XNode2Vec: Alternative Clustering Method Based on Node2Vec Algorithm

S. Bianchi

- Department of Physics and Astronomy, Bologna -

Abstract

Data classification has been a growing topic during the last decade and it is still an open issue with an endless number of applications in any field. There are dozens of clustering algorithms available based on very different techniques: ones like ***DBSCAN*** *are based on the elementary concept of density-reachability, while others like* ***Spectral Clustering*** *perform a dimensionality reduction of the data space, so that it becomes easier to cluster the transformed data with other methods.*

In this paper it's presented an additional clustering technique, called ***XNode2Vec****, which is based on the newest* ***node2vec*** *node classification method. The discussion on XNode2 Vec will cover its general functioning, giving a detailed description of its fundamental methods.*

Contents

1. Introduction

Node2vec is a recent node classification algorithm[1], strongly inspired by *DeepWalk*. The main difference between the two methods lies in *node2vec* possibility to bias the random walk, meaning that you can – statistically – control if the walk remains around the initial node or if it explores the outskirt of the network. With a bit of experience, a user can obtain very interesting results by adjusting the bias parameters of the walk p and q. The parameters p and q are namely the *"in-out parameter"* and *"return parameter"*. Moreover, if $p > max(1,q)$ the walk is less likely to sample an already visited node; the other way around produces the opposite effect. Regarding the other parameter, if $q > 1$ the walk will tend to visit nodes closer to the starting one. In addition, we can recover the exact results of *DeepWalk* by using $p = q = 1$.

Now we introduce **XNode2Vec** data clustering algorithm; since it is based on *node2vec* it must be applied on networks in order to work. The strength *xnode2vec* is however its generality, because its main feature is to generate a ready-to-use network starting from basically any type of dataset, for example *boolean*, *word-like* and *numeric*. Furthermore, there is no limit to the *dimensionality* of the dataset used, meaning that we can analyze data that lies in a N-dimensional space, without specific restrictions on the space topology[2].

The core idea is similar to *DBSCAN*, so the classification is obtained again using a density-reachability procedure, but the flexibility provided by *node2vec* and by the generality of a network makes this algorithm worthy of interest.

2. XNode2Vec Overview

The following section aims to describe in details the functioning of *xnode2vec* library and how to use it.

2.1. Fundamental Considerations

Let's start by saying that crucial feature of *node2vec* is the ability to "translate" the links in a network into a **Word2Vec** object[3]. Once the biased random walk has been made, a *word2vec* object is created, which is basically a vector of numbers that embeds the properties of the *items* encountered during the walk. This procedure is generally called *Word Embedding*.

All the further analysis are performed exclusively on these *word2vec* vectors, that are provided with many useful tools to work with.

Now, the native application of *word2vec* was – as the name suggests – on *words* and on their logical functions in a generic sentence; an extremely simplified example is shown in *Figure* 1.

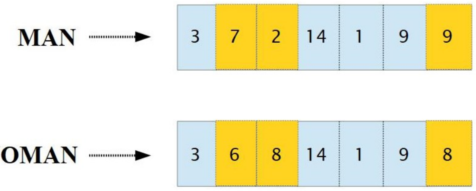

Figure 1: *Simplified scheme of the basic functioning of* **word2vec** *word embedding procedure.*

As we can see, the two words "**man**" and "**woman**" in this case are similar if we consider the two vectors describing them; most of the numbers are in fact exactly the same, while the others differ by small amounts. The peculiarity of *word2vec* is the following: the two words will be embedded by *different* vectors if we change the "logic" behind the analysis – to anticipate, this is controlled by the ***context*** parameter. In particular, the two vectors might be even more similar when examined in a context that does not involve gender differences:

- *A ... goes to work.*

but they can also be much more different in a context like:

- *A ... is pregnant.*

This already suggests a stunning generality of *xnode2vec* algorithm, also regarding the types of "items" that can be manipulated. In fact, we will see that the most challenging part was to identify the best method with which to assign **links** and their respective **weights** in the network. It's therefore important to specify that – up to now – the links distribution methods available in *xnode2vec* library are good for most types of data, but may not be the best ones, or even not good for other specific objects.

