Vatsal Follow

Jan 31 · 10 min read ★ · Listen

Node2Vec Explained

Explaining & Implementing the Node2Vec Paper in Python

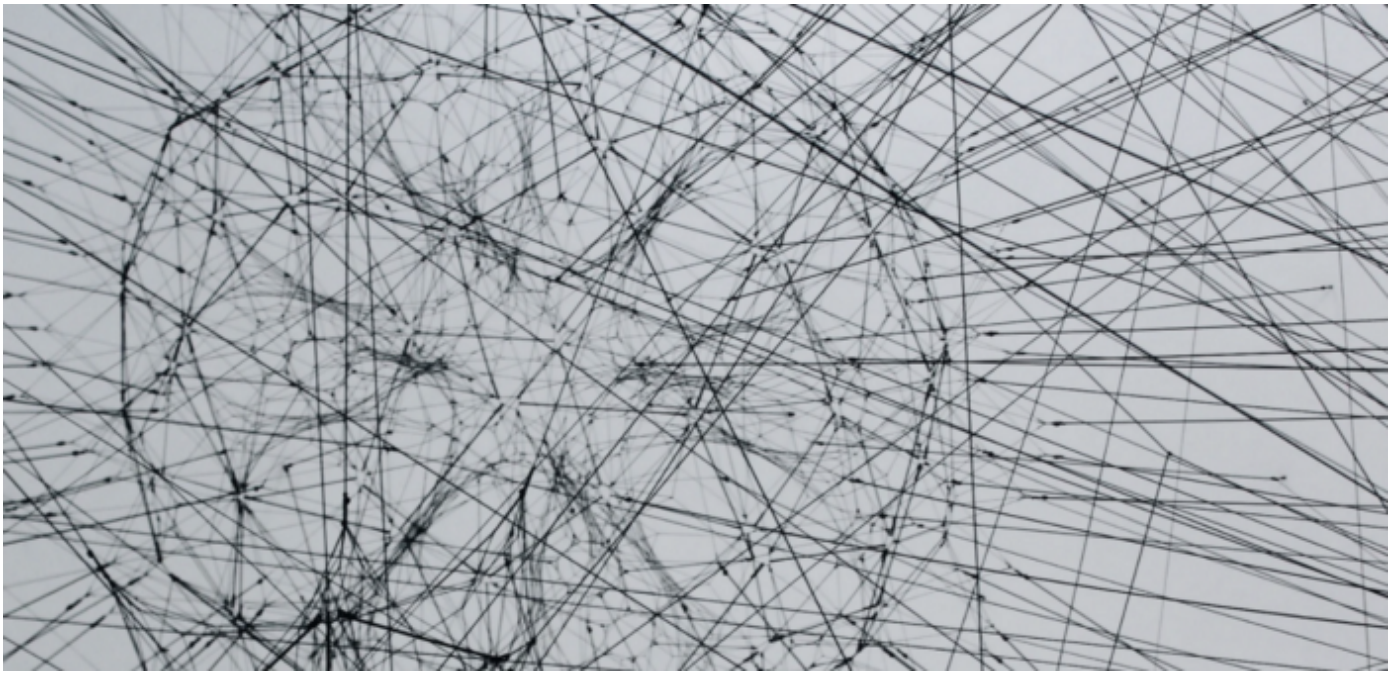


Image is taken from Unsplash by Alina Grubnyak

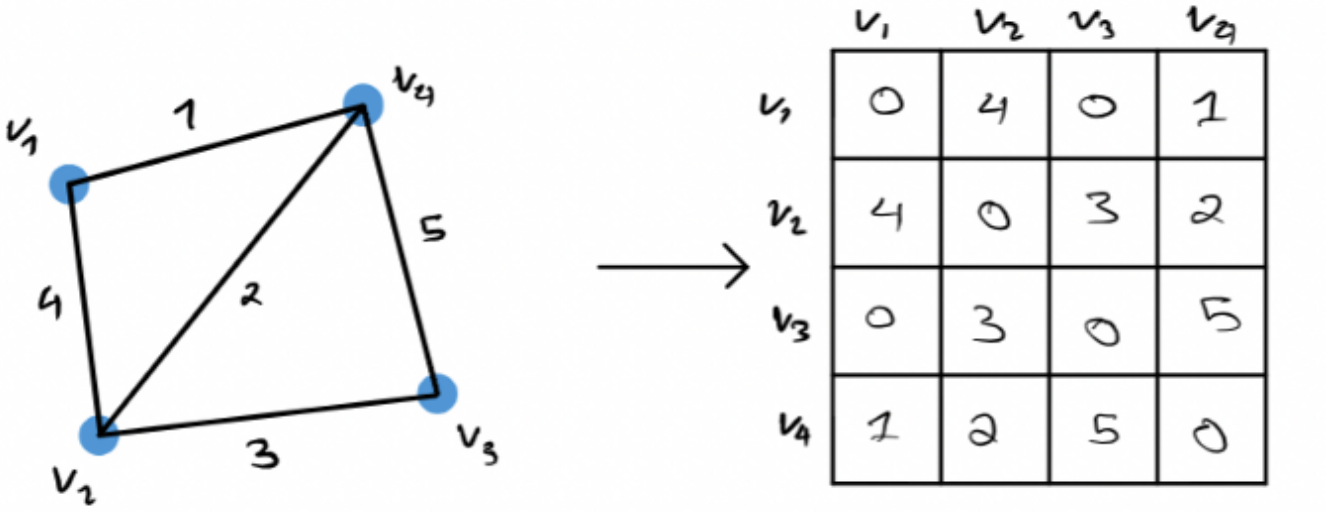
Table of Contents

Introduction

This article is going to give an intuitive and technical understanding of the node2vec algorithm by Aditya Grover and Jure Leskovec. The paper can be found and read here, for those who want to go more in-depth with the technical concepts introduced. The authors of this paper have made an open-source package in python3 called node2vec which can be installed via pip and used for scalable feature learning on networks. There are various open-source implementations of this paper that can be found online, the repository created by the author Aditya can be found here. Another very popular and highly used implementation can also be found here. I will be using the latter package for the purposes of the tutorial and examples later on in this article.

