Homework of Eigenvalues and Eigenvectors Approximation Report

Polytechnic University of Turin

Computational linear algebra for large scale problems - 01TWYSM

Caffaro Fabio
*s*276842

Introduction

For the resolution of this homework I have decided to use python language. To manage the classes and functions needed to perform the task I have implement two modules. The first one is called `sparse`. In this module I have implemented three classes of sparse matrices:

- `LIL`, a List-Of-Lists sparse matrix.
- `Unweighted_LIL`, this is a simple variation of the LIL format in which the internal lists instead of containing tuples (pos, value) memorize only the position. The original idea was to try to use this implementation for the adjacency matrix.
- `CSR`, a Compressed Sparse Row matrix format or also called Yale format

In this three classes I have implemented some utility methods and some methods to perform basic operation needed to complete the homework. For example: `from_list` to initialize the matrix from a list of tuples `(row, col)` (for the `unweighted_LIL`) or `(row, col, weight)` (for the other formats), `visualize` to plot the sparsity pattern and most importantly `dot` where I redefined the matrix-vector product taking advantage of the characteristic of the specific matrix. Note that not all the methods are present in all the classes. I have implemented mostly the ones needed for the homework.

The second module is called `graph`. In this one I have defined a class `Graph` that acts as an interface to manage the loading of the edges from a file and the construction of the matrices associated with the graph (the adjacency matrix, and the two laplacian matrices). In this case I have designed the methods: `from_file` to initialize the list of edges and the adjacency matrix, `add_edge` to add a single edge at the graph, `spy` to visualize the graph adjacency matrix sparse pattern, `laplacian_matrix` that returns the Laplacian matrix and `norm_laplacian` that returns the normalized Laplacian matrix.

1 Adjacency Matrix

1.1 Construct the adjacency matrix in a sparse storage format

For the first part of the homework I have defined a variable of type Graph specifying the path of the file containing the edges. The adjacency matrix is initialized automatically in the `__init__` method of the class Graph if a path is given, otherwise manually using the `from_file` method. The structure of the file containing the edges is supposed to be as the one provided. The file presents a header with some meta-information on the graph like name, number of nodes and edges. I've decide to extract them manually and then load the edges using `numpy.loadtxt`.

```
1 G = Graph(path='edge_file_facebook_python.txt', adj_format=CSR)
```

For the adjacency matrix, the first thought was to use a `unweighted_LIL` format. This format is very easy to use to initialize a matrix and was designed with the idea of saving some memory space in the construction of the adjacency matrix, avoiding the storage of the weights. In fact, since all the weights for the adjacency matrix are ones this information is redundant and so we can omit it. Only the adjacencies must be stored.

$$
\begin{array}{lcl}
 & \overbrace{}^{keys} & \overbrace{}^{values/adjacencies} \\
\text{row/col 0:} & '0' & \longrightarrow \quad [a_{01}, \ \ldots, \ a_{0k}] \\
\text{row/col 1:} & '1' & \longrightarrow \quad [a_{11}, \ \ldots, \ a_{1k}] \\
\ldots & \vdots & \longrightarrow \quad \vdots \\
\text{row/col n:} & 'n' & \longrightarrow \quad [a_{n1}, \ \ldots, \ a_{nk}]
\end{array}
$$

Roughly speaking, since the memory size of a python tuple (int, int) is of 64 bytes, while a single int object occupy 28 bytes the memory savage is around $(64-28) * num\_of\_edges$. In our example we have $36 * 88234 = 3176424B = 3.18MB$. Note that in reality the reduction is even more, due to other optimization done by the system.

However the main drawback of this format is that does not perform well in the matrix-vector multiplication. This step is essential to the execution of the power iteration method needed in the following steps.

So I've tried to optimize the dot product. Considering the dot product $w = Av$

and since A contains only 1, a component of the result vector $w_i = a_i v$ is equal to the sum of the elements of the vector v corresponding to ones in the row a_i or in other words to the adjacencies of node i

```
    def dot(self, v):
        res = np.zeros(self.shape[0])
        for k in self.data.keys():
            res[k] = v[self.data[k]].sum()
        return res
```

I have tested the performance of this implementation using `timeit.timeit()` function, and even with the optimizations it was still about two times slower than the one in CSR format. CSR format in fact was designed in order to express the matrix-vector multiplication in very smart and performant way. So finally I have decided to opt for this format to store the adjaceny matrix.

