Project 2 - Graph-based Clustering

August Jonasson

October 2024

Task 1 (Spectral clustering)

A

For the first task we are asked to construct a similarity (dissimilarity) graph using an appropriate method (i.e. an undirected, weighted adjacency matrix). Looking at Figure 1, we can tell that the clusters are relatively similar in density, why we choose to use the regular "k nearest neighbors" (kNN) method. Using Proposition 2 from Luxburg, which states that the multiplicity of eigenvalue 0 tells us the number of connected components, we find that $k = 5$ is the lowest k that gives a single connected component.

Regarding the weights of the edges (no self-edges allowed) we started with Gaussian weights but then switched over to 1-degree-of-freedom t-distributed weights when we found out that the decay of the Gaussian was too intense, leading to numbers that were too small to handle without encountering rounding errors. Hence, the weights are given by

$$w_{ij} = \left(1 + \frac{d_{ij}^2}{v}\right)^{-\frac{v+1}{2}},$$

where the d_{ij} represent the distances and $v = 1$ the degrees of freedom.

B

In this task we are asked to implement the unnormalized spectral clustering algorithm from Luxburg. Following this, dividing the data into two clusters represents picking out the first two eigenvectors (sorted according to the eigenvalues in ascending order) of

$$L = D - W,$$

where D is the diagonal degrees matrix and W the weighted adjacency matrix, as the two columns of a new matrix V_{300x2} and performing k-means clustering on the rows of this. However, since the first eigenvector will always be constant

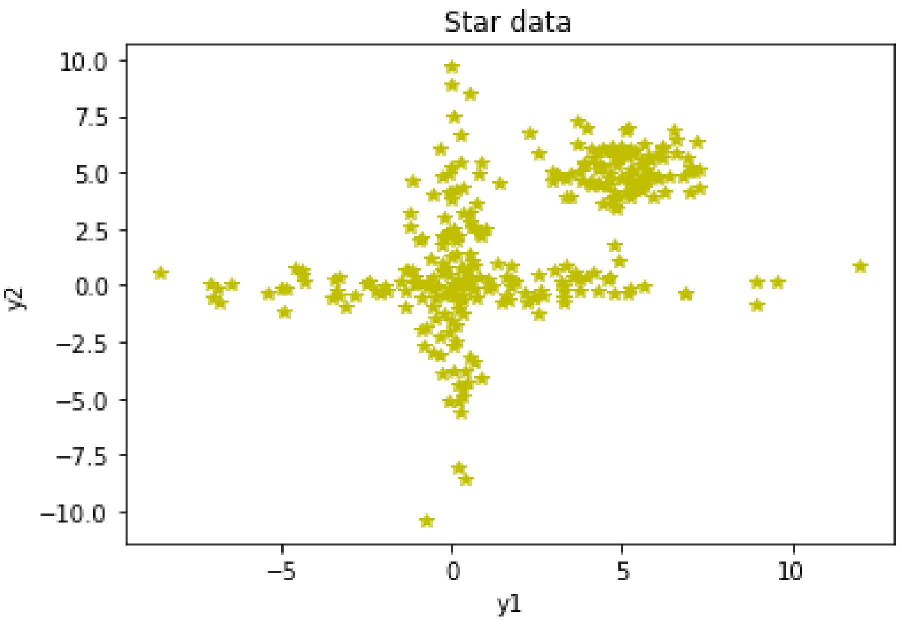

Figure 1: "Star data"

(according to Proposition 1, Luxburg), only the second eigenvector (Fiedler vector) is enough to examine.
Now, looking at Figure 2, does it seem appropriate to use the k-means algorithm to separate the clusters? Well, no, the clusters are not spherical and we do see some spatial imbalance between the clusters as well. The clusters are well separated bu k-means would not be the best choice of algorithm in order to separate these two clusters.

C

We are now more or less asked to repeat the spectral clustering algorithm using the normalized Laplacian

$$L_{sym} = I - D^{-1/2} W D^{-1/2},$$

with the caveat that we now also have to normalize the rows of the matrix V.

From Figure 3 we can tell that the results are evry similar to the unnormalized case. As such, k-means would once again not be the most appropriate algorithm for clustering the embedded data.

