✉ lorenzo.dallamico@unito.it

di.unito.it

Lecture 20.ns14

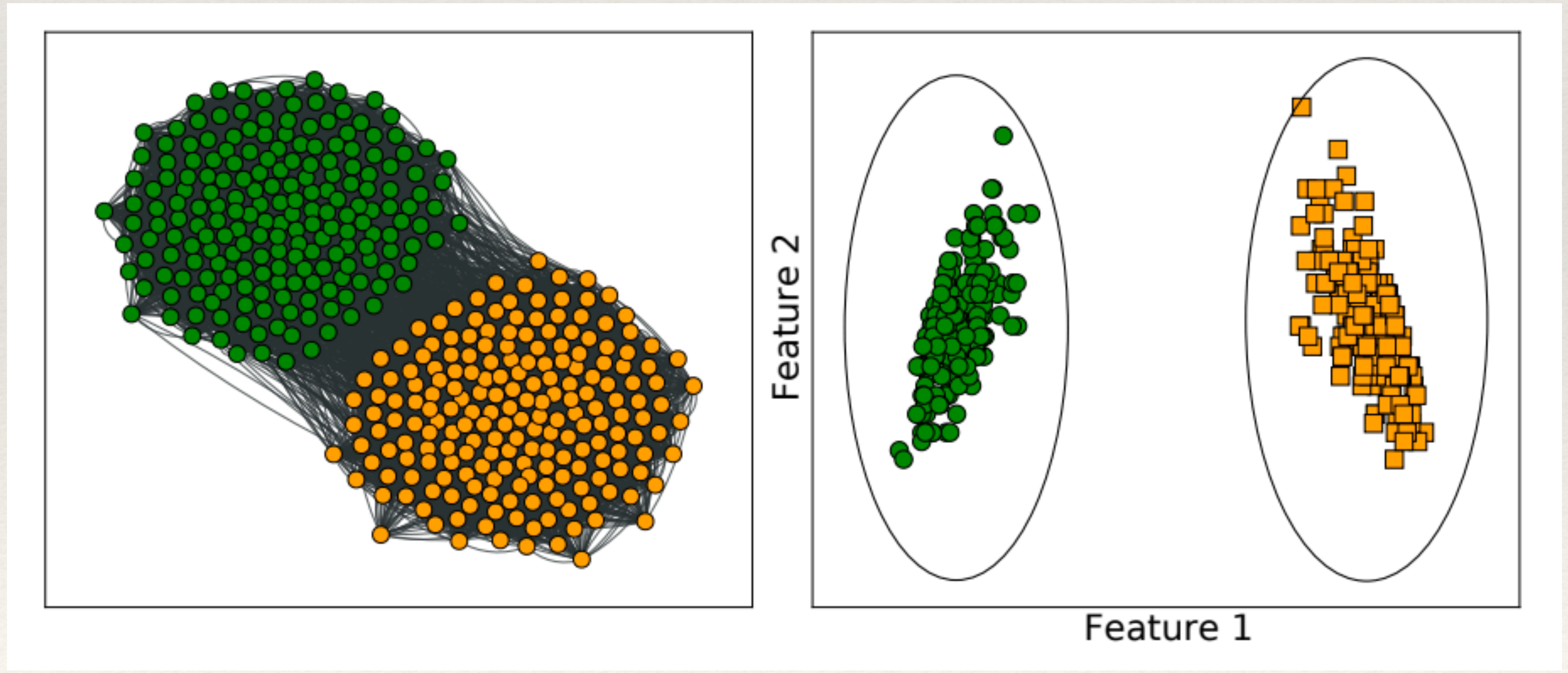
Communities 3

Course: **Complex Networks Analysis and Visualization**
Sub-Module: **NetSci**

Node embeddings (recap)

Typical community detection algorithm with node embeddings:

1. Represent each node of the graph as a vector in d dimensions $i \rightarrow x_i \in \mathbb{R}^d$
2. Cluster the points in the d dimensional space with (for instance) k-means



Spectral clustering (recap)

This methods consists in creating a node embedding using the eigenvectors of appropriate graph matrix representations

Spectral clustering

1. Build M, a graph matrix representation (a function of the adjacency matrix)
2. Compute q eigenvectors (largest or smallest according to M) of M and store them in a matrix X of size (n, q). Each row corresponds to a node and is a vector of q dimensions
3. Use k-means on X

Motivation (some examples)

When we write an cost function on a graph, it is commonly easy to write it in matrix form. Recall that if v is an eigenvector of M, then the following relation holds

$$v^T M v = \lambda v^T v = \lambda$$

We have seen the following relation involving the Laplacian matrix L = D-A

$$y^T L y = \frac{1}{2} \sum_{i,j} A_{ij} (y_i - y_j)^2$$

Motivation (some examples)

$$y^T L y = \frac{1}{2}\sum_{i,j} A_{ij}(y_i - y_j)^2$$

This is a **cost function** that we want to minimize over y, while imposing that y = 1 is not a good solution (everyone in the same class). One can easily see that the solution to this optimization problem is the eigenvector associated to the second smallest (we are minimizing the cost) eigenvalue.

Motivation (some examples)

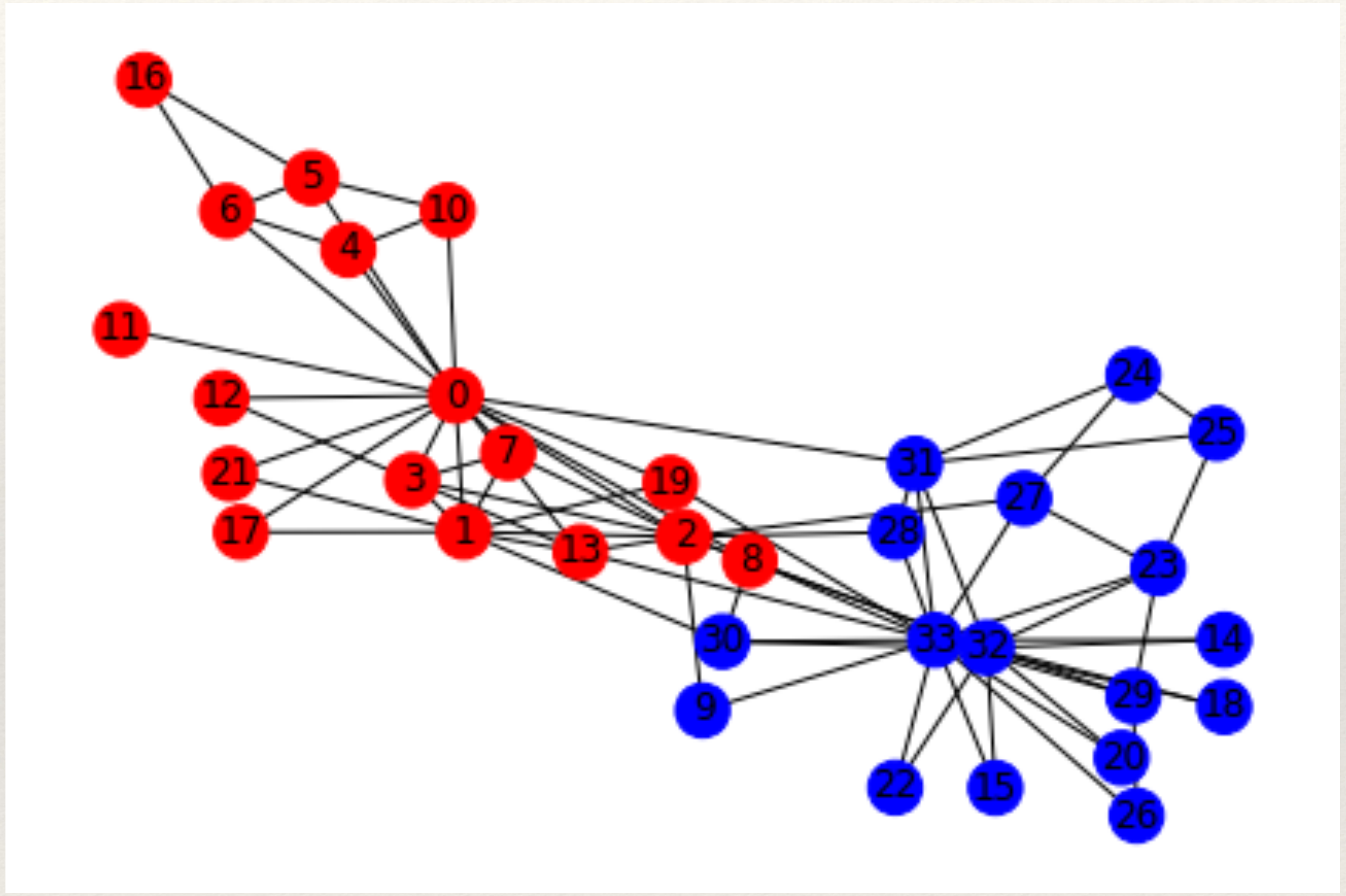
Some similar results can be obtained using, for instance, the modularity matrix B

