SpecClustering_professor

September 19, 2019

1 Spectral Clustering Algorithms

```
Notebook version: 1.1 (Nov 17, 2017)

Author: Jesús Cid Sueiro (jcid@tsc.uc3m.es)
        Jerónimo Arenas García (jarenas@tsc.uc3m.es)

Changes: v.1.0 - First complete version.
         v.1.1 - Python 3 version
```

[ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# use seaborn plotting defaults
import seaborn as sns; sns.set()

from sklearn.cluster import KMeans
from sklearn.datasets.samples_generator import make_blobs, make_circles
from sklearn.utils import shuffle
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.cluster import SpectralClustering

# For the graph representation
import networkx as nx
```

1.1 1. Introduction

The key idea of spectral clustering algorithms is to search for groups of connected data. I.e, rather than pursuing compact clusters, spectral clustering allows for arbitrary shape clusters.

This can be illustrated with two artifitial datasets that we will use along this notebook.

1.1.1 1.1. Gaussian clusters:

The first one consists of 4 compact clusters generated from a Gaussian distribution. This is the kind of dataset that are best suited to centroid-based clustering algorithms like K-means. If the

goal of the clustering algorithm is to minimize the intra-cluster distances and find a representative prototype or centroid for each cluster, *K*-means may be a good option.

[ ]:
```
N = 300
nc = 4
Xs, ys = make_blobs(n_samples=N, centers=nc,
                    random_state=6, cluster_std=0.60, shuffle = False)
X, y = shuffle(Xs, ys, random_state=0)

plt.scatter(X[:, 0], X[:, 1], s=30);
plt.axis('equal')
plt.show()
```

Note that we have computed two data matrices:

- **X**, which contains the data points in an arbitray ordering
- $\mathbf{X}_s$, where samples are ordered by clusters, according to the cluster id array, **y**.

Note that both matrices contain the same data (rows) but in different order. The sorted matrix will be useful later for illustration purposes, but keep in mind that, in a real clustering application, vector **y** is unknown (learning is not supervised), and only a data matrix with an arbitrary ordering (like **X**) will be available.

1.1.2 1.2. Concentric rings

The second dataset contains two concentric rings. One could expect from a clustering algorithm to identify two different clusters, one per each ring of points. If this is the case, *K*-means or any other algorithm focused on minimizing distances to some cluster centroids is not a good choice.

[ ]:
```
X2s, y2s = make_circles(n_samples=N, factor=.5, noise=.05, shuffle=False)
X2, y2 = shuffle(X2s, y2s, random_state=0)
plt.scatter(X2[:, 0], X2[:, 1], s=30)
plt.axis('equal')
plt.show()
```

Note, again, that we have computed both the sorted ($\mathbf{X}_{2s}$) and the shuffled ($\mathbf{X}_2$) versions of the dataset in the code above.

Exercise 1: Using the code of the previous notebook, run the *K*-means algorithm with 4 centroids for the two datasets. In the light of your results, why do you think *K*-means does not work well for the second dataset?

[ ]:
```
# <SOL>
est = KMeans(n_clusters=4)
clusters = est.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=clusters, s=30, cmap='rainbow')
plt.axis('equal')

clusters = est.fit_predict(X2)
plt.figure()
plt.scatter(X2[:, 0], X2[:, 1], c=clusters, s=30, cmap='rainbow')
plt.axis('equal')
```