1.2 Graphically visualize the sparsity pattern of the matrix A

The function `plt.spy()` requires a matrix in `scipy.sparse` format. For this reason I have decided to implement the method `visualize` in each class of sparse matrix, to plot the sparsity pattern. The function `spy` of the class `Graph` is simply a wrapper method that calls the method `visualize` of the adjacency matrix.

In the image (fig. 1) we can clearly notice several blocks. These blocks represent clusters or networks of people. People in the same network (or block) are strictly related between them and almost totally separated from people belonging to other networks. This particular structure can give us insight on the properties of the matrix. For example, it can suggest us that the space of the matrix is composed by almost separate sub-spaces. Moreover it show us how the graph is almost disconnected.

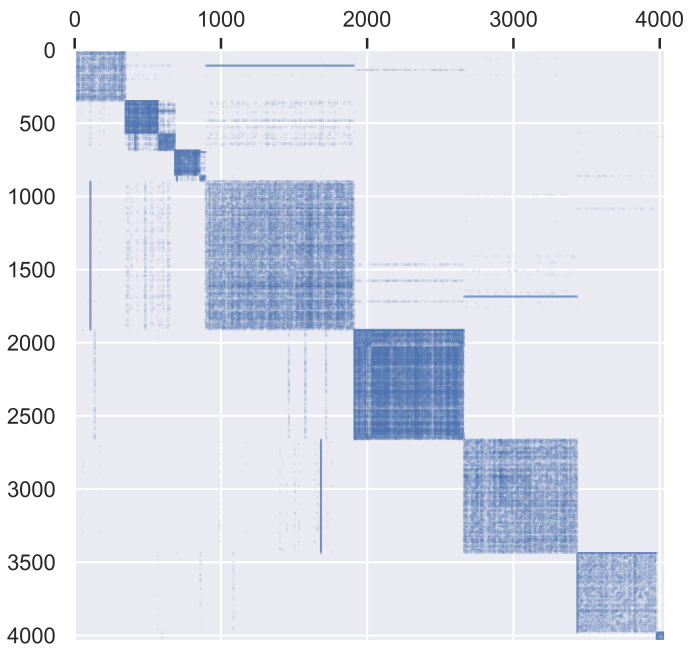

Figure 1: Sparsity pattern of the adjacency matrix

1.3 Look at the definition of the matrix A. What can you say about the spectrum of the adjacency matrix of our undirected graph? And what can you say about its eigenvectors?

Since we are working with an undirected graph, the corresponding adjacency matrix is real and symmetric. This means that all its eigenvalues are real-valued and the set of correspondent eigenvectors form an orthogonal basis of the space spanned by the columns of A.

Moreover, since A has all zeros on the main diagonal, its trace $tr(A) = 0$, and hence the sum of the eigenvalues is zero. This means that the matrix has positive and negative eigenvalues.

1.4 Eigenvector centrality

The first step to compute the eigenvector centrality score is to retrieve the largest eigenvalue and the corresponding normalized eigenvector of the adjacency matrix. To do this I have implemented a classic Power Iteration method with use

of the Rayleigh quotient:

```python
def power_iteration(A, v = None, tol = 1e-15, maxIter = 100):

    n = A.shape[0]

    if v is None:
        v = np.random.rand(n)

    v = v/np.linalg.norm(v)
    v_next = A.dot(v)
    lam = np.dot(v,v_next)

    k = 0
    while k < maxIter:
        k +=1
        lam_old = lam
        v = v_next/np.linalg.norm(v_next)       # normalization
        v_next = A.dot(v)                       # power iteration
        lam = np.dot(v, v_next)                 # Rayleigh quotient

        if np.abs(lam_old-lam) < tol:
            break

    return lam, v_next/np.linalg.norm(v_next)
```

The method receive as parameter the matrix, and optionally a starting vector and tolerance and maximum number of iterations for stopping the iteration. A fundamental point is that the method requires that the matrix implements a method `dot`. This is necessary to perform the matrix-vector multiplication treating the matrix with a black-box approach. In this manner the method can work indiscriminately, both with scipy sparse matrices and with the sparse matrices that I have implemented.