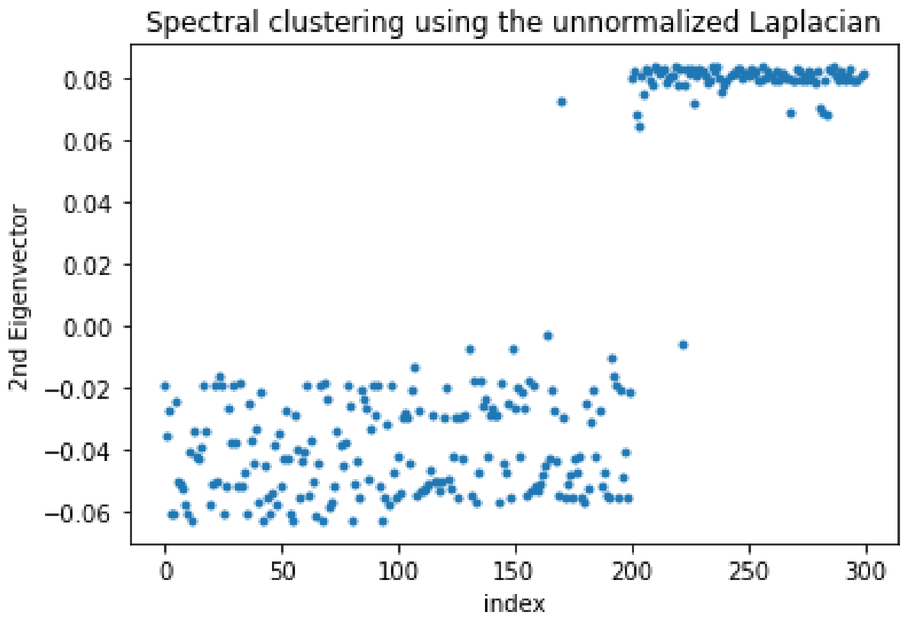

Figure 2: 2nd Eigenvector of the unnormalized Laplacian

D

Since the results from B and C were very similar it is hard to argue which one is more suitable to separating the star data. Arguably, the separation of the clusters is slightly enhanced in the normalized Laplacian case. Also, the within cluster distance looks to be slightly smaller as well in the normalized case. As such, the results from the normalized seems slightly better. However, the algorithm using the normalized Laplacian also requires an initial step of normalizing the rows of the eigen-matrix. For large amounts of data, this might be worth taking into consideration.

In general, though, the normalized Laplacian is preferable. Since it is more robust against imbalanced data.

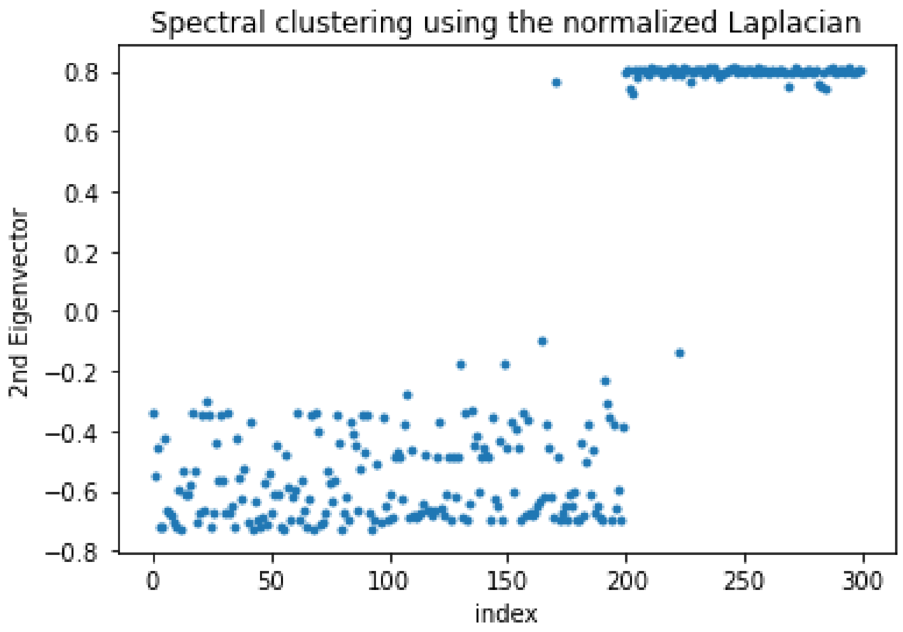

Figure 3: 2nd Eigenvector (rescaled) of the normalized Laplacian

Task 2 (Manifold learning)