```
plt.show()
# </SOL>
```

Spectral clustering algorithms are focused on connectivity: clusters are determined by maximizing some measure of intra-cluster connectivity and maximizing some form of inter-cluster connectivity.

1.2 2. The affinity matrix

1.2.1 2.1. Similarity function

To implement a spectral clustering algorithm we must specify a similarity measure between data points. In this session, we will use the *rbf* kernel, that computes the similarity between $\mathbf{x}$ and $\mathbf{y}$ as:

$$\kappa(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2)$$

Other similarity functions can be used, like the kernel functions implemented in Scikit-learn (see the metrics module).

1.2.2 2.2. Affinity matrix

For a dataset $\mathcal{S} = \{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(N-1)}\}$, the $N \times N$ **affinity matrix K** contains the similarity measure between each pair of samples. Thus, its components are

$$K_{ij} = \kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$$

The following fragment of code illustrates all pairs of distances between any two points in the dataset.

[ ]:
```
gamma = 0.5
K = rbf_kernel(X, X, gamma=gamma)
```

1.2.3 2.3. Visualization

We can visualize the affinity matrix as an image, by translating component values into pixel colors or intensities.

[ ]:
```
plt.imshow(K, cmap='hot')
plt.colorbar()
plt.title('RBF Affinity Matrix for gamma = ' + str(gamma))
plt.grid('off')
plt.show()
```

Despite the apparent randomness of the affinity matrix, it contains some hidden structure, that we can uncover by visualizing the affinity matrix computed with the sorted data matrix, $\mathbf{X}_s$.

[ ]:
```
Ks = rbf_kernel(Xs, Xs, gamma=gamma)

plt.imshow(Ks, cmap='hot')
plt.colorbar()
plt.title('RBF Affinity Matrix for gamma = ' + str(gamma))
plt.grid('off')
plt.show()
```

Note that, despite their completely different appearance, both affinity matrices contain the same values, but with a different order of rows and columns.

For this dataset, the sorted affinity matrix is almost block diagonal. Note, also, that the block-wise form of this matrix depends on parameter γ.

Exercise 2: Modify the selection of γ, and check the effect of this in the appearance of the *sorted* similarity matrix. Write down the values for which you consider that the structure of the matrix better resembles the number of clusters in the datasets.

Out from the diagonal block, similarities are close to zero. We can enforze a block diagonal structure be setting to zero the small similarity values.

For instance, by thresholding $\mathbf{K}_s$ with threshold t, we get the truncated (and sorted) affinity matrix

$$\overline{K}_{s,ij} = K_{s,ij} \cdot \mathrm{u}(K_{s,ij} - t)$$

(where $\mathrm{u}()$ is the step function) which is block diagonal.

Exercise 3: Compute the truncated and sorted affinity matrix with $t = 0.001$

[ ]:
```
t = 0.001
# Kt = <FILL IN> # Truncated affinity matrix
Kt = K*(K>t)     # Truncated affinity matrix
# Kst = <FILL IN> # Truncated and sorted affinity matrix
Kst = Ks*(Ks>t)  # Truncated and sorted affinity matrix
```

1.3 3. Affinity matrix and data graph

Any similarity matrix defines a weighted graph in such a way that the weight of the edge linking $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ is K_{ij}.

If K is a full matrix, the graph is fully connected (there is and edge connecting every pair of nodes). But we can get a more interesting sparse graph by setting to zero the edges with a small weights.

For instance, let us visualize the graph for the truncated affinity matrix $\overline{\mathbf{K}}$ with threshold t. You can also check the effect of increasing or decreasing t.

[ ]:
```
G = nx.from_numpy_matrix(Kt)
graphplot = nx.draw(G, X, node_size=40, width=0.5,)
plt.axis('equal')
plt.show()
```

Note that, for this dataset, the graph connects edges from the same cluster only. Therefore, the number of diagonal blocks in $\overline{\mathbf{K}}_s$ is equal to the number of connected components in the graph.

Note, also, the graph does not depend on the sample ordering in the data matrix: the graphs for any matrix $\mathbf{K}$ and its sorted version $\mathbf{K}_s$ are the same.

1.4 4. The Laplacian matrix

The Laplacian matrix of a given affinity matrix $\mathbf{K}$ is given by

$$\mathbf{L} = \mathbf{D} - \mathbf{K}$$

where **D** is the diagonal **degree matrix** given by

$$D_{ii} = \sum_{j}^{n} K_{ij}$$

1.4.1 4.1. Properties of the Laplacian matrix

The Laplacian matrix of any symmetric matrix **K** has several interesting properties:

P1.

L is symmetric and positive semidefinite. Therefore, all its eigenvalues $\lambda_0, \ldots, \lambda_{N-1}$ are non-negative. Remind that each eigenvector **v** with eigenvalue λ satisfies

$$\mathbf{L} \cdot \mathbf{v} = \lambda \mathbf{v}$$

P2.

L has at least one eigenvector with zero eigenvalue: indeed, for $\mathbf{v} = \mathbf{1}_N = (1, 1, \ldots, 1)^\intercal$ we get

$$\mathbf{L} \cdot \mathbf{1}_N = \mathbf{0}_N$$

where $\mathbf{0}_N$ is the N dimensional all-zero vector.

P3.

If **K** is block diagonal, its Laplacian is block diagonal.

P4.

If **L** is a block diagonal with blocks $\mathbf{L}_0, \mathbf{L}_1, \ldots, \mathbf{L}_{c-1}$, then it has at least c orthogonal eigenvectors with zero eigenvalue: indeed, each block $\mathbf{L}_i$ is the Laplacian matrix of the graph containing the samples in the i connected component, therefore, according to property P2,

$$\mathbf{L}_i \cdot \mathbf{1}_{N_i} = \mathbf{0}_{N_i}$$

where N_i is the number of samples in the i-th connected component.

Therefore, if

$$\mathbf{v}_i = \begin{pmatrix} \mathbf{0}_{N_0} \\ \vdots \\ \mathbf{0}_{N_{i-1}} \\ \mathbf{1}_{N_i} \\ \mathbf{0}_{N_{i+1}} \\ \vdots \\ \mathbf{0}_{N_{c-1}} \end{pmatrix}$$

then

$$\mathbf{L} \cdot \mathbf{v}_i = \mathbf{0}_N$$

We can compute the Laplacian matrix for the given dataset and visualize the eigenvalues:

[ ]:
```
Dst = np.diag(np.sum(Kst, axis=1))
Lst = Dst - Kst