2.2. Word Embedding

The last thing it is useful to know in order to fully understand the algorithm is how a word embedding is performed. The two most famous techniques used to generate *numerical vectors* from *words* are: **CBOW** (Continuous Bag of Words) and **Skip Gram**. *CBOW* aims to predict the *target word* from a given context, while *Skip Gram* does exactly the opposite, meaning that it predicts the *context* of a target word. There are of course downsides for both techniques and there isn't always a better choice; however it's safe to say that *Skip Gram* is usually more accurate for large-scale datasets, even if it is slower to train. Node2Vec uses a Skip Gram based algorithm indeed, where the *context* of a node is its **neighborhood** and the *context window* is the set of close neighborhoods. This also explains why *Xnode2vec* is a density-reachability clusterization algorithm, where the density is given by the magnitude of the link weight.

The *word2vec* embedded vector is therefore created by *node2vec*, that takes in account the nodes visited by the random walk, together with the usage of the *Skip Gram* technique. The similarity between two nodes is simply evaluated using the *cosine similarity*, once the vectors has been created:

$$\cos(\theta) = \frac{v_1 \cdot v_2}{\|v_1\| \, \|v_2\|} \, .$$

2.3. Presentation of the Algorithm

The library *Xnode2vec* has been written entirely in python for many reasons, first because of its easy-to-use coding experience, then for the large community behind and for the huge amount of libraries available – and well optimized – , among which we also find *node2vec*. Now, the original *node2vec* package written by A. Grover was poorly optimized; any analysis required a massive amount of computational resources, that was not compatible with any real life application. For this reason, the package that contain the random walk algorithm was replaced by a significantly faster version called **FastNode2Vec**, written by L. Abraham[4].

The idea behind the algorithm is straightforward:

1. Assuring that the dataset is written in the required format.
2. Apply the proper transformation on the dataset that generates an *edgelist*.
3. Build a **networkx** graph from the created edgelist.
4. Perform the node classification analysis on the generated network.

As anticipated, *Xnode2vec* works only with networks, so every dataset must be converted to a network that should *embed* the crucial property of the dataset itself. This of course does not exclude the possibility to analyze even a generic network that is not linked to a dataset. The library used to manage the networks is the very well known **Networkx**, which is extremely user-friendly and without any particular limitation in terms of network generation[5].

Let's reduce the generality of our discussion and let's take a more concrete example of dataset, in particular the type that suits the algorithm the most.

Let Z be a dataset defined in an N-dimensional *euclidean* space, so that the i-th point in this dataset is expressed as

$$\boldsymbol{x}_i = (x_0, x_1, \dots, x_N) \ .$$

The metric of this space is euclidean, thus the distance is calculated as

$$\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \equiv \sqrt{(\boldsymbol{x}_i - \boldsymbol{x}_j) \cdot (\boldsymbol{x}_i - \boldsymbol{x}_j)} \ . \quad (1)$$

Xnode2vec allows the creation of an **edgelist** of a **completely connected** network, whose *nodes* are the *points* of the dataset, while the *links* are given as

$$weight_{i,j} \equiv \exp(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|}{|k|}) \ , \quad (2)$$

where k is a "stretch" parameter which is usually unitary. The reason why the weights are given with (2) is because closer points in space – small distances – should have larger weights in the network; in addition, this structure enhances the gaps between the points and removes issues with coincident ones – which have zero distance. Clearly, the weights values range between $[0, 1]$. Once the edgelist is created, we convert it to a **networkx.Graph** object and we apply the clusterization algorithm on it.

Finally, let's show the objects **types** and **formats** required by the algorithm in order to work; in the next sections we will see how to use them.

- **Dataset**: There are no limitations to the dimensionality of the dataset, neither in the number of entries nor the number of features for each entry. The required structure is

$$\begin{aligned} \text{np.array}(\ [[&x_{0,0}, \dots, x_{0,n}], \\ [&x_{1,0}, \dots, x_{1,n}], \\ &\dots \\ [&x_{m,0}, \dots, x_{m,n}]]) \end{aligned}$$

  where m is the number of entries and n is the number of features.