A

For the swiss roll data we still choose the kNN method in order to build the graph. This time, $k = 4$ is enough to build a single connected component from the data points. We use the t-distributed weights here as well, since the Gaussian weights once again decayed too quickly.

B

We are now to produce an embedding using the commute-time distance. In this task this is done through the unnormalized Laplacian. The formula that we use is the one found in Proposition 6 of Luxburg. We then perform MDS on the new distance matrix and decide which two principal coordinates best "unfold" the swiss roll. This is done by trial and error and the best result can be seen in Figure 4. This was chosen as the best one based on the fact that the data points are as evenly spread out as possible.

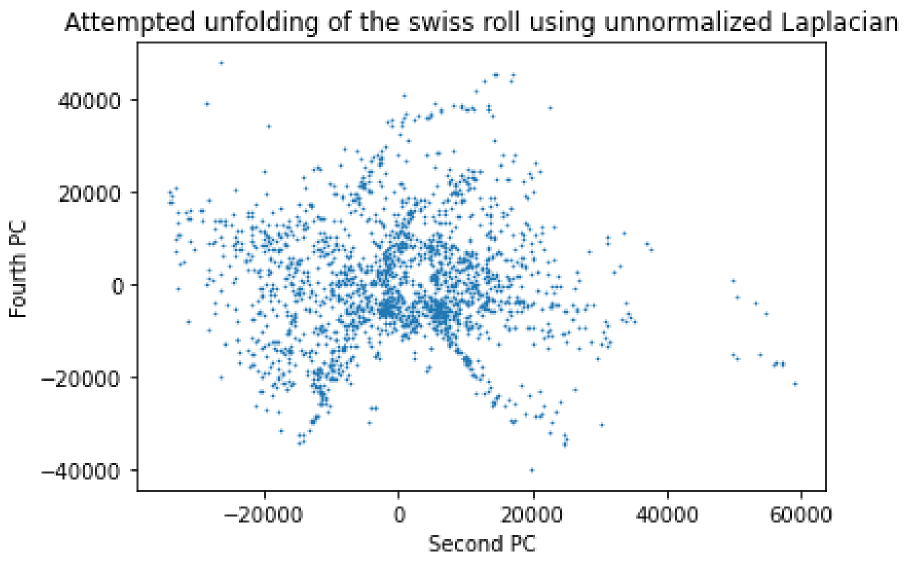

Figure 4: MDS embedding of CTD using the unnormalized Laplacian

C

Repeating Task B but now using the normalized Laplacian in order to create the commute-time distance matrix. Looking at Figure 5 we see the attempt.

D

Comparing the results from B and C it actually looks as if the results from the unnormalized Laplacian better unfolded the swiss roll. At least the points look more evenly spread out. Regarding which method is better, they should actually perform the same and the difference in these plots might be due to some randomness taking place in the built in manifold fitting function (MDS). It actually holds theoretically that

$$c_{ij}^{(L)} = c_{ij}^{(L_{sym})},$$

for all i, j. As such, the method that is computationally most efficient/stable should be chosen.