# Next, we compute the eigenvalues of the matrix
w = np.linalg.eigvalsh(Lst)
plt.figure()
plt.plot(w, marker='.');
plt.title('Eigenvalues of the matrix')
plt.show()
```

Exercise 4: Verify that $\mathbf{1}_N$ is an eigenvector with zero eigenvalues. To do so, compute $\mathbf{L}_{st} \cdot \mathbf{1}_N$ and verify that its euclidean norm is close to zero (it may be not exactly zero due to finite precission errors).

Verify that vectors $\mathbf{v}_i$ defined above (that you can compute using `vi = (ys==i)`) also have zero eigenvalue.

[ ]:
```
# <SOL>
print(np.linalg.norm(Lst.dot(np.ones((N,1)))))
for i in range(nc):
    vi = (ys==i)
    print(np.linalg.norm(Lst.dot(vi)))
# </SOL>
```

Exercise 5: Verify that the spectral properties of the Laplacian matrix computed from $\mathbf{K}_{st}$ still apply using the unsorted matrix, $\mathbf{K}_t$: compute $\mathbf{L}_t \cdot \mathbf{v}'_i$, where $\mathbf{v}'_i$ is a binary vector with components equal to 1 at the positions corresponding to samples in cluster i (that you can compute using `vi = (y==i)`), and verify that its euclidean norm is close to zero.

[ ]:
```
# <SOL>
Dt = np.diag(np.sum(Kt, axis=1))
Lt = Dt - Kt
print(np.linalg.norm(Lt.dot(np.ones((N,1)))))
for i in range(nc):
    vi = (y==i)
    print(np.linalg.norm(Lt.dot(vi)))
# </SOL>
```

Note that the position of 1's in eigenvectors $\mathbf{v}_i$ points out the samples in the i-th connected component. This suggest the following tentative clustering algorithm:

1. Compute the affinity matrix
2. Compute the laplacian matrix
3. Compute c orthogonal eigenvectors with zero eigenvalue
4. If $v_{in} = 1$, assign $\mathbf{x}^{(n)}$ to cluster i.

This is the grounding idea of some spectral clustering algorithms. In this precise form, this algorithm does not usually work, for several reasons that we will discuss next, but with some modifications it becomes a powerfull method.

1.4.2 4.2. Computing eigenvectors of the Laplacian Matrix

One of the reasons why the algorithm above may not work is that vectors $\mathbf{v}'_0, \ldots, \mathbf{v}'_{c-1}$ are not the only zero eigenvectors or $\mathbf{L}_t$: any linear combination of them is also a zero eigenvector. Eigenvector computation algorithms may return a different set of orthogonal eigenvectors.

However, one can expect that eigenvector should have similar component in the positions corresponding to samples in the same connected component.

[ ]:
```
wst, vst = np.linalg.eigh(Lst)