- **Edge List**: The edge list is required to generate the *networkx.Graph* object and has the form

$$\begin{gathered} [(node_0, node_0, weight), \\ (node_0, node_1, weight), \\ \dots \\ (node_m, node_m, weight)] \end{gathered}$$

  so it's a list of *tuples* structured as $(starting\_node, arriving\_node, weight)$.

- **DataFrame**: This is a *Pandas.DataFrame* object that is used to manipulate the network. The reasons behind the usage of *Pandas* dataframes are plenty, in particular because of their coding optimization and because of the amount of tools that they provide. The dataframe must be structured as

$$\begin{aligned} &pandas.DataFrame(\\ &\quad np.array([[0, 1, 3.7], \dots, [m, m, 12]]) \ , \\ &\quad columns = ['node1', 'node2', 'weight']) \end{aligned}$$

  where again m represents the last node.

These are essentially the only objects that we need in order to use *Xnode2vec* package that have a specific structure required.

2.4. Xnode2vec Generalities

We said that *node2vec* manages to assign a "similarity" value to every nodes in the network with respect to a target one; this value is actually the discriminant with which we populate the clusters. In fact, the clusters are generated by picking only the nodes with a *similarity* higher than a given **threshold**, which is of course a tunable parameter. It's important to point out that the user doesn't have to specify neither the number of clusters expected nor the minimum number of entries per cluster, which is very convenient.

Now, let's see how actually *Xnode2vec* is intended to be used in the most simple way, step by step.

Let Z be a dataset with m points in an n dimensional space. First, as we said, we have to generate the edgelist of the network; the way to do this is to call the function **complete_edgelist()** – which generates a *Pandas.DataFrame* object – and then convert this DataFrame to *networkx* required format by using **generate_edgelist()** :

```python
import Xnode2vec as xn2v

df = xn2v.complete_edgelist(Z)
el = xn2v.generate_edgelist(df)
G = nx.Graph()
G.add_weighted_edges_from(el)
```

In this way we created a *networkx* graph whose weights are assigned by (2). The network will have m *nodes* and the number of links is

$$links = \sum_{n=0}^{m} n \, , \tag{3}$$

since it is completely **connected** and **undirected** network. The cluster classification analysis is then carried out by the function **clusters_detection()**, which is an articulated function containing others that we won't see at the moment. Now, it is not mandatory to start from a dataset; one can also start form a generic *Pandas* DataFrame edgelist, or even directly from a *networkx* Graph. The only requirement for the DataFrame edgelist is to use the *structure* demanded by the *Xnode2vec*. Moreover, it is *strongly* suggested to use *strings* as nodes labels, since the algorithm has been designed to work preferably with them; however, there are many type-controls running, so it shouldn't be a problem.

In order to use it, we convert the *networkx* graph object to a *fastnode2vec.Graph* one, calling **nx_to_Graph()**:

```python
graph = xn2v.nx_to_Graph(G)
nodes_families, unlabeled_nodes = xn2v.
    clusters_detection(...)
```

The function **clusters_detection()** returns two vectors: *nodes_families* and *unlabeled_nodes*. As the names suggest, the two are *lists* of *numpy.array* objects containing the clustered and the unclustered nodes. Inside **clusters_detection()** the user must insert specific parameters; the function gives many analysis options, but in general the most common set of parameters is:

— MANDATORY —

- **G**: Networkx Graph object. It embeds the properties of the dataset in its links.
- **dim**: Dimension of the nodes embeddings. The larger it is, the more differences we can spot between different embeddings.
- **context**: Number of neighbor nodes before and after the given node to be included in the analysis as "context" nodes (related to the starting one).
- **walk_length**: Length of the biased random walk, which starts from the specified node. Its behavior is largely influenced by the values of the parameters p and q.

— OPTIONAL (Suggested) —

- **cluster_rigidity**: Value of the similarity threshold chosen as comparison for every entry in the clusters. The proper value can change significantly depending on the situation considered; also, the other parameters will affect the choice of *cluster_rigidity*, since they alter the computation of the similarities.
- **spacing**: Distance between two subsequent nodes analyzed by the algorithm. If a network has 100 nodes and we set $spacing = 5$, we will have 20 nodes – starting from the first one – from which the random walks begin.
- **graph**: Fastnode2vec.Graph object; this is useful to optimize the duration of the classification, since the algorithm doesn't have to read the graph every time.

