Object Recognition Report

Sinuo Liu

October 23, 2024

1 K-means Clustering

1.1 Implementation

The goal is to implement a K-means clustering algorithm to perform image segmentation and analyze its performance with different numbers of clusters.

Firstly, we initialize the centroids, where we randomly selecte K initial points from the pixel values of the image. To assign each pixel to the nearest centroid, a function is implemented to calculate the Euclidean distance between each pixel and the centroids.

```
def euclidean_distance(pixel_values, centroids):
    return np.linalg.norm(pixel_values - centroids, axis=1)
```

The core of the algorithm is the iterative update process. The classification of pixels is based on their distance from the centroids, with the pixels closest to the centroid forming the initial cluster. This initial cluster is then used to calculate the mean of the pixels within it. As the iteration progresses, each cluster is identified as a separate class and the mean of that class is recalculated as the new centroid using `np.mean` until no further change is observed (`np.linalg.norm(centroids - new_centroids) < 1e-5`) or the maximum number of iterations, indicating that the iteration is complete.

To observe the results of the image clustering process, a for-loop is set to K = 2, 3, 5, 7, and 10, respectively. Additionally, the "running time" and number of iterations are set in order to ascertain the computational cost.

1.2 Results and Observations

The results obtained for varying values of K are shown below:

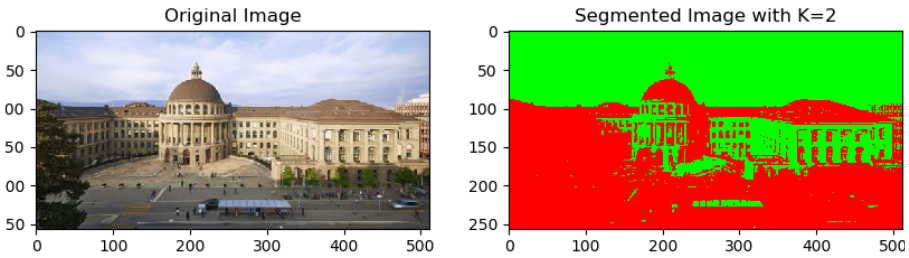

Figure 1: Segmented Image with K = 2

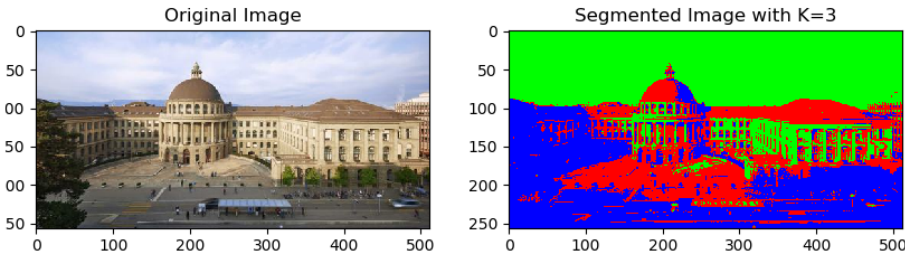

Figure 2: Segmented Image with K = 3

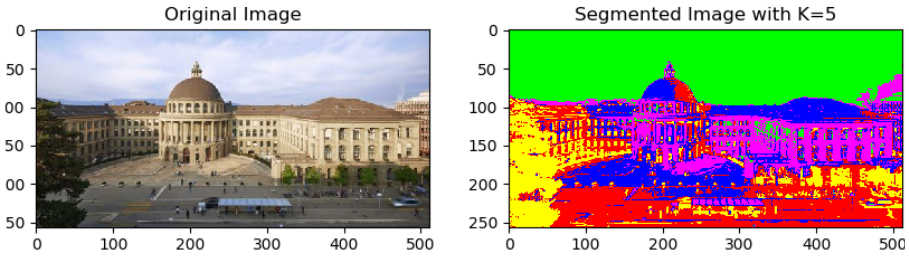

Figure 3: Segmented Image with K = 5

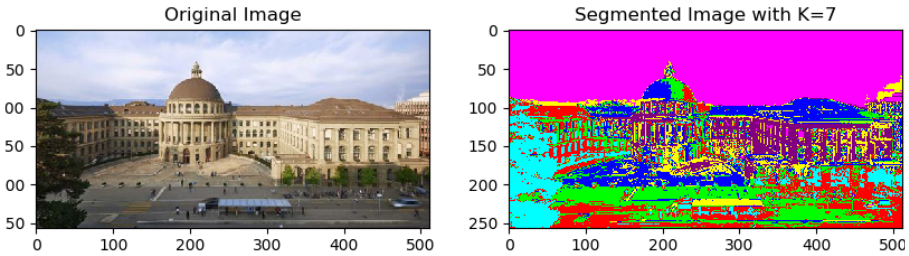

Figure 4: Segmented Image with K = 7

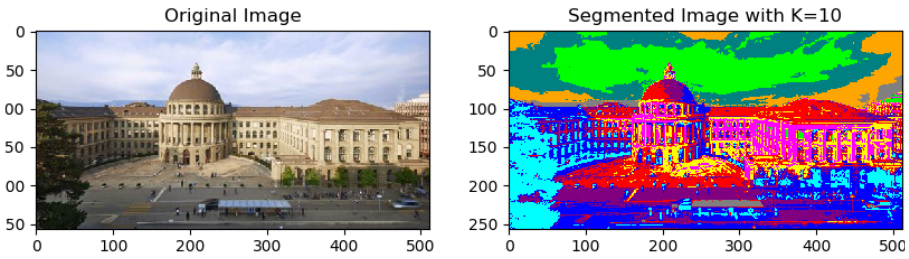

Figure 5: Segmented Image with K = 10

