Spectral Clustering - Esercitazione 5/11/2021

November 4, 2021

The objective of today's lab session is to implement a n-cluster clustering procedure based on Spectral Clustering[1]. To do so we must:

1. model the provided dataset as a graph and compute its affinity matrix A

$$A_{i,j} = e^{-\frac{\sum_{k=1}^{d} ||x_i^k - x_j^k||^2}{\sigma^2}}; \tag{1}$$

2. compute the graph's degree matrix D, *i.e.* a diagonal matrix such that

$$D_{i,i} = \sum_j A_{i,j}; \tag{2}$$

3. comput the Laplacian matrix for the graph $L = D - A$;

4. compute L's eigenvectors $\vec{v}_i$, sorted by increasing corresponding eigenvalues λ_i crescenti;

5. **Fiedler-Vector solution:** for a binary clusterization, we may simply consider the eigenvector corresponding to the 2nd smallest eigenvalue $\vec{v}_2$ (i.e. the Fiedler Vector) and use the positivity of its components as labels;

6. **K-Means solution:** in a more general case where you need n **distinct clusters**, you may consider the eigenvectors corresponding to the n smallest eigenvalues $\vec{v}_1, ...\vec{v}_n$ and use their components as the representation of old data in a new embedding space. After that, we may apply K-Means to this new data representation.

Let us translate this procedure in numpy code step by step.

1 Implementation

The following is the code we provide for the `spectral_clustering` function, with comments that indicate how to implement the corresponding steps:

[1]Yep, that's a lot of "cluster"s right there

```python
def spectral_clustering(data, n_cl, sigma=1., fiedler_solution=False):
    affinity_matrix = ... # 1
    degree_matrix = ... # 2
    laplacian_matrix = ... # 3

    eigenvalues, eigenvectors = ... # 4
    if eigenvalues.dtype == 'complex128':
        print("My dude, you got complex eigenvalues. Now I am not gonna break "
            "down, but you should totally give me higher sigmas. (;")
        eigenvalues, eigenvectors = eigenvalues.real, eigenvectors.real
    eigenvalues, eigenvectors = ... # 4

    # SOLUTION A: Fiedler-vector solution
    labels = ... # 5
    if fiedler_solution:
        return labels

    # SOLUTION B: K-Means solution
    new_features = ... # 6
    labels = ... # 6

    return labels
```

The function takes the following items as input:

- `data`, an `np.array` with shape `n_samp, n_dim`;
- `n_cl`, an integer corresponding to the number of clusters n;
- `sigma`, the hyper-parameter σ, which is at the denominator of Eq. 1 and that **must be properly tuned** (see Sec. 2).
- `fiedler_solution`, a flag (false by default), which establishes whether to return the result of the Fielder-Vector or K-Means solution.

1.1 Affinity Matrix

We may rewrite Eq. 1 as follows

$$A_{i,j} = e^{-\frac{M_{i,j}}{\sigma^2}}, \quad \text{where} \quad M_{i,j} = \sum_{k=1}^{d} ||x_i^k - x_j^k||^2; \tag{3}$$

this splits the computation into two steps. Firstly, we will comput the distance matrix M.

M is obtained as the Euclidean distance (the square root is not necessary) of each point in the dataset from every other dataset point. To translate this comparison in `numpy` we exploit **broadcasting** as follows:

```
dist_matrix = ((np.expand_dims(data, 0) -
                    np.expand_dims(data, 1)) ** 2).sum(2)
```