$$y^T B y = \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2L} \right) y_i y_j$$

In this case we are maximizing the cost function hence we look for the largest eigenvalues of B.

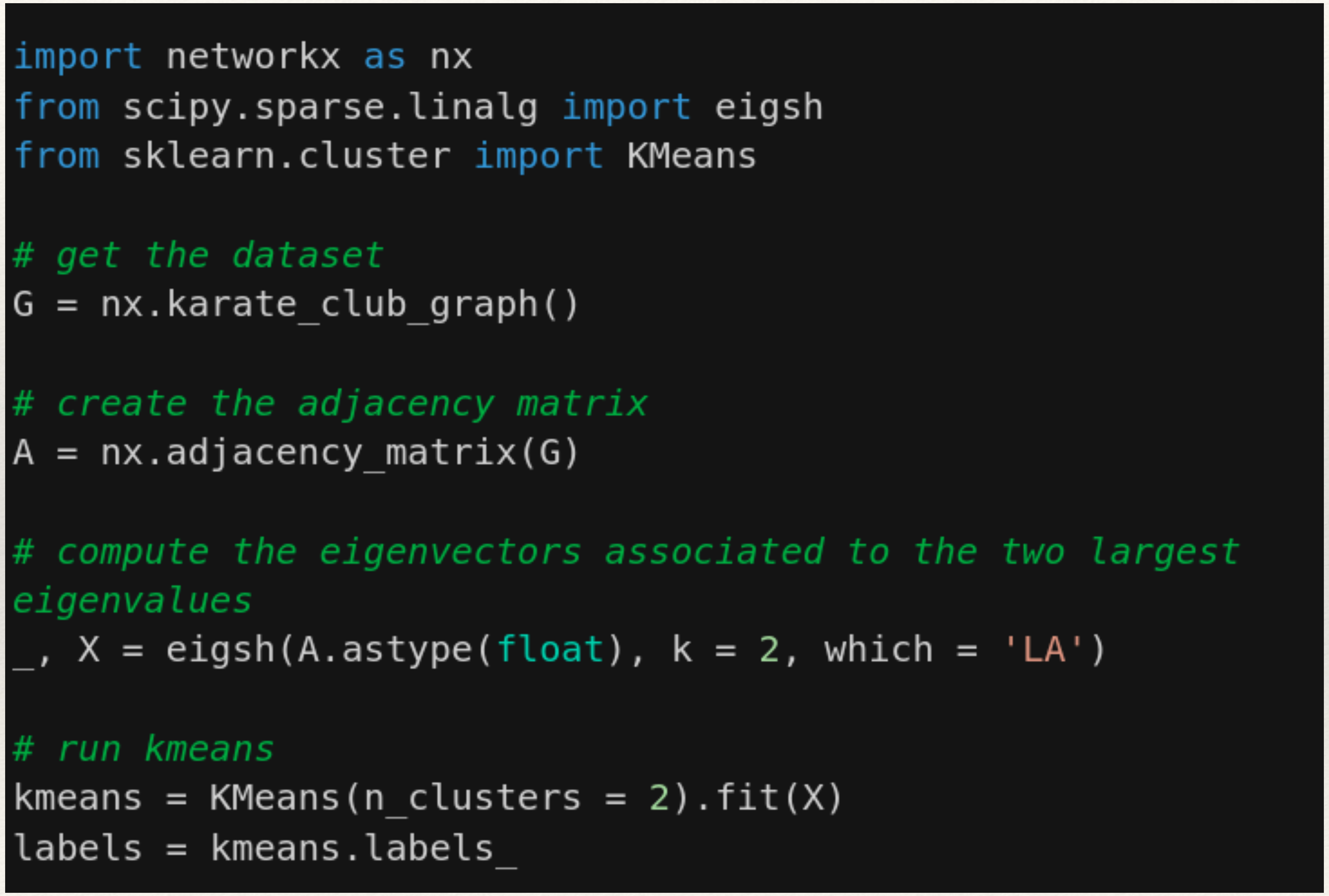
As a rule of thumb: n_eigenvalues = n_communities