What are Graphs?

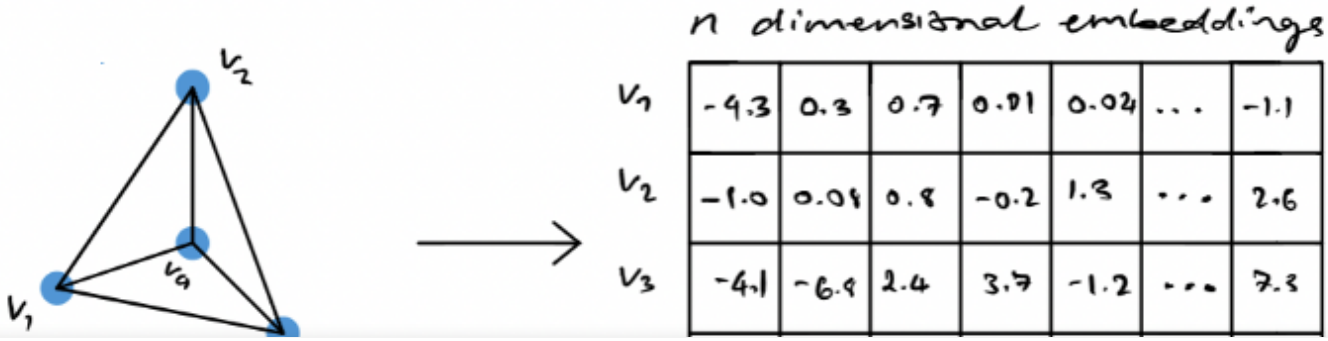
Graphs theory, a field of mathematics dedicated to the studying of graphs. Graphs are mathematical structures that model pairwise relations between nodes [1]. These nodes are connected by edges when there is a relation between nodes. Nodes can also have relations with themselves, otherwise known as a self-loop. There are various forms of graphs, each form has a distinct different characteristic than the other, graphs can be directed, undirected, weighted, bipartite, etc. Each graph can also be mapped to an adjacency matrix. The elements within the adjacency matrix indicate whether pairs of nodes are adjacent to each other or not in the graph [1]. You can reference image 1 for a visual understanding of how a graph would look.