- **dim_fraction**: Minimum fraction of already existing nodes in a cluster required to *expand* it. *Node2vec* similarity analysis gives a vector of similar nodes; this vector is then compared to the nodes already present in all the clusters. The cluster that has more nodes in common with this "similar nodes vector" is expanded with the remaining nodes. $dim\_fraction = 1$ means that the cluster can't be expanded.

- **picked**: Number of most similar nodes chosen from the classification, in order of similarity. Usually, it's a good habit to always pick all the nodes of the network, since the threshold should provide a more flexible and accurate result.

- **Epochs**: Number of random walks that start from the initial node. Clearly, more walks means better statistics, but the computational power required grows exponentially.

All these parameters should enter in **clusters_detection()** function. It's not trivial to understand how they affect the result, so it's really useful to play with them on toy models and on easy real life cases.

Once the analysis is concluded, the result is summarized as:

```
--------- Clusters Information ---------
- Number of Clusters: 5
- Total nodes: 1144
- Clustered nodes:  1125
- Number of unlabeled nodes: 19
- Nodes in cluster 1: 327
- Nodes in cluster 2: 401
- Nodes in cluster 3: 77
- Nodes in cluster 4: 182
- Nodes in cluster 5: 138
```

which are stored in *nodes_families* and *unlabeled_nodes* vectors.

Now, if we started from a network, then our classification is concluded, leaving space for a whatever further analysis on the classification results. However, if we had a **numerical dataset** – as in our example – then we should go back from the nodes to the points. This can be simply done by calling the function **recover_points()**:

```python
for i in range(len(nodes_families)):
    points_families.append(xn2v.recover_points(Z, G, nodes_families[i]))

points_unlabeled = xn2v.recover_points(Z, G, unlabeled_nodes)
```

where we recall Z to be the initial dataset. This function must be called for every cluster obtained from the analysis, so it can't read the whole list of vectors. In this way *points_families* and *points_unlabeled* are storing the clusters in a point-like format.

There are many other functions available in *Xnode2vec* that modify the initial dataset Z, or that improve the accuracy of the analysis, that we will see in the following section.

2.5. Data Manipulation and Network Adjustments

Up until now we introduced the generalities of *Xnode2vec* package, giving the basic tools to understand how to use and what is the idea behind its development. Now we can take a look at the secondary functions available, that usually turn out to be essential. Here there is the list of the functions that we are going to examine:

- **low_limit_network()**
- **labels_modifier()**
- **best_line_projection()**

The first is crucial: **low_limit_network()** can improve the analysis drastically, both in terms of performances and classification. This function allows the "suppression" of links with weight below a specific value, or even the removal of those links. The usage is pretty straightforward:

```python
delta = 0.05
G = xn2v.low_limit_network(G, delta, remove=True)
```

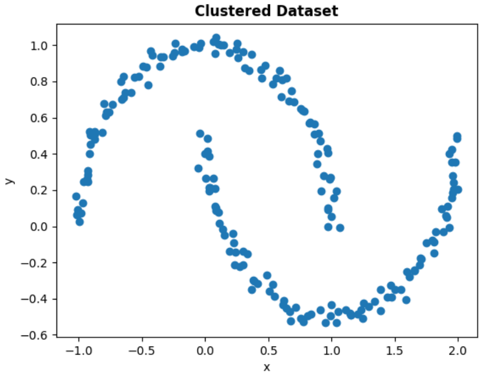

Figure 2: *Result of an XNode2Vec data classification*, $cluster\_rigidity = 0.65$, *unused low_limit_network()*