Example



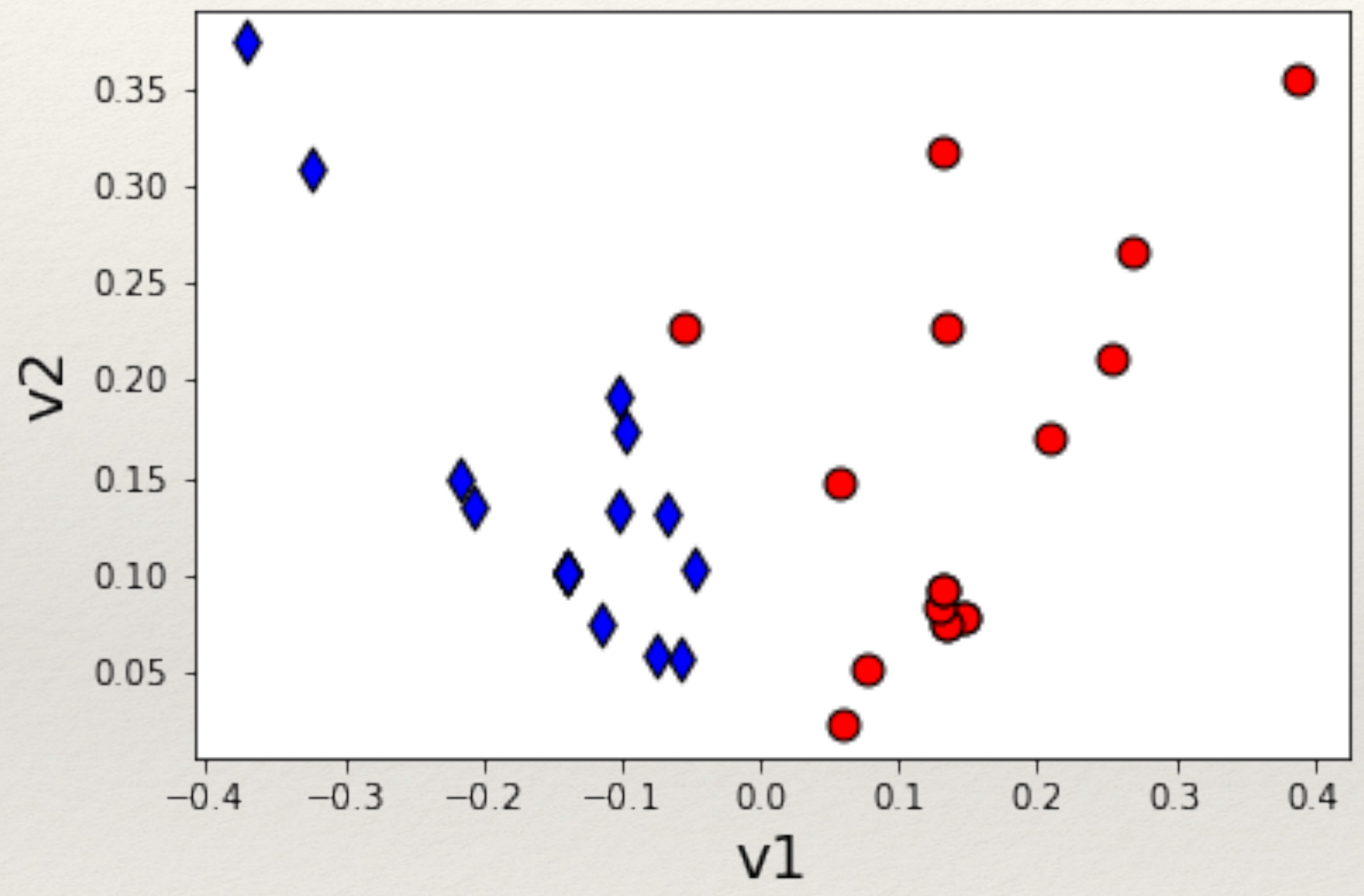
The karate club with ground truth annotated labels

Example

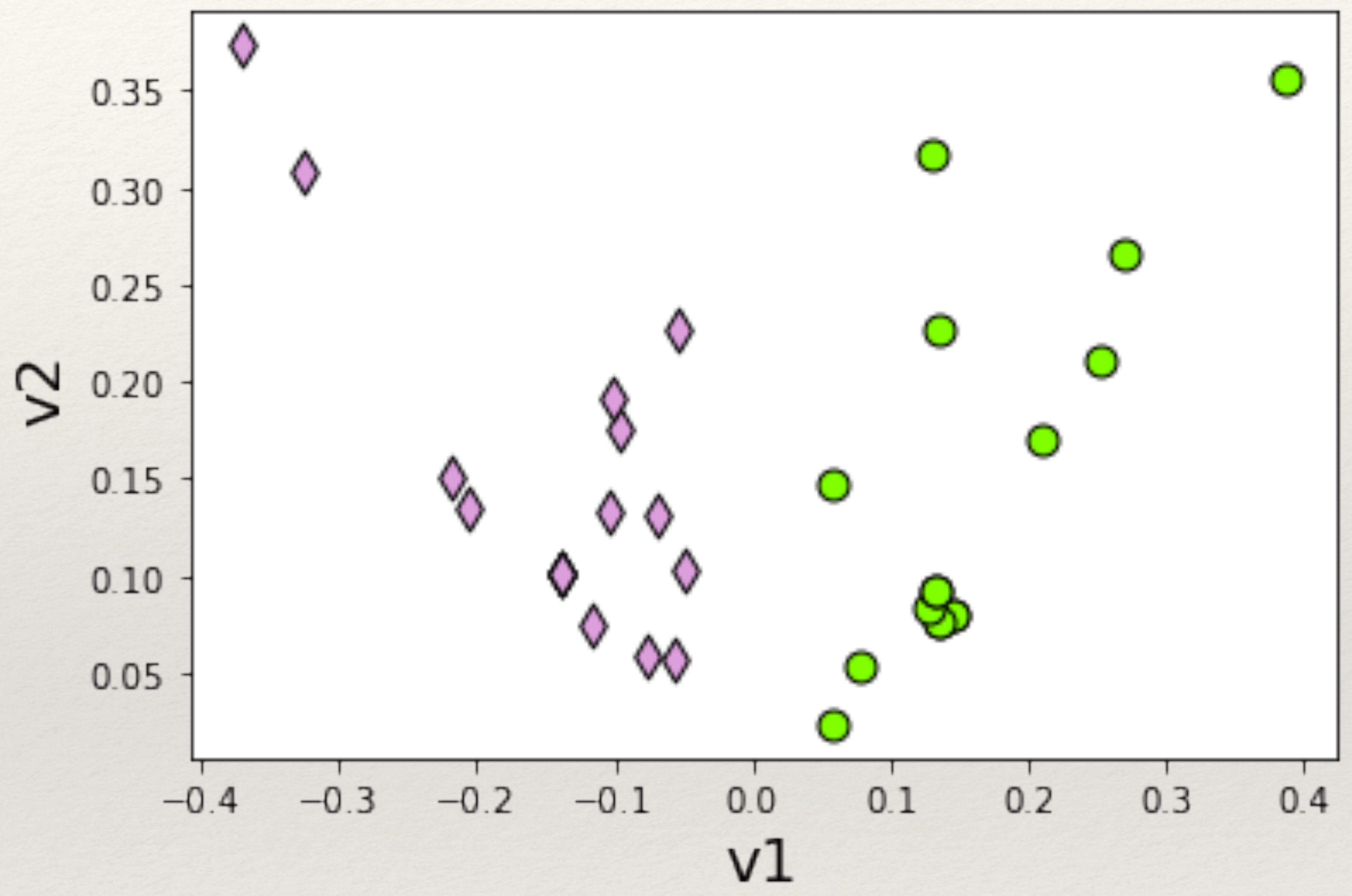


```python
import networkx as nx
from scipy.sparse.linalg import eigsh
from sklearn.cluster import KMeans

# get the dataset
G = nx.karate_club_graph()

# create the adjacency matrix
A = nx.adjacency_matrix(G)

# compute the eigenvectors associated to the two largest
eigenvalues
_, X = eigsh(A.astype(float), k = 2, which = 'LA')

# run kmeans
kmeans = KMeans(n_clusters = 2).fit(X)
labels = kmeans.labels_
```

Example



Ground truth coloring

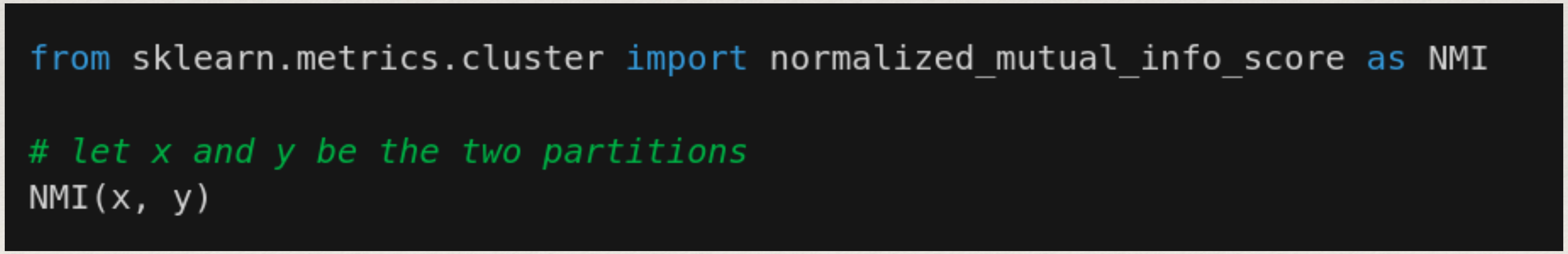


Inferred coloring

Excursus

In rare occasions we have benchmark graphs with annotated community structure. These are very useful to evaluate our algorithm. But how can we quantify the goodness of our algorithm?