For this reason, In the class `CSR` I have also implemented the method `dot` to perform the matrix vector multiplication.

```python
def dot(self, v):
    if type(v) is not np.ndarray:
        v = np.array(v)

    res = []
    for ind_i, ind_f in zip(self.row_ptr, self.row_ptr[1:]):
        a = self.val[ind_i:ind_f]
```

```
            b = v[self.col_ind[ind_i:ind_f]]
            res.append(np.dot(a,b))
        return np.array(res)
```

Having a look at the first eigenvector we can see something interesting. As we might have expected, the eigenvector has mostly components in the range 1900-2700 approximately and it's almost independent on the rest of the nodes (fig. 2). This is not surprising since, as said before, the matrix is composed by almost independent blocks and since the range 1900-2700 correspond exactly to the block number 7. This block is one of the biggest and fittest one, hence one of the most significant for the whole graph.

Figure 2: Components of the eigenvector corresponding to the eigenvalue with maximum absolute value

The node with the highest eigenvector centrality it's the node 1912, highlighted in the two plots (fig. 2 and fig. 3). This node falls exactly in the '1900-2700' block. Its eigenvector centrality is 0.095 and its degree is 755. We can also notice that the maximum degree in the graph is 1045 for node 107, so this node is not the node with the highest degree. However, as stated before it is a node

in very fit cluster, so adjacent to nodes with a high degree. The eigenvector centrality score in fact, not only consider the degree of the node itself, but also the degrees of its adjacency nodes.

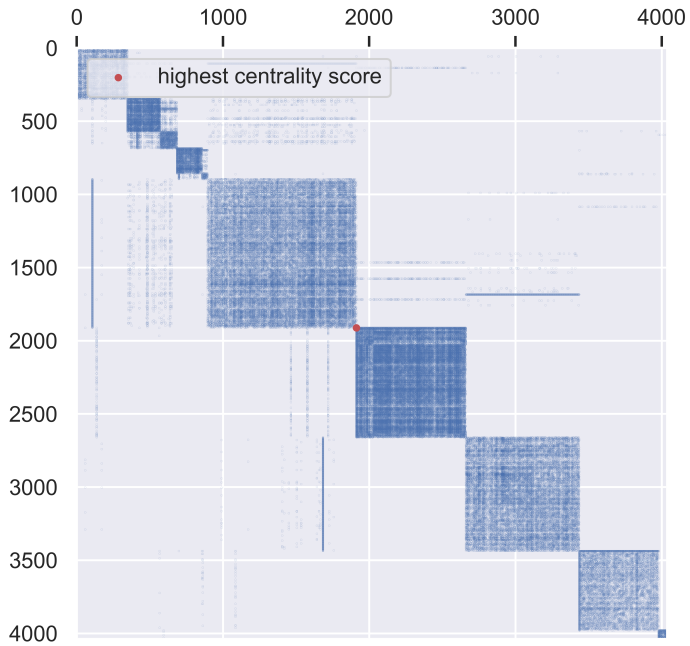

Figure 3: Sparsity pattern of the adjacency matrix. Highlighted the node with highest eigenvector centrality score

1.5 Test of results and comparison with scipy implementation

I've organized this part of the code to display the results in three sub-parts: my implementation, scipy implementation and comparison. In the first two parts I retrieve the first eigenvalue and the corresponding eigenvector with my implementation and scipy one.

In these two parts I show the results and examine the correctness checking the Chebyshev norm $\|Av - \lambda v\|_\infty$. This is in the order of 10^{-11} for my implementation and 10^{-15} for scipy approximation (fig. 4). The node with highest eigenvector centrality it's the number 1912 for both the implementation and its value is 0.095.

In the third part I compare the two implementations. First I checked the error

between the approximations of the eigenvalue as the absolute value of the difference of the two $|\lambda_{my\_imp} - \lambda_{sp\_imp}|$. This in the order of 10^{-13}, very close to machine precision, so they are practically the same. Then I check the maximum error between the two approximations of the eigenvector as the Chebyshev distance between the two eigenvectors $\|v_{my\_imp} - v_{sp\_imp}\|_\infty$ and the dot product between them to verify that they are parallel. Finally I compare the computational time required for both implementations. Here we can see the main drawback of my implementation, that is about two order of magnitude more time-expensive than the scipy implementation. However, this result is comprehensible considering that scipy is a very optimized library implemented taking advantage of low level C language functionalities.

```
(Comparison)
   Eigenvalue approx error: 2.2737367544323206e-13
   Max eigenvector approx error: 5.0437703834197486e-09
   Dot product between the two eigenvectors:  0.9999999999999986
   Time for the computation of the eigenvalues: 1.017s (my implementation), 0.011s (sp implementation)
```