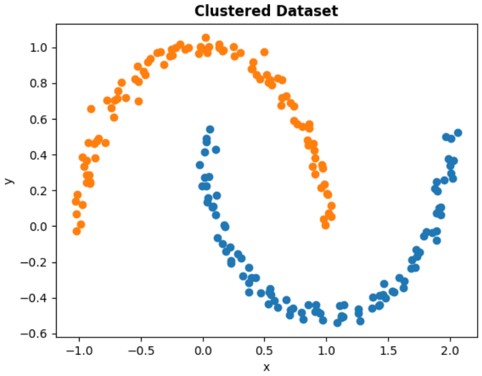

Figure 3: *Result of an XNode2Vec data classification*, $cluster\_rigidity = 0.65$, *used low_limit_network()*

where G is the network, *delta* is the coefficient that creates the threshold by multiplying the *maximum* weight in the network and *remove* specifies if the link should be removed or set to zero. Let's see some examples of how it behaves; we consider a simple dataset that is created using the well known *sklearn.datasets.make_moons*. The full code is reported in $Appendix$ A. The differences between the classification in $Figure$ 2 and 3 are astonishing. The result of the classification without using **low_limit_network()** is shown in $Figure$ 2 and as we can see it fails to detect any difference between the points. The outcome may be mitigated by adjusting the parameters, but still their sensibility is too much in order to be replicable in even slightly different situations. On the other hand, the classification supported with **low_limit_network()** delimits perfectly the two moons. In this case, there is no distinction between the *removal* or the *suppression* of the links; the only difference – negligible for this system – lies in a slightly major computational time required, since a network with more edges is heavier to handle. One evident manifestation of *Xnode2vec* flexibility is when we try to increase *cluster_rigidity* from 0.65 to 0.85; the outcome is shown in $Figure$ 4. This last classification displays a **stricter** cluster selection, that may also be the correct one from what we know, by only changing the value of a single parameter.

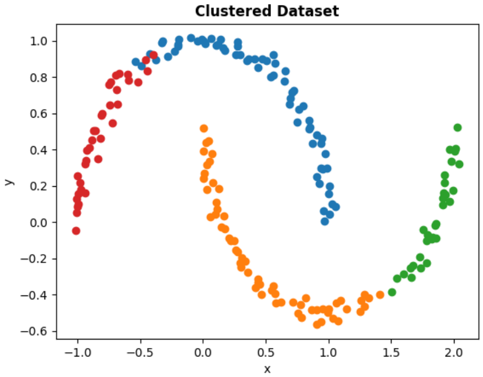

Figure 4: *Result of an XNode2Vec data classification*, $cluster\_rigidity = 0.85$, *used low_limit_network()*

Of course, if we consider a simple two dimensional dataset such as this one we can't expect many more different results than these three encountered. However, whenever we have more complex spaces and metrics it could be interesting to see what results are obtained, time permitting.

The second one is **labels_modifier()**. This function is very easy, but at the same time essential if we have to keep track of the nodes in the network. It can be use as follows:

```
new_indexes = ['node1',2,'noise','x'
    ,4,5,'main_hub',8,9,'end_point']
G = xn2v.labels_modifier(G, new_indexes)
```

where we modified the network labels into the ones specified in *new_indexes*. A situation where this could save a lot of time is this one: we have a generic dataset expressed as a *csv table* and we need to select **specific rows** of this table. If we simply apply the conversion of the *reduced* dataset to the *networkx.Graph* object, the default labels assigned to the nodes are $['1','2','3',\dots,'m']$. It's evident that in this way we are unable to recover the indexes of the correct rows. Normally the *reduced* dataset keeps track of the original indexes – *Pandas.DataFrame* does this –, therefore in order to recover the correct rows we only have to change the labels as:

```
reduced_dataframe = some_transformation(
    dataframe)
a = np.array(reduced_dataframe)
b = xn2v.complete_edgelist(a)
b = xn2v.generate_edgelist(b)
G = nx.Graph()
G.add_weighted_edges_from(b)
G = xn2v.labels_modifier(G, new_indexes)
```

where clearly all the vectors satisfy the conditions required in section 2.3. This exact procedure has been used for instance in the analysis of the *AML dataset*, that we will see later.