The adjacency matrix of a graph G (Image provided by author)

What is Node2Vec

A notable problem when working with networks, is transforming the network structure into numerical representation which can then be passed onto traditional machine learning algorithms. Node2Vec is an algorithm that allows the user to map nodes in a graph G to an embedding space. Generally, the embedding space is of lower dimensions than the number of nodes in the original graph G. The algorithm tries to preserve the initial structure within the original graph. Essentially, the nodes which are similar within the graph will yield similar embeddings in the embedding space. These embedding spaces are essentially a vector corresponding to each node in the network. The graph embeddings are commonly used as input features to solve machine learning problems oriented around link prediction, community detection, classification, etc.



Generally, when dealing with very large graphs it's quite difficult for scientists to visually represent the data they're working with. A common solution to see how a graph looks is to generate node embeddings associated with that graph and then visualize the embeddings in a lower-dimensional space. This allows you to visually see potential clusters or groups forming in very large networks.

Random Walks Generation

Having an understanding of what random walks are and how they work is crucial in understanding how node2vec works. I'll provide a high-level overview of it, but if you want a more intuitive understanding and implementation of random walks in python you can read the article I've previously written regarding this topic.

Random Walks with Restart Explained

Understand the Random Walk with Restart algorithm and its associated implementation in Python

towardsdatascience.com

As a high level overview, the simplest comparison of a random walk would be through walking. Imagine that each step you take is determined probabilistically. This implies that at each index of time, you have moved in a certain direction based on a probabilistic outcome. This algorithm explores the relationship to each step that you would take and its distance from the initial starting point.

Now you might wonder how these probabilities of moving from one node to another are calculated. Node2Vec introduces the following formula for determining the probability of moving to the node x given that you were previously at the node v.

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

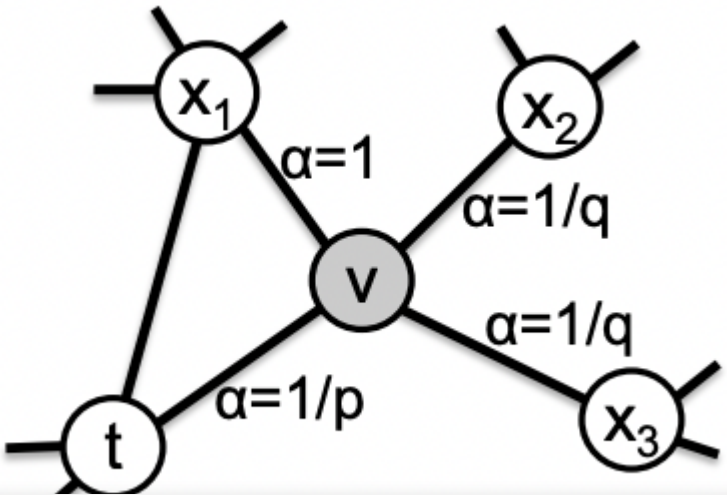
Where z is the normalization constant, and πvx is the unnormalized transition probability between nodes x and v [4]. Clearly, if there is no edge connecting x and v, then the probability will be 0, but if there is an edge, we identify a normalized probability of going from v to x.

The paper states that the easiest way to introduce a bias to influence the random walks would be if there was a weight associated with each edge. However, that wouldn't work in the case of unweighted networks. To resolve this, the authors introduced a guided random walk governed by two parameters p and q. p indicates the probability of a random walk getting back to the previous node, and q indicates the probability that a random walk can pass through a previously unseen part of the graph [4].

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

Image taken from Node2Vec Paper [4]

Where dtx represents the shortest path between nodes t and x. It can be visually seen in the illustration below.