Figure 4: Comparison of the results obtained with my implementation and with scipy one

2 Laplacian matrix

```
1 laplacian = G.laplacian_matrix()
```

The Laplacian matrix L is constructed with the `laplacian_matrix` method of the Graph class. For this matrix I've decided to use the LIL format. This choice was done since only a check of the correctness of the construction is needed on this matrix and in particular no matrix-vector multiplication are required. Note that in this case define a dot product would have been more difficult and probably slower, since it would have dealt with tuples (ind, weight).

The verification of the correctness of the construction is done using the method `get_rows()` of the laplacian matrix and veryifing that each row sums to 0

```
1 sum_of_rows = [np.sum(x) if x is not None else 0 for x in
  ↪  laplacian.get_rows()]
2 print(f"\nEach row of the Laplacian matrix sums to 0:
  ↪  {np.any(sum_of_rows != 0)}")
```

2.1 Properties of the Laplacian matrix

As for the adjacency matrix, the Laplacian matrix is also symmetric and real-valued. Hence, its spectrum is real and the set of its eigenvectors form an orthonormal base.

Moreover, we can observe that the matrix L can also be written as $L = SS^T$ where S is the matrix whose rows are the nodes of the graph and whose columns correspond to the edges. Considering a column s corresponding to an edge (i, j), the entries of the matrix S are such that:

- the entry s_i is equal to 1
- the entry s_j is equal to -1
- all the other entries in the column are equal to 0

For this reason the Laplacian matrix is positive semidefinite and so has non-negative eigenvalues. Finally, it is diagonal-dominant and since all the rows sums to zero, the smallest eigenvalue λ_1 is equal to 0 and the corresponding eigenvector is the vector composed by all ones. In fact we have:

$$\begin{bmatrix} -l_1- \\ -l_2- \\ \dots \\ -l_n- \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} l_{1i} \\ \sum_{i=1}^{n} l_{2i} \\ \dots \\ \sum_{i=1}^{n} l_{ni} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} = \lambda_1 * \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} \tag{1}$$

So $\lambda_1 = 0$. In particular the multiplicity of the eigenvalue 0 is equal to the number of connected components of the graph G (from Perron-Frobenius Theorem).

3 Normalized Laplacian Matrix

The normalized Laplacian matrix L_n is constructed with the `norm_laplacian_matrix` method of the class Graph.

```
norm_laplacian = G.norm_laplacian_matrix()
```

In this case, due to the need of storing all the weights and have a performant matrix-vector multiplication, I have again opted to store the matrix in a CSR format.

3.1 Computation of second smallest eigenvalue

The eigenvalues of the normalized Laplacian matrix L_n always lies in the range between 0 and 2. If we consider the matrix $M = 2I - L_n$ where I is the identity matrix of dimension $n \times n$ and the eigenvalues of the normalized Laplacian matrix $0 \leq \lambda_i \leq 2$, then the eigenvalues of the matrix M are $\mu_i = 2 - \lambda_i$. So considering λ_n the largest eigenvalue of the matrix L_n and λ_1 the smallest one, we have[1]:

$$\mu_n = 2 - \lambda_1 \geq \mu_{n-1} = 2 - \lambda_2 \geq \ldots \geq \mu_0 = 2 - \lambda_n \tag{2}$$

Now, we can apply the Power Iteration method to the matrix M in order to retrieve its largest eigenvalue μ_n. Then, using a deflation technique we can remove it to the matrix and apply again the Power Iteration method to the matrix $M - \mu_n \, v_n x_n^T$, where v_n is the column vector containing the eigenvector corresponding to μ_n and x_n^T is a row vector such that $x_n^T v_n = 1$.