With the **<u>normalized mutual information</u>**! This is a score between 0 (one partition does not bring any information on the other) and 1 (the two partitions are identical). We can implement it as follows



```python
from sklearn.metrics.cluster import normalized_mutual_info_score as NMI

# let x and y be the two partitions
NMI(x, y)
```

In the example of the slide before we have an agreement between the two coloring strategies with NMI = 0.84

Spectral clustering and the number of communities

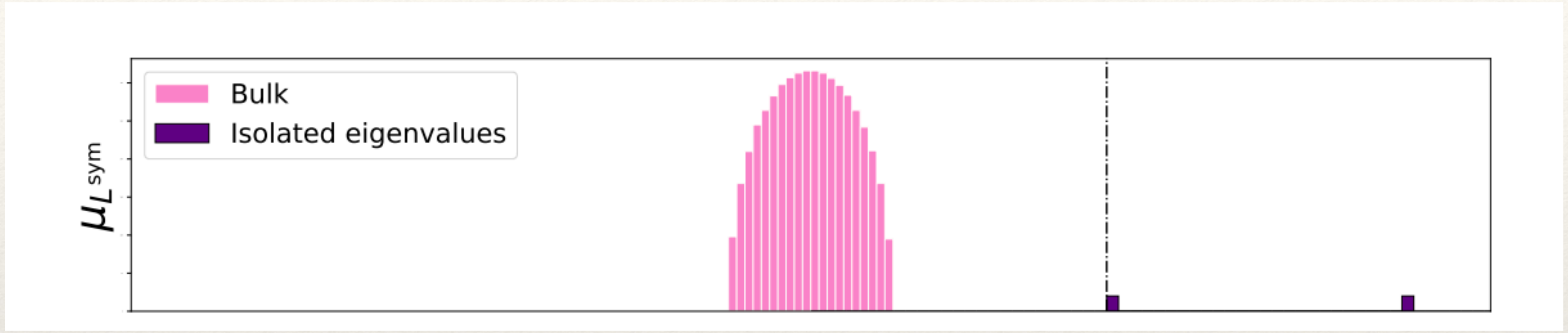
The output of spectral clustering is an embedding that we feed to k-means. Ideally, at that stage we should already know that is the number of communities in our network.
Spectral methods are among the most reliable and well grounded methods to estimate the number of communities in a network.

We rely on this definition:

Isolated eigenvalue: the distance from the closest eigenvalue is independent from the graph size
Bulk eigenvalue: the distance from the closest eigenvalue vanishes with the graph size

The estimation of the number of communities is simply the number of isolated eigenvalues the matrix M has. Of course, the choice of M is important, but let us make some examples before.

Spectral clustering and the number of communities



Histogram of the eigenvalues (spectrum) of the random walk Laplacian matrix on a graph with two communities. We can clearly see two isolated eigenvalues (the top two) that should be used to perform community detection.

Problem: this definition applies well for large graphs but not for small. Computing the whole spectrum for large graphs is inconvenient but smart and fast solutions exist to compute the number of communities from the matrix eigenvalues

The isolated eigenvalues

Why the number of isolated eigenvalues are equal to the number of communities?

$$M = Q + \Omega$$

M: matrix used to do spectral clustering

Q: expectation of M. This is a matrix of rank q (the number of communities) that contains all the information on the graph structure. This gives rise to the isolated eigenvalues of M

Omega: this is the noisy part of the spectrum and generates the bulk eigenvalues (useless)

Pros and cons of spectral clustering

Pros:

- This type of algorithms does not require any iterative process and eigenvectors can be computed <u>exactly</u>
- Several choices of M exist and they are connected to both optimization and Bayesian approaches, providing fast and good approximations.
- They have a very solid theoretical background and are well understood
- They provide ways to estimate the number of communities
- They are model-free

<u>In the end they represent something the graph is telling about itself!</u>

Pros and cons of spectral clustering

Cons:

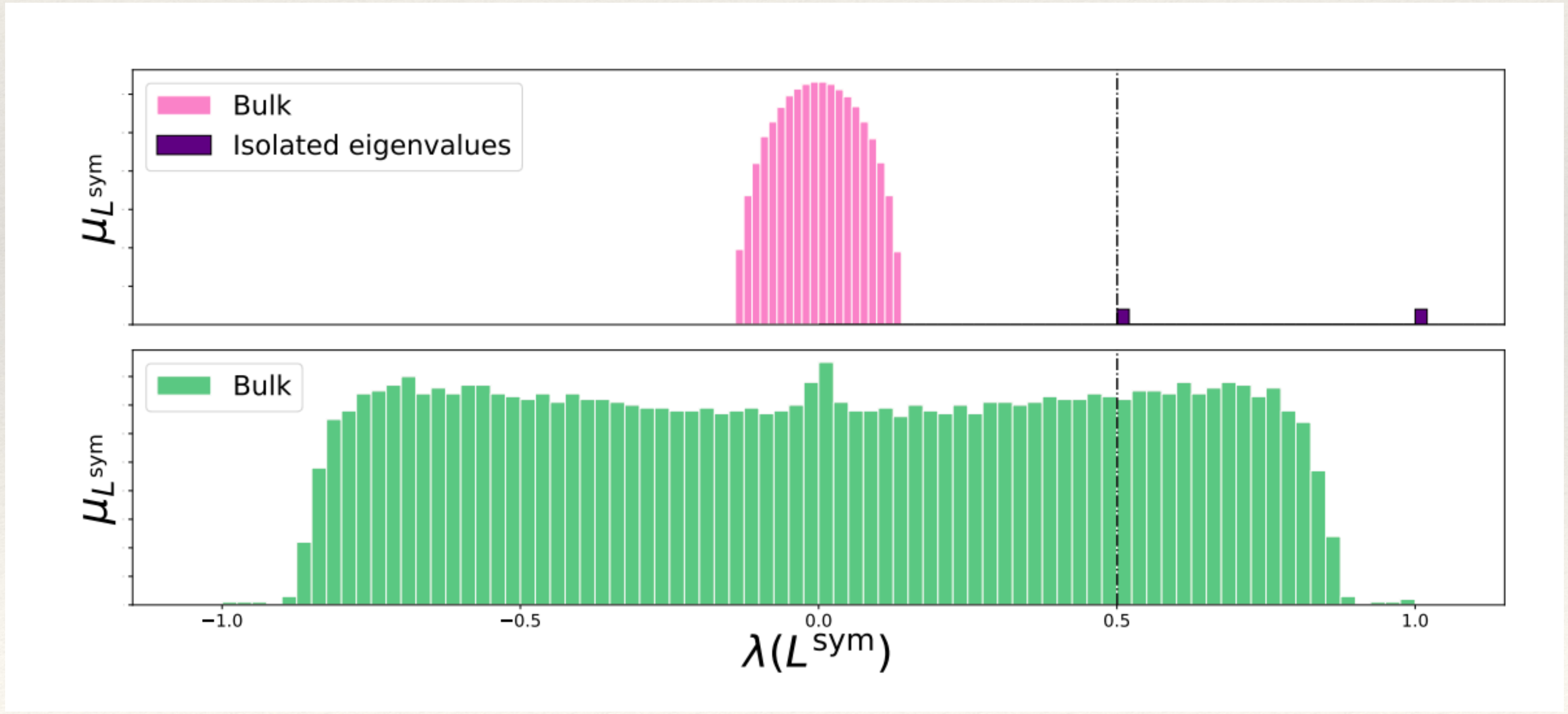
- The complexity scales with q^2 and they are unsuited for graphs with a lot of communities.
- They do not provide a hierarchical solution
- They hardly are optimal: there is typical some (slower) algorithm that performs better
- Most of the known results are studied for synthetic graphs. On real-world graphs things are still well defined, but more complicated
- You do not output a node partition but an embedding, hence you need kmeans.