Skip-Gram Architecture

Having a general knowledge of the word2vec is necessary to understand what's being done in node2vec. I wrote an article explaining the intuition and implementation of the word2vec paper a while back. You can check it out here.

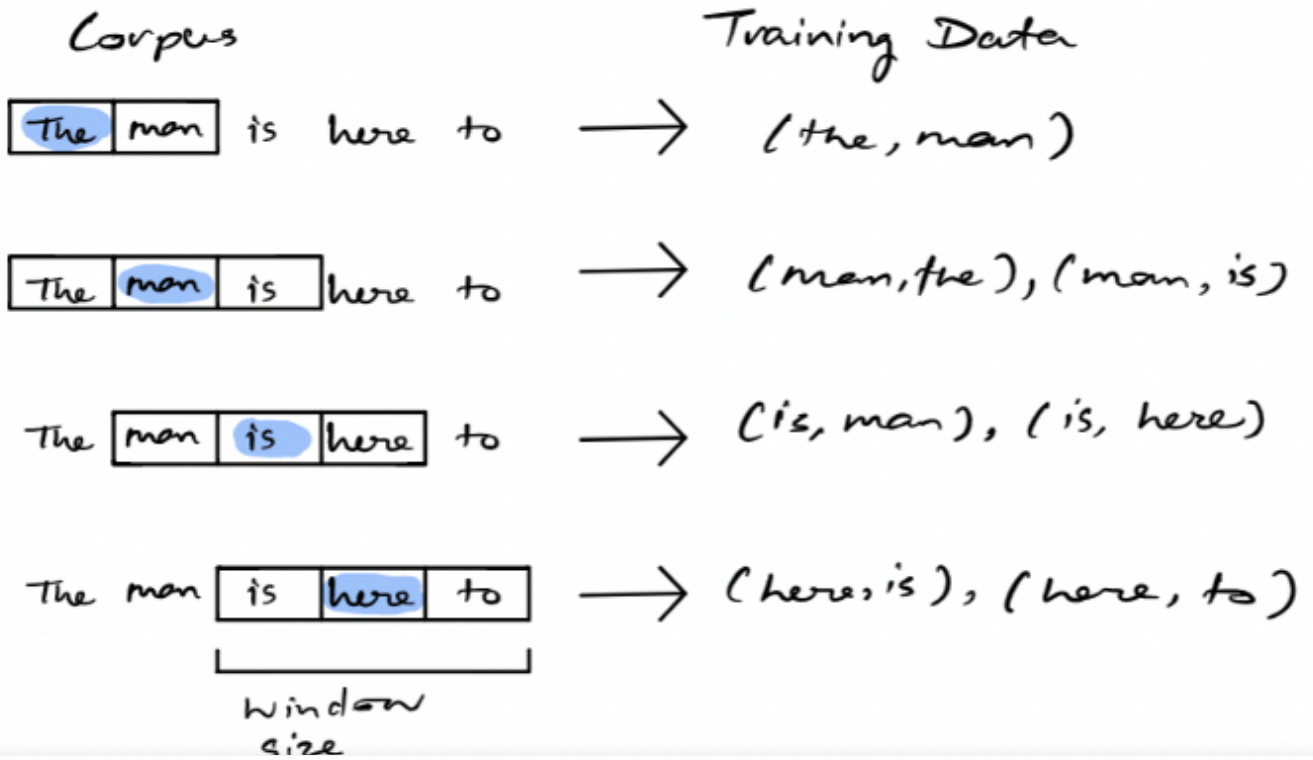
Word2Vec Explained

Explaining the Intuition of Word2Vec & Implementing it in Python

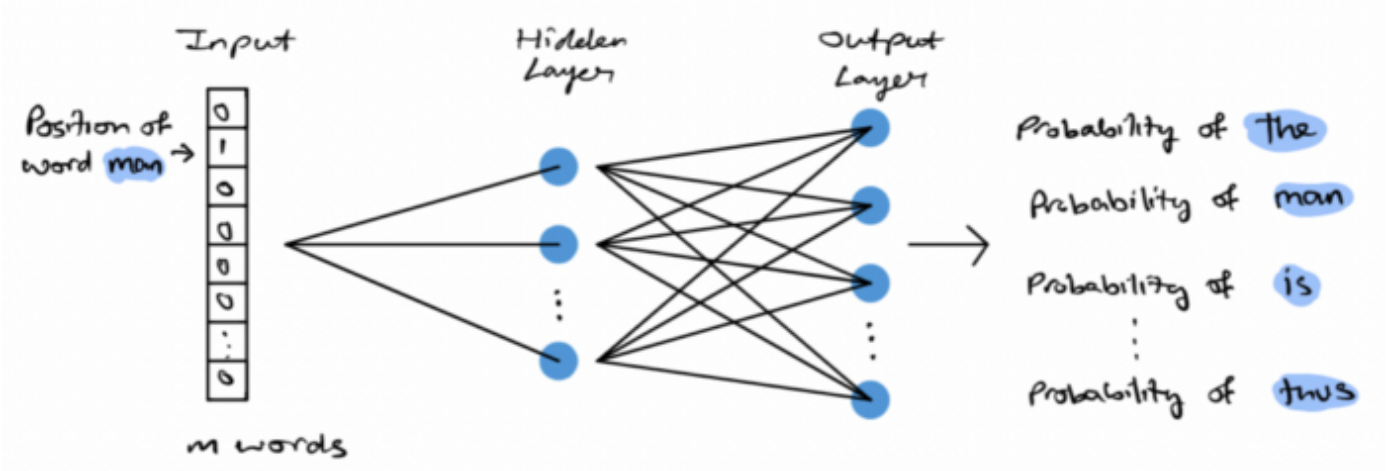
towardsdatascience.com

For the sake of this article, I will provide a high-level overview of the Skip-Gram model.

The skip-gram model is a simple neural network with one hidden layer trained in order to predict the probability of a given word being present when an input word is present [2]. The process can be described visually as seen below.