The last function is **best_line_projection()**. This is actually the beginning of a list of functions – not implemented yet – that **transform** the original dataset into a different one that will be analyzed. Of course any transformations must keep the order of the points of the original dataset, otherwise we will not be able to distinguish them anymore. As the name suggests, *best_line_projection()* performs a *best linear fit* and projects all the points of the dataset on this line. It doesn't alter the dimensionality of the dataset, in fact the line lies in a space with the same dimension as the points. The usage it's again very simple:

```
transf_dataset = xn2v.
    best_line_projection(dataset)
```

This function can be useful when we consider a blobs dataset, since it usually reduces the distance between the points and enhance the separation of the blobs. There are several transformations that could be very useful, for example a *spectral dimensionality reduction*, *non-linear transformations*, etc.. that may be implemented in the future.

3. Applications of the Algorithm

We now want to see how *Xnode2vec* performs when applied on generic datasets and even real life ones. We are interested in particular to see the response of the algorithm when the parameters are changed and what kind of **replicable** results we can obtain.

3.1. Generic Classifications

In this section we propose merely constructed datasets with no particular background, but that can potentially occur in a real life scenario. The datasets will be mostly created by using **sklearn.datasets**. The **moons** one has already been discussed in the previous section, with the full code proposed in A, so we are going to skip that one.

Another dataset that we want to analyze is a **blobs dataset**, meaning that we have points accumulations scattered on a 2-dimensional plane. We store the dataset into a variable, exactly in the form required in 2.3:

```
from sklearn.datasets import make_blobs
dataset, b = make_blobs(200, centers=4,
    random_state=42)
```

so we generated 200 points belonging to 4 different families. In addition, we fixed the random state to 42, so that we can reproduce the experiment at any time.

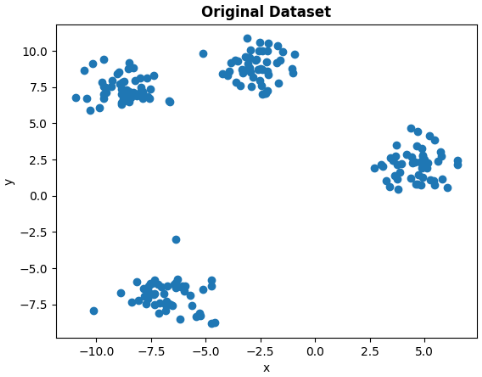

Figure 5: *Original blobs dataset created from sklearn.datasets.*

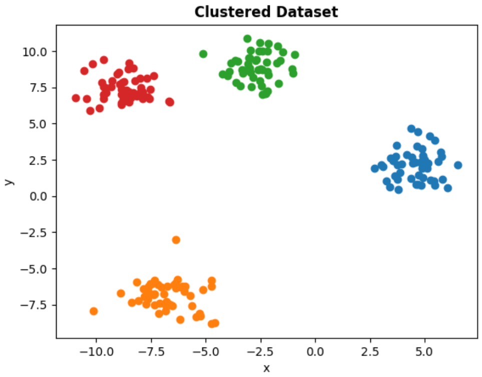

Figure 6: *Result of an XNode2Vec data classification on the blobs dataset.*

We expect the result to be always the same every time we perform the analysis, with at most little variations due to the randomness of the walk. We apply all the steps we saw earlier to generate our network that embeds the important features of the dataset:

```
df = xn2v.complete_edgelist(dataset)
df = xn2v.generate_edgelist(df)
G = nx.Graph()
G.add_weighted_edges_from(df)
```

Now we perform *Xnode2vec* analysis on the *networkx* graph.

```
graph = xn2v.nx_to_Graph(G, Weight = True)
clust, unclust = xn2v.clusters_detection(G, cluster_rigidity = 0.75, spacing = 5, dim_fraction = 1.1, Epochs=5, picked=a[:,0].size, dim=40, context=10, walk_length=20, graph=graph)
```

