Report

Q1. Interactive Foreground Segmentation Using K-Means

- K-Means Clustering function implementation

1. Define a function with number of k , data and iterations as parameters.
   def k_means_clustering(k, data_points, max_iterations=100):
2. K means focus on finding a random centroid by selecting the k , which is given in this case. So, finding the centroid.
   centroids = data_points[np.random.choice(data_points.shape[0], k, replace=False)]
3. Next step involves using euclidean distance between data points and centroid to find distances.
   distances = np.linalg.norm(data_points[:, np.newaxis] - centroids, axis=2)
4. Assign the data point to the cluster with which it has the closest distance.
   closest-distance = np.argmin(distances, axis=1)
5. Now we have to calculate the new centroid. It requires taking mean.
   new_centroid = np.array([data_points[cs == i].mean(axis=0) for i in range(k)])
6. Returns the centroid and cluster that is final.
   return cs, centroids

- Seed pixel extraction and likelihood computation

- Step 1.
  1. We need a function to extract seeds from images.
     def **extract_seed_pixels**(image, seed_image):
  2. We have to find if the seed is present or not for which we will use where 1.
     If seed is present then find the coordinates of seed. This will be where index is 0
     seeds = np.where(seed_image[:,:,0] == 1)
  3. Now we have to find pixels from the original image as well so we will do it in this step.
     ans = image[seeds[0], seeds[1]] .
     this is making an array here
  4. Return the answer.
- Step 2
  1. We have to compute likelihood.
     def **compute_likelihood**(pixel, centroids, wk=0.1):
  2. For likelihood , we need to calculate the distance between the pixel we got and the centroid we calculated earlier.
     distances = np.linalg.norm(centroids - pixel, axis=1)
  3. Now we will compute likelihood based on the distance we just calculated.
     likelihoods = np.exp(-wk * distances)
     Applying exponential function.
     Wk was given which was 0.1

Wk is small because smaller wk leads to higher likelihood. Inverse relationship between likelihood and distance.

4. Return the answer.

- Step 3:
    1. We have to create a function that assigns pixels to the original image based on the above calculations.
       def assign_pixels(image, fg_likelihoods, bg_likelihoods)
    2. Create a null array whose size should be the same as the size of the image.
       result = np.zeros(image.shape[:2], dtype=np.uint8)
    3. Using a function to iterate through each pixel:
       for i in range(image.shape[0]):
       for j in range(image.shape[1]):
    4. Check if the likelihood of the pixel belonging to the foreground is higher than the likelihood of it belonging to the background.
       if fg_likelihoods[i, j] > bg_likelihoods[i, j]
    5. If the likelihood of the foreground is higher, the corresponding entry in the result array is set to 1, indicating that the pixel is assigned to the foreground.
       result[i, j] = 1
    6. Return result.
- Step 4:
    1. This is the **main function** that will generate the images.
       def **segment_image**(image, fg_seed_image, bg_seed_image, N)
       N is the number of K. It is in an array. Multiple values.
    2. First we have to **extract the seeds**. Both foreground and background.
       Extracts background and front ground seed pixels from the input image.
       fg = extract_seed_pixels(image, fg_img)
       bg = extract_seed_pixels(image, bg_img)
    3. Send the seeds found above **into k means**.
       It gives clusters and centroids in return.
       fg_clusters, fgc = k_means_clustering(N, fg)
    4. **Reshaping the array** to meet the size of original:
       fgl = np.zeros((image.shape[0], image.shape[1], N))
       bgl = np.zeros((image.shape[0], image.shape[1], N))
    5. Now we have to **compute likelihood:**
       for i in range(N):
       fgl[:,:,i] = compute_likelihood(image.reshape(-1, 3), fgc[i]).reshape(image.shape[:2])
       bgl[:,:,i] = compute_likelihood(image.reshape(-1, 3), bgc[i]).reshape(image.shape[:2])
    6. **Sums the likelihoods** across all background clusters for each pixel.
       fglsum = fgl.sum(axis=2) #sums of likelihood for each point

bglsum = bgl.sum(axis=2)

7. Returns the answer.

- Experimentation and comparison with different N values

- Step 5:
    1. Read the images
       test_image = plt.imread("img/lady test.png")
       foreground_seeds = plt.imread("img/lady stroke 1.png")
       background_seeds = plt.imread("img/lady stroke 2.png")
    2. Array for different k values:
       N_values = [12,32,54,64,78 , 85]
    3. Send the images and values of k to function to give answers.
       for N in N_values:
           segmented_image = seg(test_image, foreground_seeds, background_seeds, N)
           plt.imshow(segmented_image)
           plt.title(Segmented Image (N={N})')
           plt.axis('off')
           plt.show()