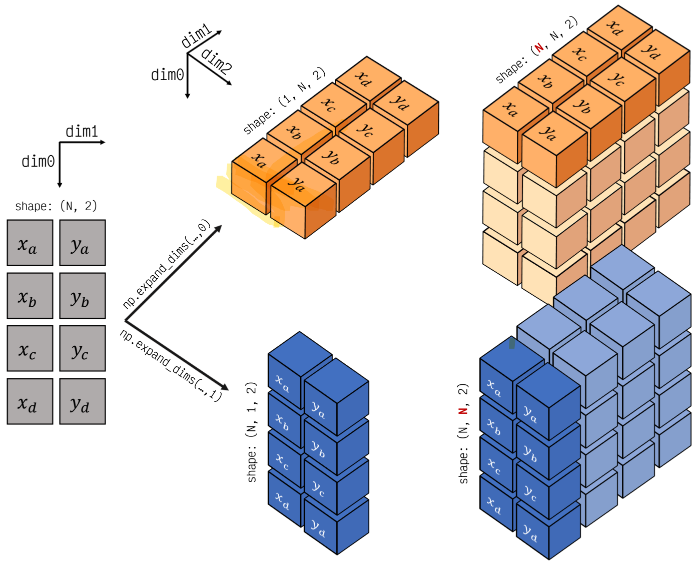

Figure 1: N corresponds to `n_samp` and 2 to `n_dim`.

The `data` `np.array` is first expanded twice with two distinct fictitious dimensions. We get two `np.arrays` with shape `1, n_samp, n_dim` and `n_samp, 1, n_dim` respectively (see Fig. 1). The shapes of these array allow for broadcasting. By writing down a subtraction between them, both get repeated to obtain shape `n_samp, n_samp, n_dim` as shown in Fig. 1.

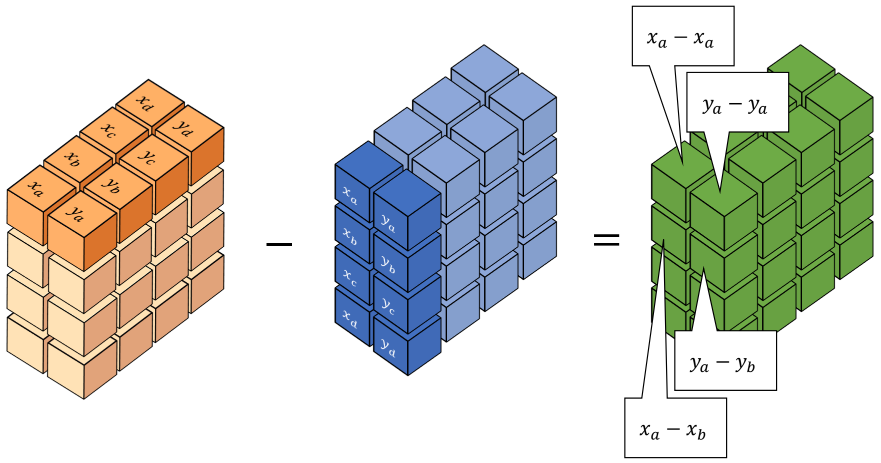

Figure 2: Subtraction between expanded `np.arrays`.

The difference between the two `arrays` is `(n_samp, n_samp, n_dim)`-shaped and contains in each of its cells the difference between two coordinates of two datapoints in the initial dataset as shown in Fig. 2.