As seen above, given some corpus of text, a target word is selected over some rolling window. The training data consists of pairwise combinations of that target word and all other words in the window. This is the resulting training data for the neural network. Once the model is trained, we can essentially yield a probability of a word being a context word for a given target [2]. The following image below represents the architecture of the neural network for the skip-gram model.



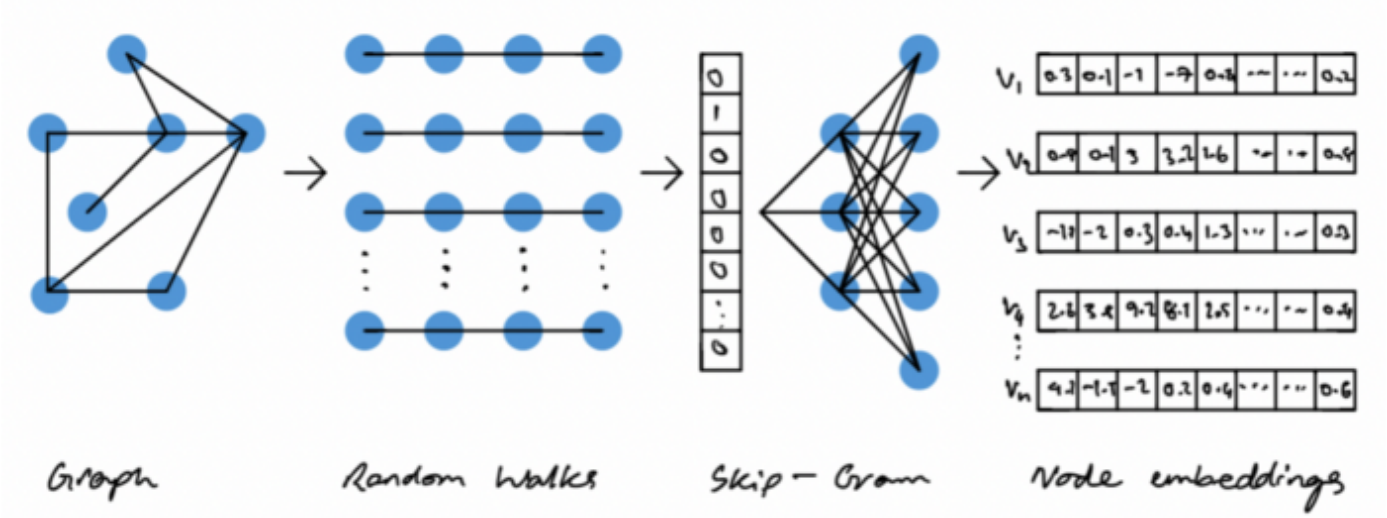
Skip-Gram Model architecture (Image provided by author)