Sparsity

For long, sparsity (few edges compared to the number of possible edges) has known to be the Achille's hill of spectral clustering. The reason is that the bulk on sparse graphs "swallows" the informative eigenvalues and they become no-longer informative, as shown in the following plot

dense regime

sparse regime



Sparsity

Since real-world networks are known to be sparse, this has long been considered a big problem of spectral clustering. Nonetheless, recent advances have shown that different matrices M can still accomplish high performance community detection in sparse graphs.

One example is the regularized random walk Laplacian matrix

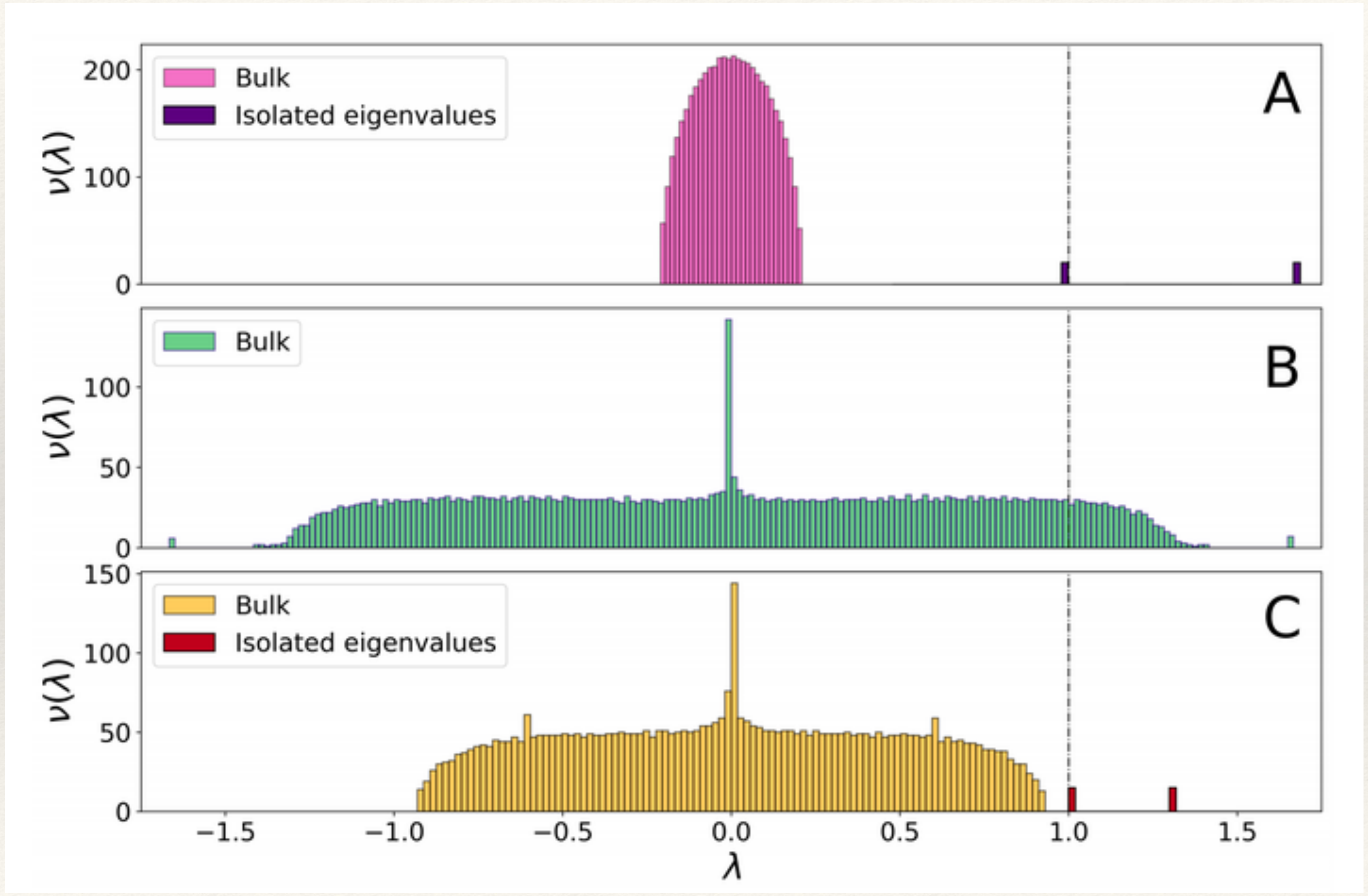
$$\begin{aligned} L_\tau &= D_\tau^{-1} A \\ D_\tau &= D + \tau I_n \\ \tau &= \frac{\langle k^2 \rangle}{\langle k \rangle} - 1 \end{aligned}$$

