exercise_12_clustering

February 2, 2020

0.1 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook. 2. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)). 3. Concatenate your solutions for other tasks with the output of Step 2. On linux, you can use `pdfunite`, there are similar tools for other platforms, too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` version 5.5 or later by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

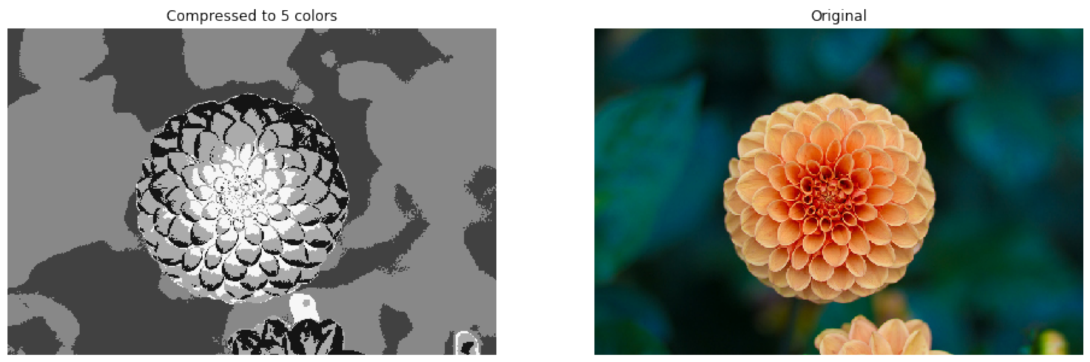
```
In [2]: import matplotlib.pyplot as plt
        import numpy as np
        from PIL import Image
        from sklearn.datasets import load_sample_image

        %matplotlib inline

        def compare_images(img, img_compressed, k):
            """Show the compressed and uncompressed image side by side.
            """

            fig, axes = plt.subplots(1, 2, figsize=(16, 12))
            axes[0].set_axis_off()
            if isinstance(k, str):
                axes[0].set_title(k)
            else:
                axes[0].set_title(f"Compressed to {k} colors")
            axes[0].imshow(img_compressed)
            axes[1].set_axis_off()
            axes[1].set_title("Original")
            axes[1].imshow(img)
```

1 K-Means

In this first section you will implement the image compression algorithm from Bishop, chapter 9.1.1. Take an RGB image $X \in \mathbb{R}^{h \times w \times 3}$ and interpret it as a data matrix $X \in \mathbb{R}^{N \times 3}$. Now apply *k*-means clustering to find *k* colors that describe the image well and replace each pixel with its associated cluster.

```
In [4]: # Alternatively try china.jpg
        X = load_sample_image("flower.jpg")

        # or load your own image
        # X = np.array(Image.open("/home/user/path/to/some.jpg"))

In [3]: def kmeans(X, k):
            """Compute a k-means clustering for the data X.

            Parameters
            ----------
            X : np.array of size N x D
                where N is the number of samples and D is the data dimensionality
            k : int
                Number of clusters

            Returns
            -------
            mu : np.array of size k x D
                Cluster centers
            z : np.array of size N
                Cluster indicators, i.e. a number in 0..k - 1, for each data point in X
            """

            # TODO: Compute mu and z

            N, D = X.shape

            # 1. Initialize the centroids at random
            #mu = X.max() * np.random.random((k, D)) + X.min()
            mu = np.array([ X[np.random.randint(0, N)] for _ in range(k) ])
            z = np.zeros(N, dtype=np.int8)

            # 1. Initialize the centroids with K-means++
            mu = np.empty((k, D))
            # 1.1 Choose random centroid mu_1
            mu[0] = X[np.random.randint(N)]
            # 1.2 For each x_i compute Distance D_i^2
            Di2 = np.linalg.norm(X - mu[0], axis=1) ** 2
            for i in range(1, k):
                # 1.3 Sample the next centrouid mu_i from {x_i} with probability proportional
                potential = Di2.sum()
                sample = np.random.random_sample(1) * potential # only 1 but could be multiple
                candidate = np.searchsorted(np.cumsum(Di2), sample)
                mu[i] = X[candidate]
                # 1.4 Recompute distance Di2
                Di2 = np.minimum(Di2, np.linalg.norm(X - mu[i], axis=1) ** 2)
            # 1.5 Continue Steps 3 and 4 und all k have been chosen

            # 2. Update cluster indicators
            for i, x in enumerate(X):
                cluster = np.argmin(np.linalg.norm(x - mu, axis=1) ** 2)
                z[i] = int(cluster)

            # 3. Update centroids
            for i in range(k):
                mask = z == i
                N_k = np.sum(mask)
                mu[i] = (1 / N_k) * np.sum(X[mask])

            # 4. Has objective converged
            if np.isnan(mu).any():
                print("Contained empty clouds")
                print(mu)
                #return kmeans(X, k)

            return mu, z
```

```
In [4]: # Cluster the color values
        k = 5
        mu, z = kmeans(X.reshape((-1, 3)), k)

        # Replace each pixel with its cluster color
        X_compressed = mu[z].reshape(X.shape).astype(np.uint8)

        # Show the images side by side
        compare_images(X, X_compressed, k)
```