The results indicate that the classification is more accurate when K equals 5. When K = 5, we can see that different regions of the image, such as the building, sky, ground, and tree, are represented by distinct clusters. As K is small, the image is divided into large, broad regions with less detail. While K is increasing, with more clusters, the segmentation becomes finer, capturing more details. However, we also start to notice a bit of over-segmentation, where smaller areas are broken down into even more clusters, potentially making the image look noisier. For example, when K = 10, the categorisation is very detailed, e.g. clouds are included, but there is still a lot of noise.

It is also noticeable that the classification performance is very poor when K = 7. The possible reason for this is that the performance of K-means is strongly dependent on the choice of the initial centroids, and it is easy to fall into local optima, especially when K is large.

Next, we analyze the duration of each K-value classification run and the total number of iterations performed. The result is shown as:

```
K = 2, Time taken: 16.0321 seconds, Iterations: 11
K = 3, Time taken: 44.2067 seconds, Iterations: 31
K = 5, Time taken: 33.4500 seconds, Iterations: 23
K = 7, Time taken: 69.0328 seconds, Iterations: 48
K = 10, Time taken: 78.6107 seconds, Iterations: 49
```

The average complexity of the K-Means algorithm is $O(knT)$, where k is the number of clusters to be input, n is the sample size in the entire dataset, and T is the number of iterations required. With increasing K, the algorithm has to manage more centroids, which generally increases the computational cost. However, as seen with K = 5, more centroids can sometimes lead to faster convergence if the data points quickly align to their closest cluster centers. Moreover, the time taken for convergence is heavily influenced by how well the initial centroids are chosen. Poor initial centroid placement can lead to more iterations before the clusters stabilize, which seems to be the case for K = 3 where the number of iterations is higher.

Generally, a higher number of clusters implies a higher computational cost because each pixel has to be compared to more centroids. However, if the centroids reach stability sooner, the overall number of iterations might be reduced, balancing out the cost.