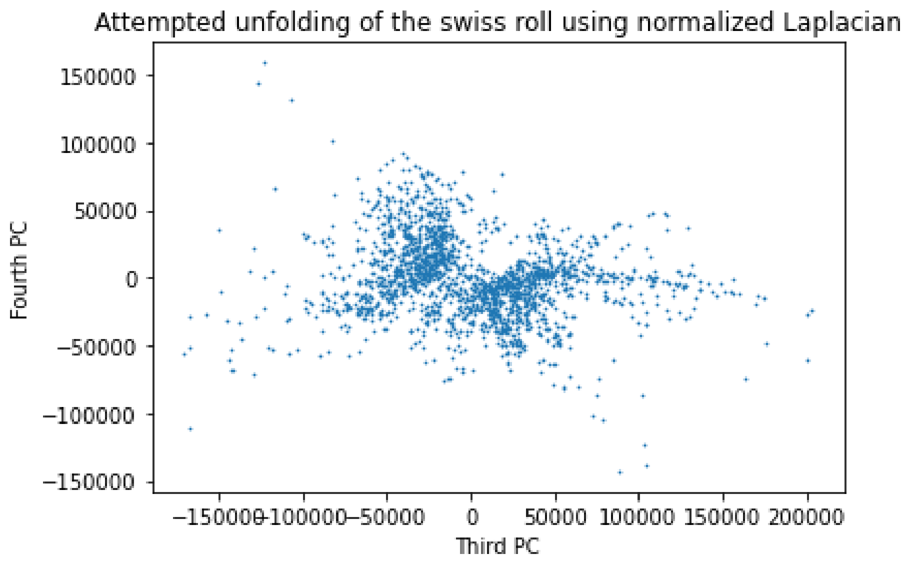

Figure 5: MDS embedding of CTD using the normalized Laplacian

```
import pandas as pd
import scipy as sp
import numpy as np
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt

"""
Star data task
"""

# a 300x2 dataframe (matrix) containing "star data"
df = pd.read_csv("Star_Data.txt", sep='\s+', header=None)
df.columns = ["y1", "y2"]

"""
Can actually see from this plot that k=5 should be enough to get a fully
connected graph using nearest neighbor (undirected, i.e. made symmetric).
That this k will suffice can be seen from the right-most point in the following
plot.
"""
plt.plot(df["y1"], df["y2"], "y*")
plt.title('Star data')
plt.xlabel("y1")
plt.ylabel("y2")
plt.show()

"""
Assuming Euclidean distances are appropriate for the Star_Data.txt file, we
create the dissimilarity matrix (distance matrix) using the
"""
dist_matrix = sp.spatial.distance_matrix(df, df)


"""
Using the sklearn nearest neighbor function in order to create an unweighted
adjacency matrix. Since this function creates self-edges, we have to choose
k+1 as the input to the function.
"""
nbrs = NearestNeighbors(n_neighbors=6,
                        metric="precomputed",
                        algorithm="brute").fit(dist_matrix)

# getting the unweighted adjacency matrix
W = nbrs.kneighbors_graph(dist_matrix).toarray()

# removing self-edges
I = np.identity(300)
W = W - I

# making graph undirected (symmetric)
W = W + np.transpose(W)
W[W > 1] = 1


"""
Weighing the edges using a t-distribution with 1 degree of freedom. Initially,
a Gaussian was used by this decay was to intense and left such small values
that rounding errors became an issue (machine epsilon was reached).
"""
nbr_dists = W * dist_matrix
v = 1 # degrees of freedom
temp_weights = (1 + dist_matrix**2 / v)**(-(1+v) / 2)

# putting the weights into the adjacency matrix W
W = temp_weights * W

# retrieving the diagonal degrees matrix
D = np.diag([sum(row) for row in W])
```

```
# unnormalized Laplacian
L = D - W

"""
Approximating eigenvalues and sorting in ascending order. If there is only one
zero-eigenvalue, then the graph is fully connected (according to proposition
in von Luxburg).
"""

# eigvals and vecs from unnormalized Laplacian
eigvals, eigvecs = np.linalg.eigh(L)

# two clusters => need first two eigenvectors
V = eigvecs[:, :2]

# plotting the eigvals in order to deduce if there are any spectral gaps
plt.plot(eigvals, ".", label = "eigenvalues")
plt.show()

plt.plot(V[:,1], ".")
plt.title("Spectral clustering using the unnormalized Laplacian")
plt.ylabel("2nd Eigenvector")
plt.xlabel("index")
plt.show()

# kmeans_star = KMeans(n_clusters = 2).fit(V)

"""
Now to the normalized Laplacian
"""

D_invsqrt = sp.linalg.pinv(np.sqrt(D))
L_sym = np.matmul(np.matmul(D_invsqrt, L), D_invsqrt)