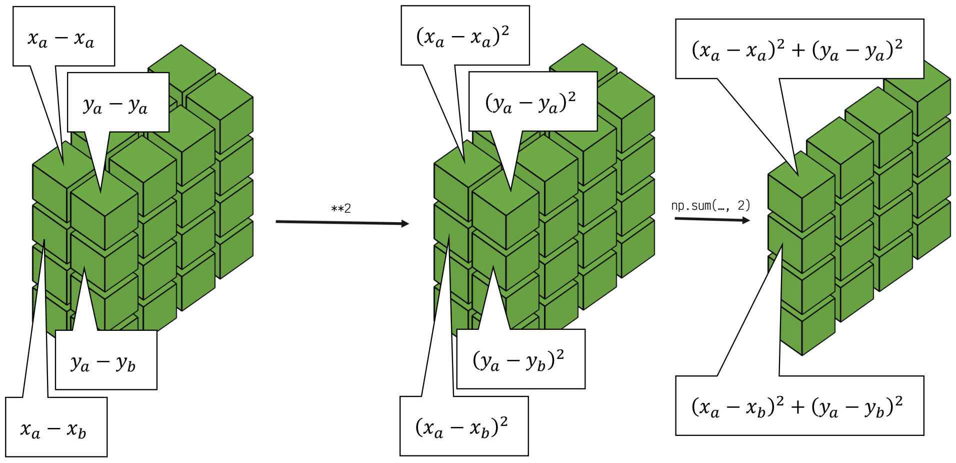

Figure 3: Reduction of the difference between `arrays`.

To get M, the differences between coordinats must be squared and summed for corresponding points. In Fig. 3, we graphically show how the content of `arrays` manipulated by these transformations is changing.

Once we have M, we can easily obtain the affinity matrix A by translating Eq. 3 in `numpy` code:

```python
affinity_matrix = np.exp(-dist_matrix / (sigma ** 2))
```

1.2 Degree Matrix

According to Eq. 2, D is a diagonal matrix whose elements are given by the column-wise sums of the values of A[2]. A diagonal matrix can be easily initialized in `numpy` with `np.diag(...)`.

```python
degree_matrix = np.diag(affinity_matrix.sum(1))
```

1.3 Laplacian Matrix

It is trivial to translate the formula that computes the Laplacian matrix L from D and A:

```python
laplacian_matrix = degree_matrix - affinity_matrix
```

1.4 Eigenvalues & Eigenvectors

The calculation of eigenvalues and eigenvectors of L is implemented in `numpy` with function `np.linalg.eig(...)`. By carefully reading the documentation for this function, we learn that:

- the function returns `w, v`, where `w` is the array containing the eigenvalues and `v` is the `np.array` containing the eigenvectors;
- `w` and `v` are **not ordered**;
- `"v[:,i] is the eigenvector corresponding to the eigenvalue w[i]."` (Fig. 4).

Figure 4: "It's a surprise tool that will help us later."

[2]row-wise also works b.t.w., since A is symmetrical

```
eigenvalues, eigenvectors = np.linalg.eig(laplacian_matrix)
```

Notice that the function might return complex values (in this exercise, if σ is low), hence the provided trace already includes additional code to convert the results to real values and output a warning.

Now, we must order the eigenvalues and eigenvectors according to the criterion required by the exercise: by **increasing values of the corresponding eigenvalues**. To sort the eigenvalues, we might simply call `np.sort(...)`. However, since we must apply the same ordering to eigenvectors too, we must explicitly extract the indices that allow their ordering. To do this, we use `np.argsort(...)`, which returns the indices needed for sorting an `np.array` in an increasing manner: this is just what we need!

```
sorted_indices = np.argsort(eigenvalues)
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

Please not that, due to the implementative choice made by `numpy` authors which is explicitly stated in the docs, the eigenvectors must be indexed on their second axis (Fig. 4).

1.5 Fiedler-Vector Solution

With our eigenvectors ordered, we can easily produce the Fiedler-Vector solution, which requires returning the sign of the components of the second eigenvector $\vec{v}_2$ as labels. The provided code for the solution works well even with boolean-typed labels, which means that we can simply select the second eigenvector (*i.e.,* use index `1` on the second axis for the reasons mentioned in Sec. 1.4) and test its element-wise positivity with a simple `> 0` comparison:

```
labels = eigenvectors[:, 1] > 0
if fiedler_solution:
    return labels
```

Note that this solution only works if the number of clusters to be recognized n is exactly 2. For this reason, the provided solution includes an additional security check that verifies this condition and possibly raises an exception at the start of the `spectral_clustering` function:

```
if fiedler_solution and n_cl != 2:
    raise Exception("Cannot apply Fiedler to more than 2 clusters!")
```

1.6 K-Means Solution

A solution based on K-Means requires that we select the components of eigenvectors from $\vec{v}_1$ to $\vec{v}_n$ (bounds included) as a new representation for our data.

To do this, we must again select on the second axis (again, as anticipated in Sec. 1.4):