2 Bag-of-Words Classification

2.1 Implementation

2.1.1 Local Feature Extraction

In the first step, a simple feature point detector is implemented that generates points on a grid, leaving a border around the image with 8 pixels.

```
stepX = (w - 2 * border) / (nPointsX - 1)
stepY = (h - 2 * border) / (nPointsY - 1)
for i in range(nPointsY):
    for j in range(nPointsX):
        x = border + j * stepX
        y = border + i * stepY
        vPoints.append([x, y])
```

Finally 100 grid points should be returned. Each grid points will be described using the Histogram of Oriented Gradients (HOG) descriptor. The descriptor is based on a patch consisting of a 4×4 grid of cells surrounding the grid point. Within each cell, an 8-bin histogram is computed by analyzing

the orientations of the gradients of the pixel intensities. The `magnitude` is computed as the Euclidean norm of the gradient vectors, while the `angle` is determined using the `arctan2` function, which provides the direction of the gradient. The angles are then converted from radians to degrees and adjusted to fall within the range of [0, 180].

```
magnitude = np.sqrt(grad_x ** 2 + grad_y ** 2)
angle = np.arctan2(grad_y, grad_x) * (180 / np.pi)
angle[angle < 0] += 180
```

However, the magnitude of the gradient can vary significantly across different parts of an image. If certain features have much higher magnitudes, they might dominate the histogram, which can reduce performance. So I decide to normalize the gradient magnitudes in each cell before accumulating them in the histogram.

```
cell_magnitude = cell_magnitude / np.linalg.norm(cell_magnitude)
```

For each cell, a histogram `hist` is initialized to hold the magnitude contributions for each angle bin. The histograms from all 16 cells are then concatenated, resulting in a 128-dimensional feature vector for the given grid point. The dimension of the `descriptor` should be `[100,128]`.

2.1.2 Bag-of-words Vector Encoding

After constructing the codebook, the function `bow_histogram` computes a bag-of-words (BoW) histogram corresponding to each given image. For each feature vector, the code calculates the Euclidean distance from the feature to each centroid using `np.linalg.norm`. Once the distances are computed, the index of the closest centroid is identified, and the corresponding bin in the histogram is incremented accordingly. `histo` stores the count of features assigned to each centroid. After calculating these counts, the histogram is normalized to ensure that the values sum to one, facilitating comparison across different images.

The `create_bow_histograms` function generates BoW histograms from images for training images and test images. For each image, it generates feature points and extracts HOG features using `descriptors_hog` function. It calculates the BoW histogram by determining the nearest centroid for each feature vector. The resulting histogram is appended to the `vBoW` list with dimensions $[n\_imgs, k]$.

To classify a new test image, we calculate its BoW histogram and assign it to the category of the nearest neighbor from the training histograms by finding the smallest Euclidean distance.

```
DistPos = np.min(np.linalg.norm(vBoWPos - histogram, axis = 1))
DistNeg = np.min(np.linalg.norm(vBoWNeg - histogram, axis = 1))
```

2.2 Results and Observations

The maximum number of iterations is set to 50, and the following accuracy scores are obtained for different K values:

```
K = 4, test pos accuracy: 0.878, test neg accuracy: 0.720
K = 5, test pos accuracy: 1.000, test neg accuracy: 0.900
K = 6, test pos accuracy: 0.959, test neg accuracy: 0.960
K = 7, test pos accuracy: 1.000, test neg accuracy: 0.840
K = 8, test pos accuracy: 0.898, test neg accuracy: 0.960
K = 9, test pos accuracy: 0.898, test neg accuracy: 0.960
K = 10, test pos accuracy: 0.918, test neg accuracy: 0.960
K = 15, test pos accuracy: 0.878, test neg accuracy: 0.940
K = 20, test pos accuracy: 0.939, test neg accuracy: 0.880
```

From the results, the overall results are performing well and it can be seen that the performance is better when K = 6. When K = 6, the accuracy of detecting positive test samples is 0.959 and the accuracy of detecting negative test samples is 0.960. The accuracies of both positive test and negative test are comparable and both perform very well.

When the K value is large, it is very likely to overclassify when categorizing features and the running time for clustering increases.

3 Conclusion

In the K-means clustering task, when K = 5 is classified quite well and the performance is balanced with the computational cost. In the Bag-of-words task, the score of classification on the test image is better when K = 6, with approximately 0.96 accuracy score for both the positive test and the negative test.