A corpus can be represented as a vector of size N, where each element in N corresponds to a word in the corpus. During the training process, we have a pair of target and context words, the input array will have 0 in all elements except for the target word. The target word will be equal to 1. The hidden layer will learn the embedding representation of each word, yielding a d-dimensional embedding space [2]. The output layer is a dense layer with a softmax activation function. The output layer will essentially yield a vector of the same size as the input, each element in the vector will consist of a probability. This probability indicates the similarity between the target word and the associated word in the corpus.

How it Works

The process for node2vec is fairly simple, it begins by inputting a graph and extracting a set of random walks from the input graph. The walks can then be represented as a

model yields an embedding for each node (or word in this analogy). The entire process can be seen below.



Node2Vec Architecture (Image provided by the author)

Implementation

This section of the article will focus on the implementation of node2vec in Python. The following are the libraries and versions used in the article. For this implementation, we will generate a random graph, apply node2vec to the graph and then visualize the embeddings in a lower dimensional space using PCA. If you want to go through the notebook which was used for this example you can find it here.

Requirements

```
Python=3.8.8
networkx=2.5
pandas=1.2.4
numpy=1.20.1
matplotlib=3.3.4
node2vec=0.4.4
seaborn=0.11.1
sklearn=0.24.1
gensim=4.0.1
```

Generate Network

```
import networkx as nx
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.decomposition import PCA
from node2vec import Node2Vec as n2v
sns.set()

def generate_graph_deg_dist(deg_dist, n):
    '''
    This function will generate a networkx graph G based on a degree distribution
    provided by the user.

    params:
        deg_dist (Dictionary) : The key will be the degree and the value is the probability
                                of a node having that degree. The probabilities must sum to
                                1
        n (Integer) : The number of nodes you want the graph to yield

    example:
        G = generate_graph_deg_dist(
                deg_dist = {
                    6:0.2,
                    3:0.14,
                    8:0.35,
                    4:0.3,
                    11:0.01
                },
                n = 1000
        )
    '''
    deg = list(deg_dist.keys())
    proba = list(deg_dist.values())
    if sum(proba) == 1.:
        deg_sequence = np.random.choice(
```