In this manner we retrieve the second largest eigenvalue of the matrix M, μ_{n-1}. Now to recover the second smallest eigenvalue of the matrix L_n, we simply compute $\lambda_2 = 2 - \mu_{n-1}$.

```
# M = 2I - Ln
M = 2*sp.sparse.eye(sp_norm_laplacian.shape[0]) - sp_norm_laplacian
# computation of largest eig of M
eig, eigv = power_iteration(M, maxIter=1000)
# deflation
M = M - eig*eigv[:, sp.newaxis]*eigv[sp.newaxis,:]

eig2, eigv2 = power_iteration(M, maxIter=1000)
eig2 = 2 - eig2
```

Another way to retrieve the second smallest eigenvalue is to consider the fact that 0 is an eigenvalue of the Laplacian matrix L and so also of the normalized Laplacian matrix L_n. This means that the first eigenvalue of the matrix M is $\mu_n = 2 - \lambda_1 = 2$. The associated eigenvector is $v_1 = D^{1/2}\mathbf{1}$. Where D is the diagonal matrix containing the degrees and $\mathbf{1}$ is the column vector containing all ones. We can verify this empirically computing the eigenvector and checking that $L_n v_1 - \lambda_1 v_1 = 0$:

```
comp_eigv = sp.sparse.csr_matrix(np.diag(G.degrees)).power(1 /
↪  2).dot(np.ones(4039))
```

[1] Note that this relation is still valid substituting 2 with any constant $C \geq 2$

```
comp_eigv = comp_eigv / np.linalg.norm(comp_eigv, 2)
print(np.linalg.norm(M.dot(eigv_prova) - 2*eigv_prova, np.inf))
>> 1.8318679906315083e-15
```

Knowing that, we can compute the eigenvector and apply directly the deflation.

```
M = 2 * sp.sparse.eye(norm_laplacian.shape[0]) - sp_norm_laplacian
  ↪ # M = 2I - Ln
M = M - 2 * comp_eigv[:, sp.newaxis] * comp_eigv[sp.newaxis, :]
  ↪ # deflation

eig2, eigv2 = power_iteration(M, maxIter=1000)
eig2 = 2 - eig2
```

Finally, I have found also a third way to retrieve the second smallest eigenvalue of the normalized Laplacian matrix using the shitf-inverted power iteration method. However this way is more an experiment that I have done since it is more empirical and is based on some considerations on the matrix. The shift-inverted iteration requires a shift μ and applies the power iteration to the shifted matrix $(X - \mu I)^{-1}$, where X is the original matrix that we are analyzing.

If λ_i are the eigenvalues of X, $\sigma_i = \frac{1}{\lambda_i - \mu}$ are the eigenvalues of the inverse-shifted matrix. In this manner, the iteration converge to the eigenvalue σ_k corresponding to the eigenvalue λ_k closest to μ.

Knowing that the smallest eigenvalue of L_n is 0 and knowing that the graph is almost disconnected we can imagine that the second eigenvalues is relatively small. So we can try to apply the shift-invert method with a shift μ small. Now, the problem is that we don't have a clear way to measure how small. Maybe with some experience, one can have a guess. However I have tried with a brute force approach starting from $\mu = 1e-15$ and progressively increasing it. The method converge incredibly fast to the second smallest eigenvalue till a value of $\mu = 1e-3$.

Doing these experiments, I have noted something interesting. Even applying $\mu = 0$ the method converge to the desired eigenvalue. I think that this is one of the few cases in which machine arithmetic can help us. In fact, I think that this strange behavior can be explained considering that maybe, due to round effects the matrix L_n is like as if it was already been shifted by a very small value. Hence, the invert power method converge to the second smallest eigenvalue.

```
    eig2, eigv2 = inverse(norm_laplacian, mu=1e-5)
```

I have compared the results of all these methods, with an implementation with scipy (fig. 5 and fig. 6). For each method I have compared the approximation error of the pair eigenvalue/eigenvector looking at the maximum error of $\|Av - \|_\infty$, the time required for the computation, the absolute value of the difference of my eigenvalues and the eigenvalue obtained with scipy and the l2 distance between my eigenvectors and the one obtained with scipy. I have used a normalized Laplacian matrix in a scipy sparse format in order to take advantage of the methods to perform the LU decomposition and the operations of matrix-matrix subtraction, without having to implement them.

The converge is much slower than before, for this reason I have incremented the maximum number of iterations to 1500.

