Kmeans Algorithm

Kmeans algorithm is an iterative algorithm that tries to partition the dataset into *K*pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to **only one group**. It tries to make the inter-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

The way kmeans algorithm works is as follows:

1. Specify number of clusters *K*.

    1. n_cluster = 15

2. Initialize centroids by first shuffling the dataset and then randomly selecting *K* data points for the centroids without replacement.

```
def get_random_identifiers(x):
    indexes = np.random.randint(x.shape[0], size=n_cluster)
    return [{i: x[item]} for i, item in enumerate(indexes)]
```

3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.

- Compute the sum of the squared distance between data points and all centroids.

- Assign each data point to the closest cluster (centroid).
  -

```
def cluster_on_euclidean_distance(x, means: List[dict] = None):
    euclid = []
    for mean in means:
        euclid.append(euclidean(x, list(mean.values())[0]))
    min_dist = euclid.index(min(euclid))
    return list(means[min_dist].keys())[0]
```

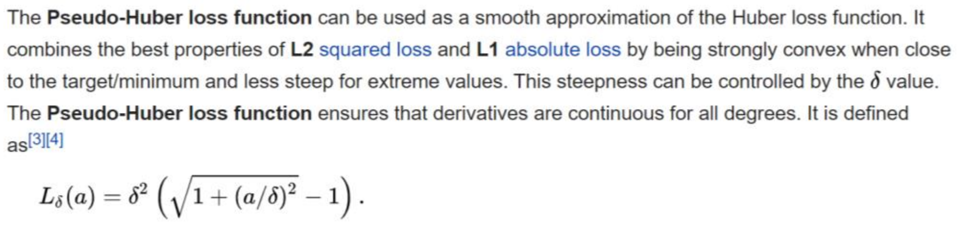
-

- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.
  -

```
def get_distance_euclidean(x, n, y=None, means=None):
    if n < 10:
        if means is None and y is None:
            means = get_random_identifiers(x)
            y = get_distance_euclidean_first_round(x, means)
        labeled_dictionary = get_label_dict(x, y)
        new_means = []
        for mean in means:
            new_means.append(
                {list(mean.keys())[0]:
get_huber_index(labeled_dictionary[list(mean.keys())[0]],

list(mean.values())[0])})
        new_y = get_distance_euclidean_first_round(x, new_means)
        return get_distance_euclidean(x, n + 1, new_y, new_means)
    else:
        return means, y
```

Identifiers



The **Pseudo-Huber loss function** can be used as a smooth approximation of the Huber loss function. It combines the best properties of **L2** squared loss and **L1** absolute loss by being strongly convex when close to the target/minimum and less steep for extreme values. This steepness can be controlled by the δ value. The **Pseudo-Huber loss function** ensures that derivatives are continuous for all degrees. It is defined as[3][4]

$$L_\delta(a) = \delta^2 \left(\sqrt{1 + (a/\delta)^2} - 1\right).$$

Wi=1/($\sqrt{((\mathrm{xi} - \mathrm{m})^2} + 1)$)

```
def get_huber_weight(x, mu, gamma):
    subtraction = np.subtract(x, mu)
    return 1 / pow((1 + sum(pow((subtraction / gamma), 2))), (1 / 2))
```

$$\mu = \frac{\sum Xi\ Wi.\mu}{\sum Wi.\mu}\mu$$

```
def get_huber_index(x, mu):
    mu1 = mu
    nu = np.zeros(x.shape[1], int)
    den = 0
    for item in x:
        wi = get_huber_weight(item, mu1, gamma=10)
        nu = nu + np.dot(item, wi)
        den = den + wi
    return nu / den
```

Read file:

```
def read_file(file_name: str):
    img = Image.open(file_name, 'r')
    pix_val = list(img.getdata())
    return img, np.asarray(pix_val, dtype=np.float32)
```

Show output:

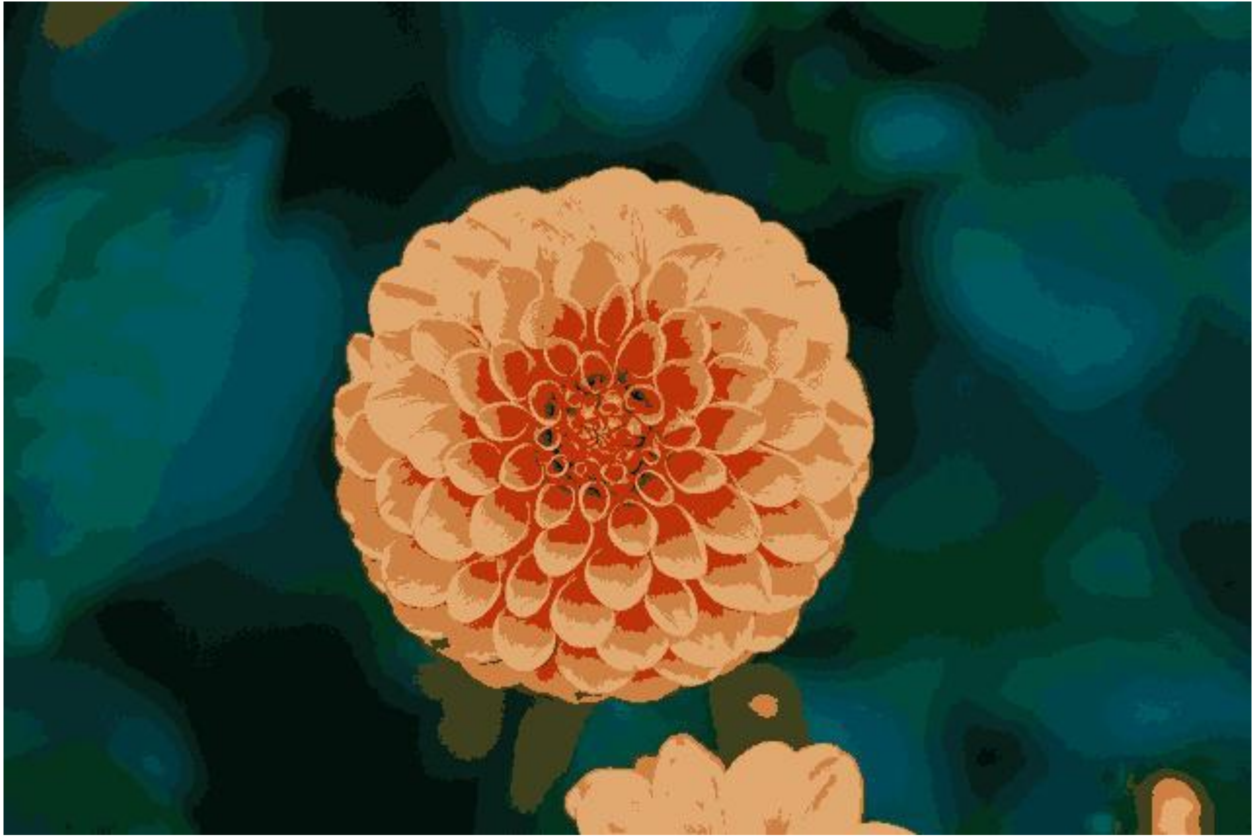
```
def imag_filtering(y, img: Image, identifiers):
    new_image = []
    for x in range(img.size[1]):
        new_image_row = []
        for i in range(img.size[0]):
            new_image_row.append(identifiers[y[x * (img.size[0]) + i]])
        new_image.append(new_image_row)
    new_image = np.asarray(new_image, dtype=np.uint8)
    new_image = Image.fromarray(new_image, 'RGB')
    new_image.save('flower_out2.jpg')
    new_image.show()
```

result

input:

Out put:

1. n_cluster = 15



2.n_cluster = 3