The visual results are shown in *Figure* 5 and 6. If we want to print out a generic overview of the classification we can call the method **summary_clusters()** as:

```
>>> summary_clusters(clust, unclust)

--------- Clusters Information ---------
- Number of Clusters: 4
- Total nodes: 200
- Clustered nodes:  198
- Number of unlabeled nodes: 2
- Nodes in cluster 1: 49
- Nodes in cluster 2: 50
- Nodes in cluster 3: 49
- Nodes in cluster 4: 50
```

that can be useful as a quick result checking. It's evident that the algorithm has recognized correctly all the four families, leaving only two nodes unlabeled. The values of the parameters are expressed in the code listed above and we may want to discuss them a little bit. First of all, the dataset has 200 nodes, therefore the function **complete_edgelist()** produces a network with 20100 edges, since (3) holds. In this case we don't have to suppress any of the edges, because the spatial distances between the points are definitely enough for the classification. By looking at the *known* dataset in *Figure* 5, we suspect that the families will mostly have the same amount of entries, therefore we can select a relatively small **walk_length** $\sim$ 20; also, we don't need a large statistics, since the families are fairly separated, thus we can set **Epochs** $\sim$ 5. Regarding the *embedding dimension* parameter, surely we don't need too many features to distinguish 200 points, so **dim** $\sim$ 40 is even too much, but higher values are also fine. Finally, **context** $\sim$ 15 is suggested by the fact that the nodes neighborhoods are not going to bee neither too small nor too large; smaller values may lead to worst results, as shown in *Figure* 7.

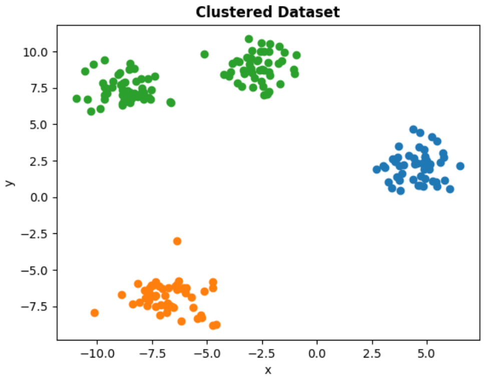

Figure 7: *Result of an XNode2Vec blobs classification, using* $\mathtt{context} = 10$*.*

For the same reason, higher values of **context** will still give good – and possibly better – results, but the computational time increases because of the more optimization required for the walk. If we further decrease the value of **context**, we will end up with only one cluster containing all the nodes.

Appendices

A. Moons Dataset

```
import Xnode2vec as xn2v
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

dataset, b = make_moons(200, noise=0.035)

df = xn2v.complete_edgelist(dataset) # Pandas edge list generation
df = xn2v.generate_edgelist(df) # Networkx edgelist format
G = nx.Graph()
G.add_weighted_edges_from(df) # Generate the network
G = xn2v.low_limit_network(G, 0.7, remove = True)
nodes_families, unlabeled_nodes = xn2v.clusters_detection(G, cluster_rigidity = 0.65, spacing = 5, dim_fraction = 1.1, Epochs=5, picked=G.number_of_nodes(), Weight=True, dim=20, context=20, walk_length=40)

# ================ Plot ================
points_families = []
points_unlabeled = []
for i in range(len(nodes_families)):
    points_families.append(xn2v.recover_points(dataset, G, nodes_families[i]))
points_unlabeled = xn2v.recover_points(dataset, G, unlabeled_nodes)

for i in range(0,len(nodes_families)):
    plt.scatter(points_families[i][:,0], points_families[i][:,1])
plt.xlabel(’x’)
plt.ylabel(’y’)
plt.title(’Clustered Dataset’, fontweight=’bold’)
plt.show()
```

References

[1] A. Grover, J. Leskovec. *"node2vec: Scalable Feature Learning for Networks"*. KDD '16, August 13 - 17, 2016.

[2] S. Bianchi. *"XNode2Vec - An Alternative Data Clustering Procedure"*. `https://github.com/Stefano314/Xnode2vec`, 2021.

[3] T. Mikolov, K. Chen, G. Corrado, J. Dean. *"Efficient Estimation of Word Representations in Vector Space"*. ICLR, 2013.

[4] L. Abraham. *"FastNode2Vec"*. Zenodo, `https://github.com/louisabraham/fastnode2vec`, 2020.

[5] Aric A. Hagberg, Daniel A. Schult, Pieter J. Swart, *"Exploring network structure, dynamics, and function using NetworkX"*. Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008.