Sparsity

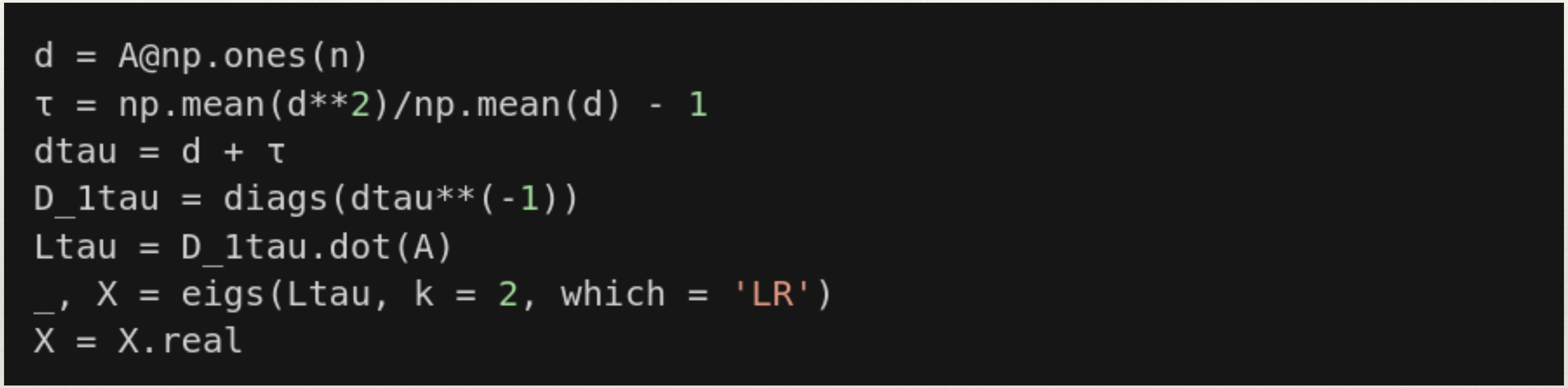
Random walk Laplacian: dense regime

Random walk Laplacian: sparse regime

Regularized random walk Laplacian: sparse regime



A useful algorithm

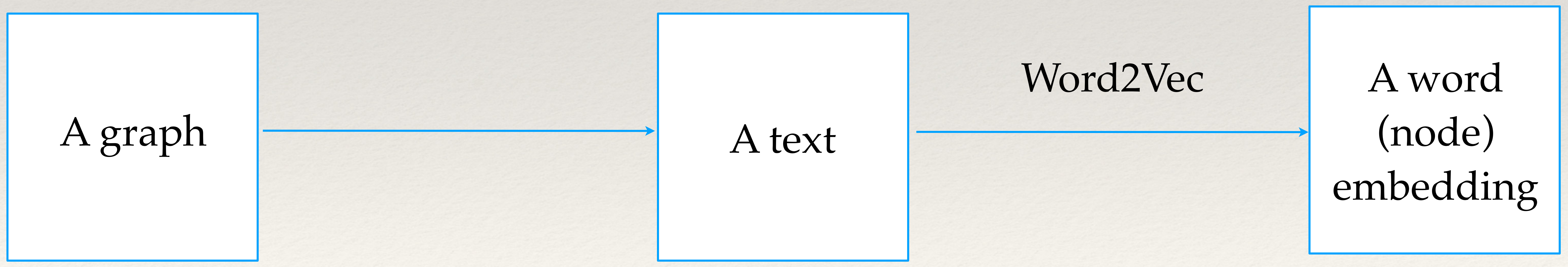


```python
d = A@np.ones(n)
τ = np.mean(d**2)/np.mean(d) - 1
dtau = d + τ
D_1tau = diags(dtau**(-1))
Ltau = D_1tau.dot(A)
_, X = eigs(Ltau, k = 2, which = 'LR')
X = X.real
```

DeepWalk

DeepWalk

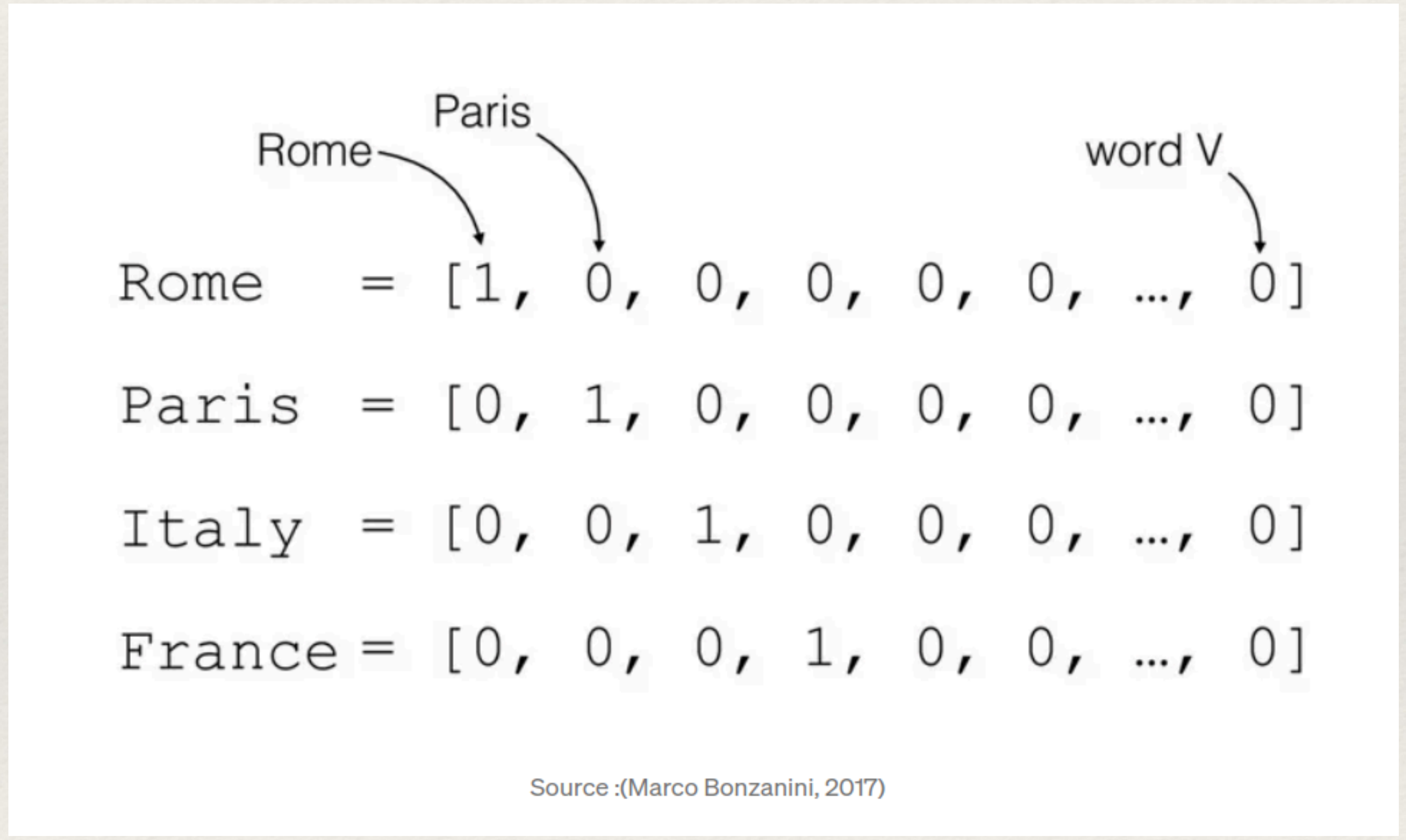
The DeepWalk algorithm is an alternative method to produce node embeddings of a graph. It builds upon another popular algorithm called *Word2Vec* that was created to learn word distributed representations (or embeddings).



Word2Vec

Problem: how to mathematically encode words?