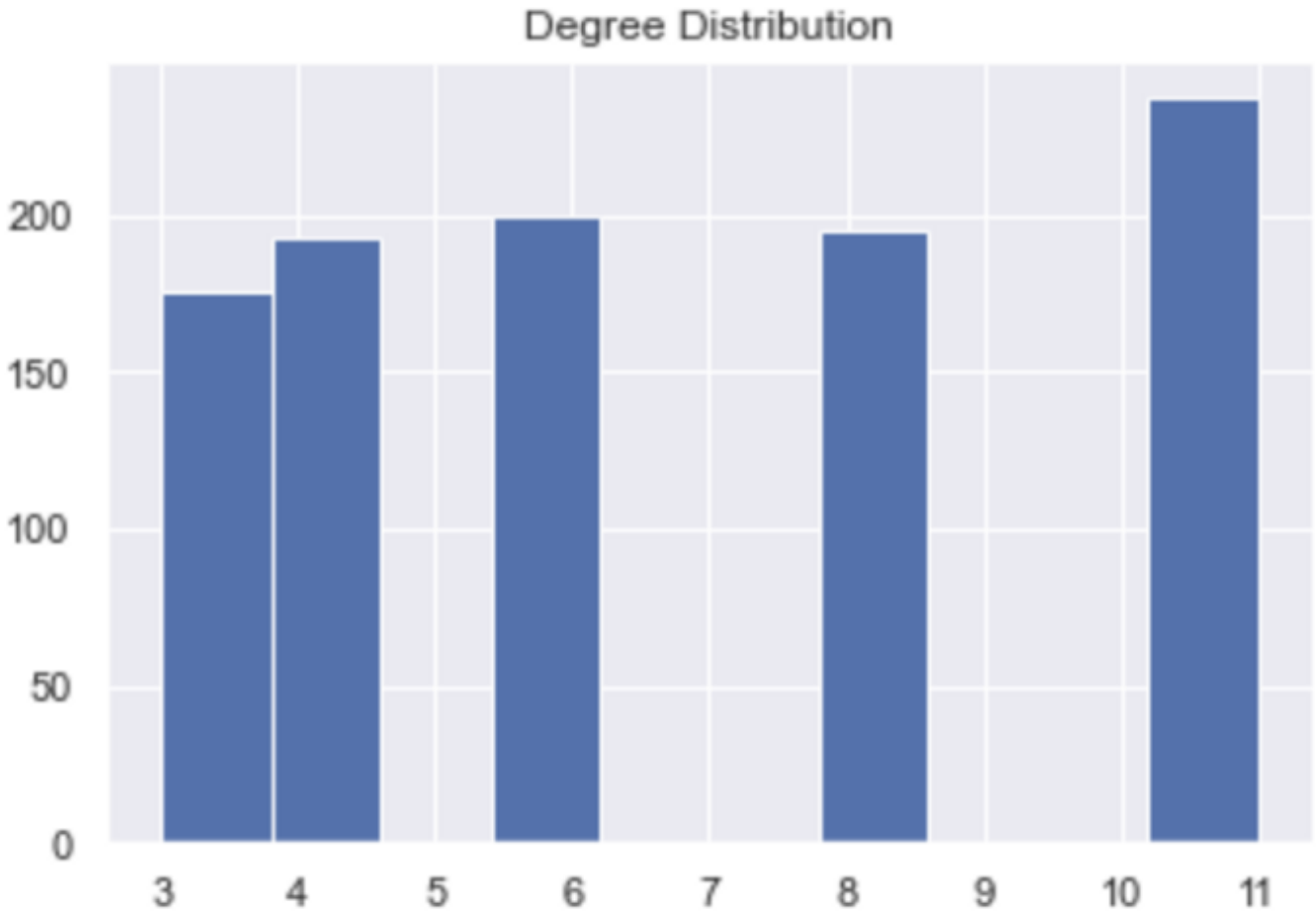

```
        if sum(deg_sequence) % 2 != 0:
            # to ensure that the degree sequence is always even for the configuration model
            deg_sequence[1] = deg_sequence[1] + 1

        return nx.configuration_model(deg_sequence)
    raise ValueError("Probabilities do not equal to 1")

G = generate_graph_deg_dist(
        deg_dist = {
            6:0.2,
            3:0.14,
            8:0.35,
            4:0.3,
            11:0.01
        },
        n = 1000
)

print(nx.info(G))

# visualize degree distribution
plt.clf()
plt.hist(list(dict(G.degree()).values()))
plt.title('Degree Distribution')
plt.show()
```

generate_graph.py hosted with ❤ by GitHub view raw

The script above will generate a random graph for us to use node2vec on. The user will specify the number of nodes and the degree distribution they want for the randomly generated network. The network will be generated through the configuration model. The configuration model essentially generates a random graph through assigning edges to match a degree sequence. Do note that since this is random, the resulting network will be different each time. Furthermore, this is just an example network to run node2vec on, just because the resulting network is a multi-graph doesn't mean that node2vec can only run on other multi-graphs. Node2Vec can be ran on directed, undirected, weighted, multi, or regular networks. When I ran the function above for `n = 1000` the following is the stats


```
Name:
Type: MultiGraph
Number of nodes: 1000
Number of edges: 3332
Average degree:   6.6640
```