    Effect of values of k on the images:

    - Too Low K (Under-segmentation):
        - If K is too low, the algorithm may not be able to capture the complexity and diversity of the data adequately.
        - This can lead to ***under-segmentation***, where different regions or clusters in the image are merged into a smaller number of clusters.
        - Under-segmented images may fail to capture fine details or distinct regions in the original image, resulting in loss of information.
    - Too High K (Over-segmentation):
        - Conversely, if K is too high, the algorithm may create too many clusters, leading to over-segmentation.
        - ***Over-segmented*** images may have too many small clusters, resulting in fragmentation of regions that should be considered as a single entity.
        - This can introduce noise into the segmentation and may make it difficult to interpret or analyze the results.
    - Optimal K (Balanced Segmentation):
        - The optimal value of K strikes a balance between under-segmentation and over-segmentation.
        - It captures the underlying structure of the data well, identifying distinct regions or clusters without creating too many or too few clusters.
        - The optimal K may vary depending on the complexity of the image and the characteristics of the data being segmented.

Q2. Face Recognition Using K-NN

- **- Pre-processing and data splitting**
  It includes splitting the data into 2 points. Train and test. I splitted the data into 80 - 20 ratio and 60 - 40 ratio.
  X_train_sub, X_test_sub, y_train_sub, y_test_sub = train_test_split(n_data[indices], [i] * 170, test_size=20, random_state=42)

- **- k-NN classifier implementation**
  Step 1:

  1. In order to not manually compute the labels , I used counters from the built- in library.
  2. I used a class because it looked easier to assign matrices and k.
     class **KNN**Classifier:
       def __init__(self, k, distance_metric='euclidean'):
         self.k = k
         self.distance_metric = distance_metric
  3. Computation of distance is an important step. Same as k means , euclidean distance is used. But if the matrix is cosine, then compute cosine distance. Using a **distance** function for this:
     def _distance(self, x1, x2):.
       if self.distance_metric == 'euclidean':
         return np.linalg.norm(x1 - x2)
       elif self.distance_metric == 'cosine':
         return np.dot(x1, x2) / (np.linalg.norm(x1) * np.linalg.norm(x2))
       else:
         raise ValueError("error")
  4. The main function to find the label is names as labelfinder:
     def **labelfinder**(self, X_train, y_train, x_test)
     It first finds the distance between x train and x test.
     Sort them in ascending order.
     Count the occurrences of each label
     Return the label that occurs the most.

- **- Evaluation with different K values**
  Different values of k are used:
  [2,5,7,11,13,15,17,19,37]

Answers:
80 - 20 ratio dataset:
k = 2 euclidean accuracy = 100.0 %
k = 5 euclidean accuracy = 98.5 %
k = 7 euclidean accuracy = 96.5 %
k = 11 euclidean accuracy = 95.5 %
k = 13 euclidean accuracy = 95.0 %
k = 15 euclidean accuracy = 93.0 %
k = 17 euclidean accuracy = 93.0 %
k = 37 euclidean accuracy = 87.0 %

for 10-7 ratio data:
k = 2 euclidean accuracy = 97.85714285714285 %
k = 5 euclidean accuracy = 95.14285714285714 %
k = 7 euclidean accuracy = 93.0 %
k = 11 euclidean accuracy = 92.0 %
k = 13 euclidean accuracy = 91.57142857142857 %
k = 15 euclidean accuracy = 91.57142857142857 %
k = 17 euclidean accuracy = 90.85714285714286 %
k = 37 euclidean accuracy = 86.0 %

- **- Evaluation with fewer training images**
  Mentioned above already.
- **- Comparison with SVM and GaussianNB**
  SVM Accuracy 100.0
  GaussianNB Accuracy 85.0

  SVM Accuracy 98.14285714285714
  GaussianNB Accuracy 85.0

- **- Dimensionality reduction and visualization**
  For a 3D visualization, 3 sides are required. X , y and z. I.e principle components.
  pca = **PCA**(n_components=3)
  Creates a PCA object with three principal components.
  X_trainpca = pca.fit_transform(X_trainn)
  Fit the PCA model to the training data X_trainn and transform it into a new feature space defined by the three principal components.
  X_testpca = pca.transform(X_testt)
  Transforms the test data X_testt using the same PCA model fitted on the training data. This ensures consistency in dimensionality reduction between the training and test data
  Then plot them.