Solution 1: we perform a "one-hot-encoding" setting to 1 the position the word has in the dictionary



```
Rome   = [1, 0, 0, 0, 0, 0, ..., 0]
Paris  = [0, 1, 0, 0, 0, 0, ..., 0]
Italy  = [0, 0, 1, 0, 0, 0, ..., 0]
France = [0, 0, 0, 1, 0, 0, ..., 0]
```

But: the representation is too sparse, the size of the embedding too large and it does not convey any semantic similarity

Word2Vec

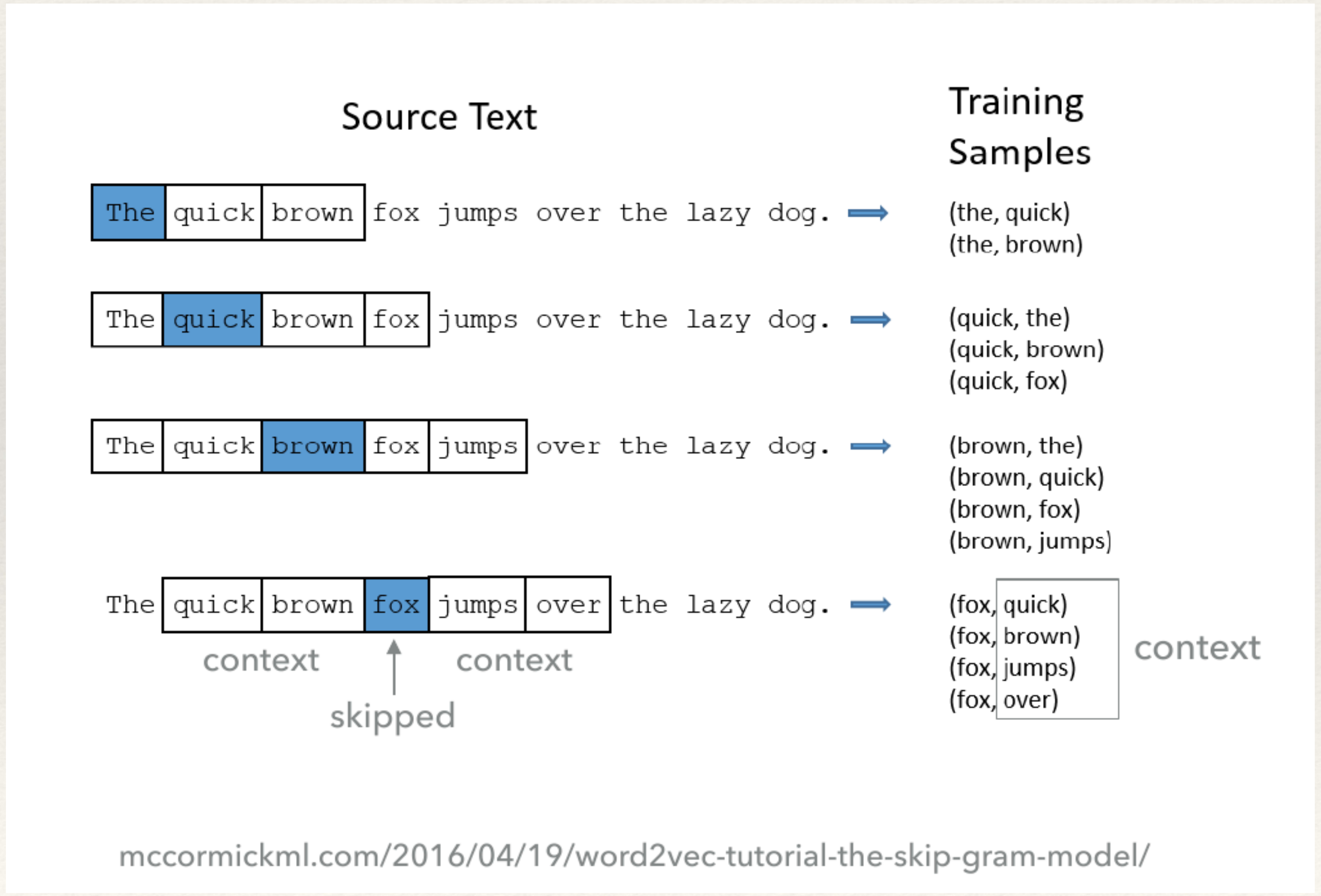
Problem: how to mathematically encode words?

Solution 2: we obtain a distributed word representation based on their context
"You shall know a word by the company it keeps" (John Rupert Firth)

Basic idea: for each word, define a set of context words in the text and train an algorithm so to obtain similar representations for words that are in similar contexts.

Skip-Gram model

Given a window size, all words within a distance from a central word are said to be in its context. The central word is "skipped"


Source Text
Training Samples
The quick brown fox jumps over the lazy dog.
(the, quick)
(the, brown)
The quick brown fox jumps over the lazy dog.
(quick, the)
(quick, brown)
(quick, fox)
The quick brown fox jumps over the lazy dog.
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)
The quick brown fox jumps over the lazy dog.
context
skipped
context
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)
context
mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Word2Vec

1. We define an input embedding x and an output (context) embedding y for each word of the vocabulary.

$$X, Y \in \mathbb{R}^{n \times d}$$

2. For each word in the text we obtain its context words and define an objective function that aligns the input embedding of that word (i) with the output embedding of the context words (k). Recall the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$c_1 = \log \sigma(x_i^T y_k)$$

Word2Vec

3. Now, draw a number of negative samples at random from the dictionary and define an objective function that prevents the embeddings alignment. This approach recalls the one of the modularity in which we maximize over the actual realization and minimize over the randomization

$$c_2 = \sum_{t=1}^{q} \mathbb{E}_{k \sim P_{\text{neg}}} \left[\log \sigma(-x_i^T y_k) \right]$$

Word2Vec

4. Optimize over all the words in text using stochastic gradient descent

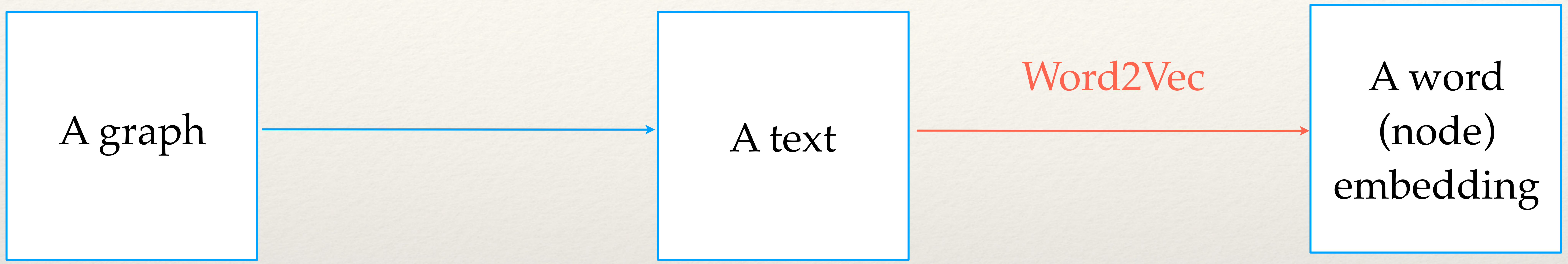
$$\mathcal{T} : \text{ text corpus}$$

$$\mathcal{C}_i : \text{ context words of i}$$

$$\mathcal{L}(X,Y) = \sum_{i \in \mathcal{T}} \left[\sum_{j \in \mathcal{C}_i} \log \sigma(x_i^T y_j) + \sum_{t=1}^{q} \mathbb{E} \left[\log \sigma(-x_i^T y_k) \right] \right]$$