2 Gaussian Mixture Models & EM

Now you will repeat the same exercise with GMMs.

```
In [1]: def gmm_log_probability(X, pi, mu, sigma):
            """Compute the joint log-probabilities for each data point and component.

            Parameters
            ----------
            X : np.array of size N x D
                where N is the number of samples and D is the data dimensionality
            pi : np.array of size k
                Prior weight of each component
            mu : np.array of size k x D
                Mean vectors of the k Gaussian component distributions
            sigma : np.array of size k x D x D
                Covariance matrices of the k Gaussian component distributions

            Returns
            -------
            P : np.array of shape N x k
                P[i, j] is the joint log-probability of data point i under component j
            """

            # TODO: Compute P

            # log p(X, Z | , ţ, )
            N, D = X.shape
            k = pi.shape[0]


            #s1 = 1 / (np.sqrt( 2 * np.pi * sigma))
            #s2 = np.exp(-(np.square(X - mu) / (2 * sigma)))
            #P = pi * s1 * s2
            #print(P.shape)

            from scipy.stats import multivariate_normal
            P = np.empty((N, k))
            for i in range(k):
                P[i] = np.log(pi[i] * multivariate_normal.pdf(X, mu[i], sigma[i]))

            return P


        def em(X, k, tol=0.001):
            """Fit a Gaussian mixture model with k components to X.

            Parameters
```

```
----------
X : np.array of size N x D
    where N is the number of samples and D is the data dimensionality
k : int
    Number of clusters
tol : float
    Converge when the increase in the mean of the expected joint log-likelihood
    is lower than this

The algorithm should stop when the relative improvement in the optimization
objective is less than rtol.

Returns
-------
pi : np.array of size k
    Prior weight of each component
mu : np.array of size k x D
    Mean vectors of the k Gaussian component distributions
sigma : np.array of size k x D x D
    Covariance matrices of the k Gaussian component distributions
"""

# https://github.com/scikit-learn/scikit-learn/blob/a95203b/sklearn/mixture/gmm.py

# TODO: Compute pi, mu, sigma
N, D = X.shape

# 1. Initialize model parameters
pi = np.ones(k)
mu = np.random.random((k, D))
sigma = np.array([np.identity(D) for i in range(k)])

from scipy.stats import multivariate_normal

num_iters = 0
converged = False
#while not(converged):
while num_iters < 5:
    # 2. E step. Evaluate the responsibilities
    gamma = np.zeros((N, k), dtype=float)
    for i in range(k):
        gamma[:,i] = pi[i] * multivariate_normal.pdf(X, mu[i,:], sigma[i])
    gamma_norm = np.sum(gamma, axis=1)[:,np.newaxis]
    gamma /= gamma_norm

    # 3. M step. Re-estimate the parameters
    pi = np.mean(gamma, axis=0)
    mu = np.dot(gamma.T, X) / np.sum(gamma, axis=0)[:,np.newaxis]
```

```
            for i in range(k):
                x = X - mu[i,:]
                x_mu = np.matrix(x)
                gamma_diag = np.diag(gamma[:,i])
                gamma_diag = np.matrix(gamma_diag)
                sigma_i = x.T * gamma_diag * x
                sigma[i,:,:] = (sigma_i) / np.sum(gamma, axis=0)[:,np.newaxis]
                #sigma[i] = (1 / N_k) * np.sum(gamma[i] * np.square(X - mu[i]).T)

            print(gamma)
            print(mu)
            print(sigma)
            print(pi)
            #likelihood = gmm_log_probability(X, pi, mu, sigma)
            #if likelihood <= tol:
            #    converged = True

            print(likelihoods.shape)
            print(np.mean(likelihoods))
            # test
            converged = True

            num_iters += 1

        return pi, mu, sigma
```

```
In [ ]: # Fit the GMM
        k = 5
        pi, mu, sigma = em(X.reshape((-1, 3)), k)

        # Determine the most likely cluster of each pixel
        log_p = gmm_log_probability(X.reshape((-1, 3)), pi, mu, sigma)
        z = log_p.argmax(axis=1)

        # Replace each pixel with its cluster mean
        X_compressed = mu[z].reshape(X.shape).astype(np.uint8)

        # Show the images side by side
        compare_images(X, X_compressed, k)
```

```
/Users/lilith/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:88: RuntimeWarning:
```

3 Sampling Unseen Datapoints

You have trained a generative model which allows you to sample from the learned distribution. In this section, you sample new images.

```
In [ ]: def gmm_sample(N, pi, mu, sigma):
            """Sample N data points from a Gaussian mixture model.

            Parameters
            ----------
            N : int
                Number of data points to sample
            pi : np.array of size k
                Prior weight of each component
            mu : np.array of size k x D
                Mean vectors of the k Gaussian component distributions
            sigma : np.array of size k x D x D
                Covariance matrices of the k Gaussian component distributions

            Returns
            -------
            X : np.array of shape N x D
            """

            # TODO: Sample X

            return X
```

```
In [ ]: # Sample pixels and reshape them into the size of the original image
        X_sampled = gmm_sample(np.prod(X.shape[:-1]), pi, mu, sigma).reshape(X.shape).astype(np

        # Compare the original and the sampled image
        compare_images(X, X_sampled, "Sampled")
```

Explain what you see in the generated images. (1-3 sentences)