With the first method, maximum error of the approximation is in the order of 10^-5. The eigenvalue retrieved with this method is very close to the one retrieved with scipy. With the second method the results are similar to the

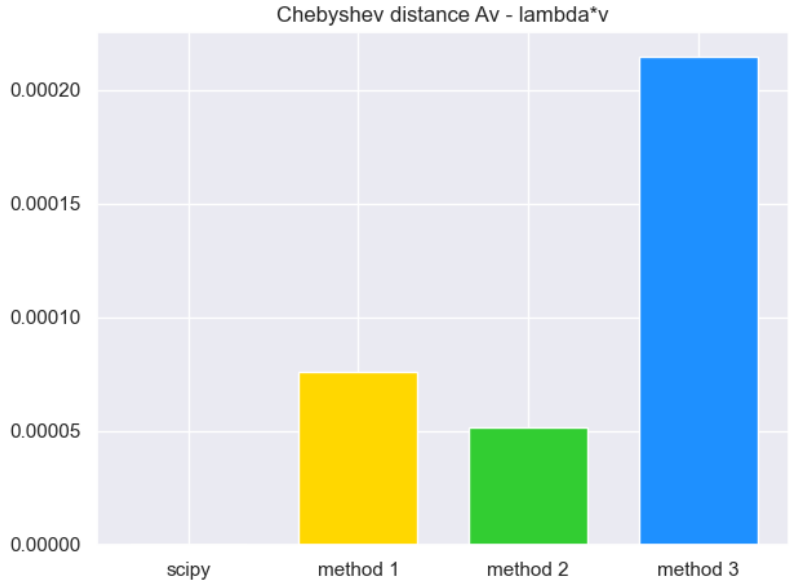

Figure 5: Maximum error computed in the approximation of the pair eigenvalue/eigenvector

first one. The approximation of the pair eigenvalue/eigenvector seems to be a bit better, but how we can see from the plot the error of both eigenvalue and eigenvector with respect to scipy are greater than the ones of method 1. The time required is practically the same of method 1. I would have expected to have better performances with the second method, since only one power iteration is

needed.

Finally, we can see how with the Inverse method probably we have the best results. The time required is extremely low, even lower than scipy implementation. The approximation is not perfect, but the maximum error is in the order 10^{-4} so it could be acceptable. Both eigenvalue and eigenvector computed with this method are the closest to scipy results.

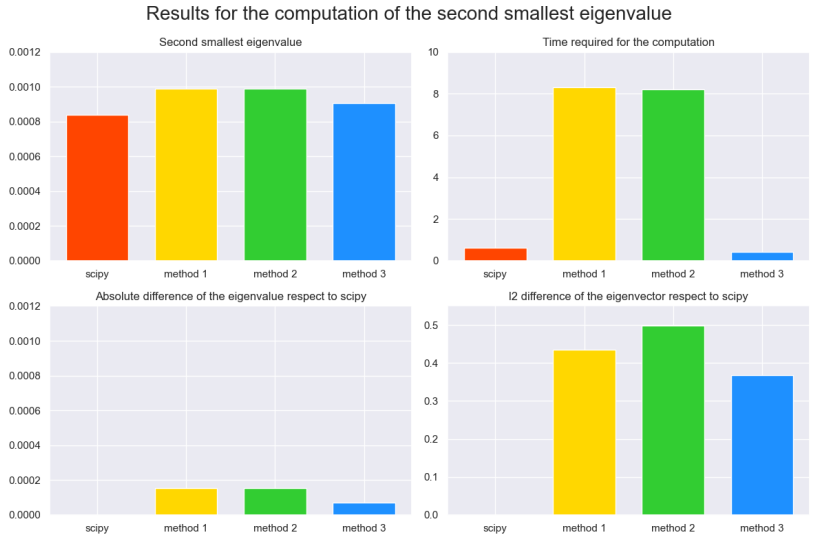

Figure 6: Comparison of the results

3.2 Interpretation of the result

The algebraic multiplicity of the eigenvalue 0 of the Laplacian and of the normalized Laplacian matrix of a graph, corresponds to the number of disconnected components present in the graph. Hence, if the second smallest eigenvalue is equal to 0, we have two disconnected components.

In our example the multiplicity of the 0 eigenvalue is equal to 1. This means that the graph is connected. However the value of the second smallest eigenvalue is quite low. This means that there are at least two components in the graph with low 'interactions'.

As we have seen before, the adjacency matrix of the graph is composed by blocks almost separated. In particular looking again at the sparsity pattern (fig. 1) we can notice how the last two blocks (starting from around node 3500) are almost completely isolated. Their only interactions with the rest of the graph are with nodes around 1000. This explain the small value of λ_2.