for n in range(nc):
    plt.plot(vst[:,n], '.-')
```

[ ]:
```
plt.imshow(vst[:,:nc], aspect='auto')
plt.grid(False)
plt.title('Display of first 4 eigenvectors of Ks')
```

1.4.3 4.3. Non block diagonal matrices.

Another reason to modify our tentative algorithm is that, in more realistic cases, the affinity matrix may have an imperfect block diagonal structure. In such cases, the smallest eigenvalues may be nonzero and eigenvectors may be not exactly piecewise constant.

Exercise 6 Plot the eigenvector profile for the shuffled and not thresholded affinity matrix, **K**.

[ ]:
```
#ă<SOL>
D = np.diag(np.sum(K, axis=1))
L = D - K
w, v = np.linalg.eigh(L)
for n in range(nc):
    plt.plot(v[:,n], '.-')
# </SOL>
```

Note that, despite the eigenvector components can not be used as a straighforward cluster indicator, they are strongly informative of the clustering structure.

- All points in the same cluster have similar values of the corresponding eigenvector components $(v_{n0}, \ldots, v_{n,c-1})$.
- Points from different clusters have different values of the corresponding eigenvector components $(v_{n0}, \ldots, v_{n,c-1})$.

Therefore we can define vectors $\mathbf{z}^{(n)} = (v_{n0}, \ldots, v_{n,c-1})$ and apply a centroid based algorithm (like K-means) to identify all points with similar eigenvector components. The corresponding samples in $\mathbf{X}$ become the final clusters of the spectral clustering algorithm.

One possible way to identify the cluster structure is to apply a K-means algorithm over the eigenvector coordinates. The steps of the spectral clustering algorithm become the following

1.5 5. A spectral clustering (*graph cutting*) algorithm

1.5.1 5.1. The steps of the spectral clustering algorithm.

Summarizing, the steps of the spectral clustering algorithm for a data matrix $\mathbf{X}$ are the following:

1. Compute the affinity matrix, $\mathbf{K}$. Optionally, truncate the smallest components to zero.
2. Compute the laplacian matrix, $\mathbf{L}$
3. Compute the c orthogonal eigenvectors with smallest eigenvalues, $\mathbf{v}_0, \ldots, \mathbf{v}_{c-1}$
4. Construct the sample set $\mathbf{Z}$ with rows $\mathbf{z}^{(n)} = (v_{0n}, \ldots, v_{c-1,n})$
5. Apply the K-means algorithms over $\mathbf{Z}$ with $K = c$ centroids.
6. Assign samples in $\mathbf{X}$ to clusters: if $\mathbf{z}^{(n)}$ is assigned by K-means to cluster i, assign sample $\mathbf{x}^{(n)}$ in $\mathbf{X}$ to cluster i.

Exercise 7: In this exercise we will apply the spectral clustering algorithm to the *two-rings* dataset $\mathbf{X}_2$, using $\gamma = 20$, $t = 0.1$ and $c = 2$ clusters.

- Complete step 1, and plot the graph induced by $\mathbf{K}$

[ ]:
```
# <SOL>
g = 20
t = 0.1
K2 = rbf_kernel(X2, X2, gamma=g)
K2t = K2*(K2>t)
G2 = nx.from_numpy_matrix(K2t)
graphplot = nx.draw(G2, X2, node_size=40, width=0.5)
plt.axis('equal')
plt.show()
# </SOL>
```

- Complete step 2, 3 and 4, and draw a scatter plot of the samples in $\mathbf{Z}$

[ ]:
```
# <SOL>
D2t = np.diag(np.sum(K2t, axis=1))
L2t = D2t - K2t
w2t, v2t = np.linalg.eigh(L2t)
Z2t = v2t[:,0:2]