eigvals_sym, eigvecs_sym = np.linalg.eigh(L_sym)

V_sym = eigvecs_sym[:, :2]
U = []

for i in range(len(V_sym)):
    temp = []
    for j in range(len(V_sym[0])):
        temp.append(V_sym[i, j] / np.sqrt(sum(V_sym[i]**2)))
    U.append(temp)

U = np.array(U)
```

```
plt.plot(eigvecs_sym[:,0], ".")
plt.show()

plt.plot(U[:,1], ".")
plt.title("Spectral clustering using the normalized Laplacian")
plt.ylabel("2nd Eigenvector")
plt.xlabel("index")
plt.show()


"""
Swiss Roll Task 2
"""

df = pd.read_csv("Swiss_Roll.txt", sep='\s+', header=None)
df.columns = ["y1", "y2", "y3"]

num_rows = df.shape[0]
norm = np.linspace(0, 1, num_rows)
cmap = cm.get_cmap('viridis')
colors = cmap(norm)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(df["y1"], df["y2"], df["y3"], c=colors)
ax.set_title('3D Swiss Roll with Continuous Color Change')
ax.set_xlabel('Y1')
ax.set_ylabel('Y2')
ax.set_zlabel('Y3')
plt.show()


dist_matrix = sp.spatial.distance_matrix(df, df)

nbrs = NearestNeighbors(n_neighbors=5,
                        metric="precomputed",
                        algorithm="brute").fit(dist_matrix)

W = nbrs.kneighbors_graph(dist_matrix).toarray()

I = np.identity(2000)
W = W - I
W = W + np.transpose(W)
W[W > 1] = 1
```

```
nbr_dists =W * dist_matrix
v = 1 # degrees of freedom
temp_weights = (1 + dist_matrix**2 / v)**(-(1+v) / 2)

W= temp_weights * W
D = np.diag([sum(row) for row in W])
L = D - W

eigvals, eigvecs = np.linalg.eigh(L)

volV = np.sum(D)

L_pinv = sp.linalg.pinv(L)
CTD_un = np.zeros((len(df), len(df)))
for i in range(len(df)):
    for j in range(len(df)):
        CTD_un[i,j] = volV*(L_pinv[i,i] - 2*L_pinv[i,j] + L_pinv[j,j])

CTD_un = np.array(CTD_un)
eigvals_un, eigvecs_un = np.linalg.eigh(CTD_un)

embedding_un = MDS(n_components = 5, dissimilarity = "precomputed")
MDS_un = embedding_un.fit_transform(CTD_un)


plt.scatter(MDS_un[:,1], MDS_un[:,3], s=0.5)
plt.xlabel("Second PC")
plt.ylabel("Fourth PC")
plt.title("Attempted unfolding of the swiss roll using unnormalized Laplacian")
plt.show()

"""
Now to the normalized Laplacian
"""

D_invsqrt = sp.linalg.pinv(np.sqrt(D))
L_sym = np.matmul(D_invsqrt, np.matmul(L, D_invsqrt))
eigvals_sym, eigvecs_sym = np.linalg.eigh(L_sym)

# based on the Laplacian L_sym
CTD_n = np.zeros((len(df), len(df)))
L_sym_pinv = sp.linalg.pinv(L_sym)
for i in range(len(df)):
    for j in range(len(df)):
        CTD_n[i,j] = volV*(L_sym_pinv[i,i]- 2*L_sym_pinv[i,j] +L_sym_pinv[j,j])
```

```
CTD_n = np.array(CTD_n)
eigvals_n, eigvecs_n = np.linalg.eigh(CTD_n)
plt.plot(eigvals_n, ".")

embedding_n = MDS(n_components = 5, dissimilarity = "precomputed")
MDS_n = embedding_n.fit_transform(CTD_n)

plt.scatter(MDS_n[:,2], MDS_n[:,3], s = 0.5)
plt.xlabel("Third PC")
plt.ylabel("Fourth PC")
plt.title("Attempted unfolding of the swiss roll using normalized Laplacian")
plt.show()
```