5. Use the embedding X to represent words

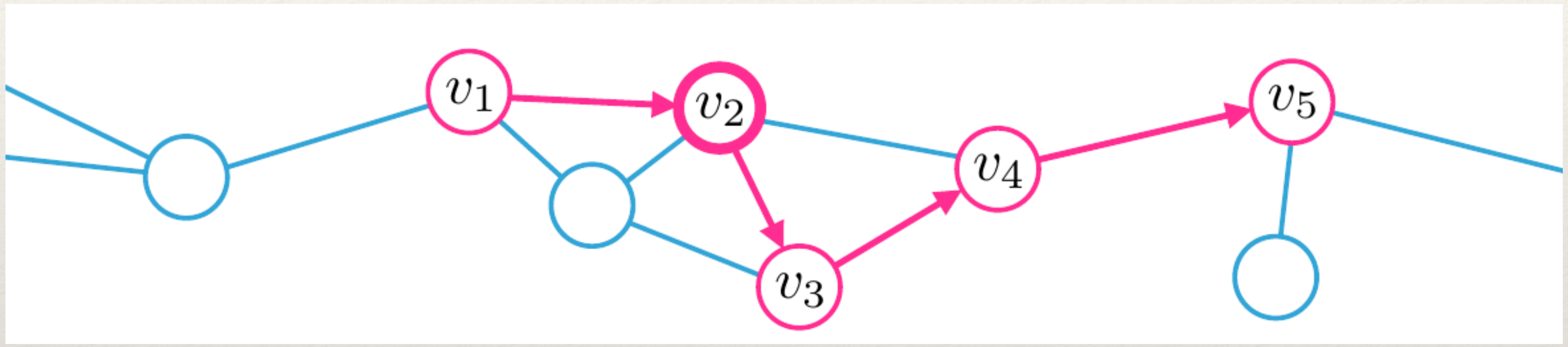
Back to graphs



How do we generate a text from a graph? We consider our neighbors as the context we live in.

DeepWalk

1. Create a several random walks on the graph. Each of them is a sentence of the text

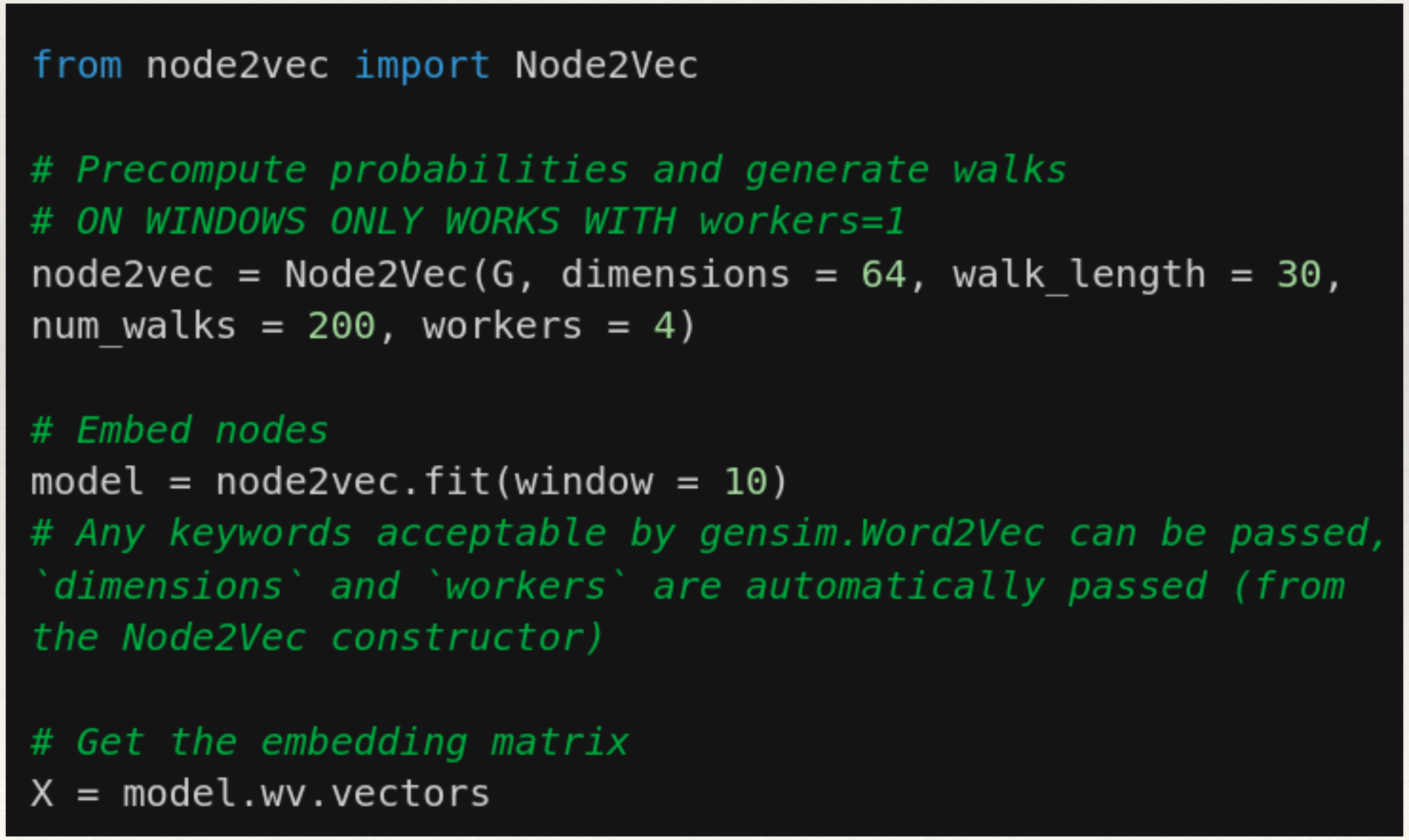


2. Given the "text" use Word2Vec to obtain the embedding of the words (aka the nodes)

An alternative version of DeepWalk exists and is called Node2Vec. In this case we have two parameters p, q that allow us to performed biased random walks on the network. For $p = q = 1$ Node2Vec = DeepWalk

DeepWalk in Python

At this repository you can find a Python implementation of this algorithm:
https://github.com/eliorc/node2vec



```python
from node2vec import Node2Vec

# Precompute probabilities and generate walks
# ON WINDOWS ONLY WORKS WITH workers=1
node2vec = Node2Vec(G, dimensions = 64, walk_length = 30,
num_walks = 200, workers = 4)

# Embed nodes
model = node2vec.fit(window = 10)
# Any keywords acceptable by gensim.Word2Vec can be passed,
`dimensions` and `workers` are automatically passed (from
the Node2Vec constructor)

# Get the embedding matrix
X = model.wv.vectors
```

DeepWalk summary

Maps the graph to a text and exploits a word embedding algorithm. Once the embedding has been obtained one can do several things, among which, community detection by simply performing clustering on the embedded space.

Pros:

- it has a strong connection to variational Bayesian inference and, in general, it does not lead to overfitting
- it does not have major convergence issues on real graphs
- it highly parallelizable.
- The complexity to produce the embedding is independent of the number of communities and it can be used for very large graphs

DeepWalk summary

Cons:

- The number of communities must not be specified before, so it has to be figured out from the embedding
- Higher embedding dimensions generally give better results but at a higher computational cost
- Even if it is a "competing" algorithm, it is not optimal in terms of performance

Conclusion

Conclusion

Community detection is the "simple" task of partitioning the network into groups. The core problem is, however, that no simple definition can be provided and several different angles can be taken. We focused on five types of approaches:

1. Optimization-based
2. Bayesian
3. Hierarchical (distance-based)
4. Spectral
5. DeepWalk

Each of these methods has its strength and weaknesses of which one must be aware. Typically the task of partitioning a network into groups has no unique right answer, but to best understand the results one might want to compare the output of different algorithms to make sense of the network structure.