plt.scatter(Z2t[:,0], Z2t[:,1], s=20)
plt.show()
# </SOL>
```

- Complete step 5

[ ]:
```
est = KMeans(n_clusters=2)
clusters = est.fit_predict(Z2t)
```

- Finally, complete step 6 and show, in a scatter plot, the result of the clustering algorithm

[ ]:
```
plt.scatter(X2[:, 0], X2[:, 1], c=clusters, s=50, cmap='rainbow')
plt.axis('equal')
plt.show()
```

1.5.2 5.2. Scikit-learn implementation.

The spectral clustering algorithm in Scikit-learn requires the number of clusters to be specified. It works well for a small number of clusters but is not advised when using many clusters and/or data.

Finally, we are going to run spectral clustering on both datasets. Spend a few minutes figuring out the meaning of parameters of the Spectral Clustering implementation of Scikit-learn:

http://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html

Note that there is not equivalent parameter to our threshold t, which has been useful for the graph representations. However, playing with γ should be enough to get a good clustering.

The following piece of code executes the algorithm with an 'rbf' kernel. You can manually adjust the number of clusters and the parameter of the kernel to study the behavior of the algorithm. When you are done, you can also:

- Modify the code to allow for kernels different than the 'rbf'
- Repeat the analysis for the second dataset (*two_rings*)

[ ]:
```
n_clusters = 4
gamma = .1  # Warning do not exceed gamma=100
SpClus = SpectralClustering(n_clusters=n_clusters,affinity='rbf',
                            gamma=gamma)
SpClus.fit(X)

plt.scatter(X[:, 0], X[:, 1], c=SpClus.labels_.astype(np.int), s=50,
            cmap='rainbow')
plt.axis('equal')
plt.show()
```

[ ]:
```
nc = 2
gamma = 50  #Warning do not exceed gamma=300

SpClus = SpectralClustering(n_clusters=nc, affinity='rbf', gamma=gamma)
SpClus.fit(X2)

plt.scatter(X2[:, 0], X2[:, 1], c=SpClus.labels_.astype(np.int), s=50,
            cmap='rainbow')
plt.axis('equal')
plt.show()
```

[ ]:
```
nc = 5
SpClus = SpectralClustering(n_clusters=nc, affinity='nearest_neighbors')
SpClus.fit(X2)

plt.scatter(X2[:, 0], X2[:, 1], c=SpClus.labels_.astype(np.int), s=50,
            cmap='rainbow')
plt.axis('equal')
plt.show()
```

1.6 5.2. Other clustering algorithms.

1.6.1 5.2.1. Agglomerative Clustering algorithms

Bottom-up approach:

- At the beginning, each data point is a different cluster
- At each step of the algorithm two clusters are merged, according to certain performance criterion
- At the end of the algorithm, all points belong to the root node

In practice, this creates a hierarchical tree, that can be visualized with a dendogram. We can cut the tree at different levels, in each case obtaining a different number of clusters.

Criteria for merging clusters We merge the two closest clusters, where the distance between clusters is defined as:

- Single: Distance between clusters is the minimum of the distances between any two points in the clusters
- Complete: Maximal distance between any two points in each cluster
- Average: Average distance between points in both clusters
- Centroid: Distance between the (Euclidean) centroids of both clusters
- Ward: We merge centroids so that the overall increment of *{within-cluster} variance is minimum.*

Python implementations Hierarchical clustering may lead to clusters of very different sizes. Complete linkage is the worst strategy, while Ward gives the most regular sizes. However, the affinity (or distance used in clustering) cannot be varied with Ward, thus for non Euclidean metrics, average linkage is a good alternative.

There are at least three different implementations of the algorithm:

- Scikit-learn: Only implements 'complete', 'ward', and 'average' linkage methods. Allows for the definition of connectivity constraints
- Scipy
- fastcluster: Similar to Scipy, but more efficient with respect to computation and memory.

[ ]: