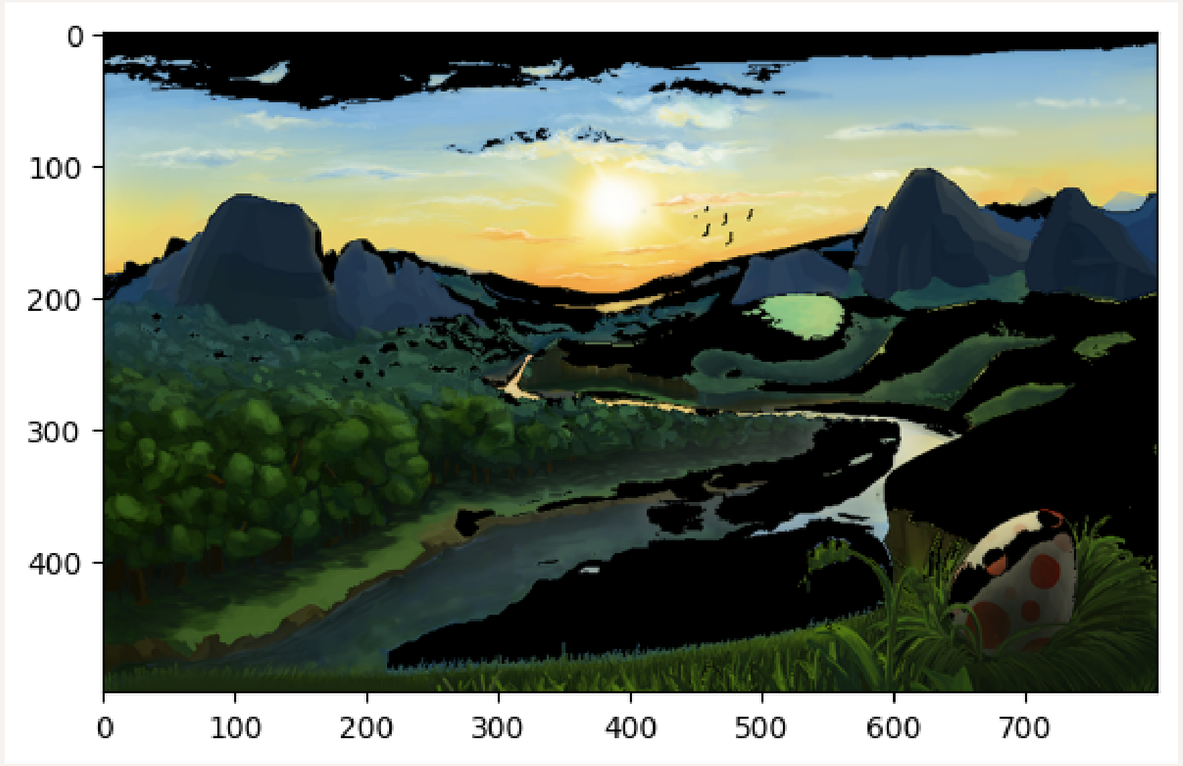
```
masked_image = masked_image.reshape((-1, 3))
```
[23] ✓ 0.0s

```
cluster = 2
masked_image[labels == cluster] = [0, 0, 0]
```
[24] ✓ 0.0s

```
masked_image = masked_image.reshape(image.shape)
```
[25] ✓ 0.0s

```
plt.imshow(masked_image)
plt.show()
```
[26] ✓ 0.4s

- After creating a copy of the original input image and reshaping it into a 2D array with 3 channels
- It select all the pixels in the masked_image array that belong to the specified cluster index .
- These pixels are then set to the RGB value [0, 0, 0], effectively masking them out and replacing them with black pixels.
- Finally, the masked image is reshaped back into its original shape and assigned to the variable masked_image.


0
100
200
300
400
0
100
200
300
400
500
600
700


0
200
400
600
800
1000
0
200
400
600
800
1000

Thank You