```
new_features = eigenvectors[:, 1:n_cl+1]
```

`new_features`, with shape `n_dim, {n_cl}`, constitutes a suitable representation for the data-points in an alternative embedding space. On this new representation, we can perform the previously seen K-Means clustering. The code we provide already imports class `KMeans` from the well-known `sklearn` library. By checking out its documentation, we see that we must specify the number of clusters to be learned when instantiating this class. We also see that it exposes the following methods (which are a standard API for all `sklearn` models):

- `fit`, which takes a dataset `X` as input and "trains" the model;
- `predict`, which takes a batch of data-points `X` as input and returns the model's prediction (this should be used only after `fit`).

Additionally, we also have the `fit_predict` method, combining the previous two and directly returning the model's prediction on the same data used for fitting. We use this to directly obtain the result of `KMeans` clustering with the following code line:

```
labels = KMeans(n_cl).fit_predict(new_features)
```

2 Tuning

To complete the session, we must come up with suitable values for the hyperparameters `n_cl` and `sigma` for both provided dataset (see function `main_spectral_clustering`). The values for `n_cl` are obviously 2 for the `two_moon_dataset` and 3 for the `gaussians_dataset`, since they contain two and three clusters of points respectively.

Regarding `sigma`, we must proceed by trial-and-error until a value that allows for a correct discrimination of the clusters is found. For instance, we show in Fig. 5 that clusterization fails by either choosing values that are too high or too low. A good result is obtained with values ~ 0.1 for the `two_moon_dataset` and ~ 2 for the `gaussians_dataset`.

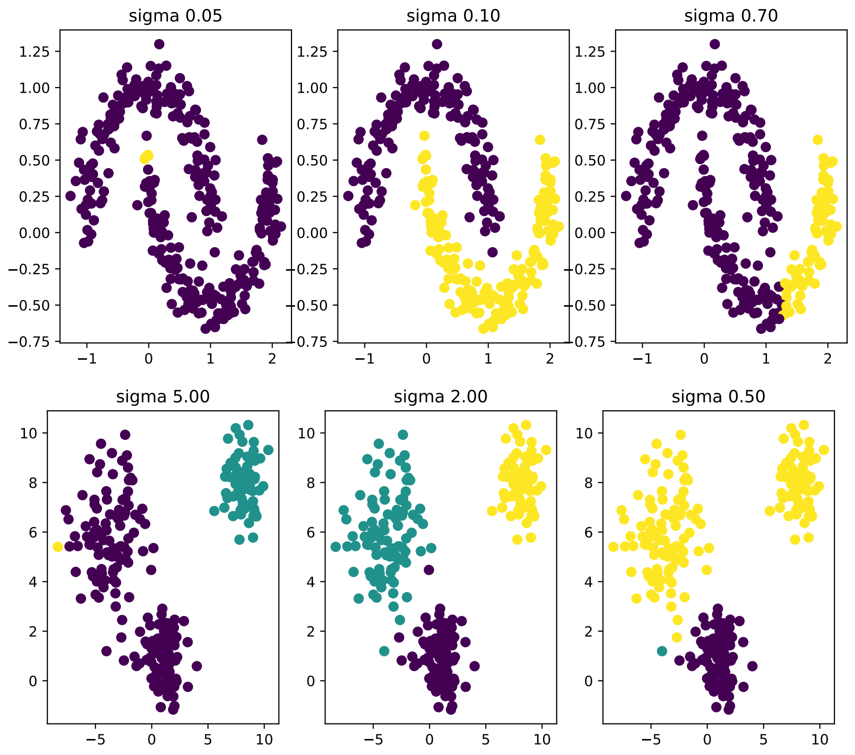

Figure 5: Results produced by `spectral_clustering` for varying `sigma`.