The stats associated with the generated network (Image provided by author)



The degree distribution associated with the generated network (Image provided by author)

If there's a network in particular you want to test this algorithm out on then feel free to exclude this segment of the article. Apply node2vec on the graph you've generated. In general, there should be a data preprocessing phase after graph generation, usually if

```
Parameter Info
- graph: a graph g, where all nodes must be integers or strings
- dimensions: embedding dimensions (default: 128)
- walk_length: number of nodes in each walk (default: 80)
- num_walks: number of walks per node (default: 10)
- weight_key: the key for the weight attribute on weighted graphs
(default: 'weight')
- workers: number of workers for parallel execution (default: 1)
- p: the probability of a random walk getting back to the previous
node (default: 1)
- q: probability that a random walk can pass through a previously
unseen part of the graph (default: 1)
```

The Node2Vec.fit method accepts any keyword argument acceptable by `gensim.Word2Vec`. The parameters mentioned above are documented in the node2vec library [3]. For the purposes of this article, I've set the window value to be 1, the min_count to be 1, the batch_words to be 4, and the dimensions to be 16. The rest of the parameters not mentioned are set to the default values provided by word2vec. Feel free to adjust these parameters accordingly to your own problem.

```
WINDOW = 1 # Node2Vec fit window
MIN_COUNT = 1 # Node2Vec min. count
BATCH_WORDS = 4 # Node2Vec batch words

g_emb = n2v(
  G,
  dimensions=16
)

mdl = g_emb.fit(
    vector_size = 16,
    window=WINDOW,
    min_count=MIN_COUNT,
    batch_words=BATCH_WORDS
)

input_node = '1'
```


Since this implementation uses word2vec in the backend, you are able to identify other nodes similar to an input node. Keep in mind that the input node must be passed in as a string, and it will output the top N (default is 10) most similar nodes to the input node in the form of a list in descending order of similarity.

```
('433', 0.8332405090332031)
('277', 0.8205791115760803)
('14', 0.7698894739151001)
('340', 0.7587181329727173)
('777', 0.7562111020088196)
('770', 0.7561677694320679)
('647', 0.7538766264915466)
('415', 0.7531223893165588)
('389', 0.7500568628311157)
('428', 0.7285109758377075)
```

The most similar nodes to the input_node 1 (Image provided by the author)

Convert to DataFrame

```
emb_df = (
    pd.DataFrame(
        [mdl.wv.get_vector(str(n)) for n in G.nodes()],
        index = G.nodes
    )
)
```

emb2df.py hosted with ❤ by GitHub view raw

Visualize Embeddings

```
        pd.DataFrame(
            pca_mdl,
            columns=['x','y'],
            index = emb_df.index
        )
    )
    plt.clf()
    fig = plt.figure(figsize=(6,4))
    plt.scatter(
        x = emb_df_PCA['x'],
        y = emb_df_PCA['y'],
        s = 0.4,
        color = 'maroon',
        alpha = 0.5
    )
    plt.xlabel('PCA-1')
    plt.ylabel('PCA-2')
    plt.title('PCA Visualization')
    plt.plot()
```

visualize_embeddings.py hosted with ❤ by GitHub view raw

Node2Vec embedding visualized through PCA. Each dot represents a node in the original network (Image provided by author)

Do be aware that for the purposes of this article, this visualization has no meaning. This is because we've generated the network with edges at random. If you used an actual network representing some dataset, you could make some interesting observations. In the graph, each individual dot corresponds to a node, as per the paper described nodes would be closer together in proximity if they are similar with each other. It would be a simple way to see if there are any clusters / communities formed with your data.

Why use Node2Vec?

1. It's scalable and parallelizes easily
2. Open sourced in python & spark
3. Unique approach to learning feature representation via node embeddings
4. The structure of the original network is preserved through the embeddings
5. Node2Vec has many real world applications including and not limited to node classification, community detection, link prediction, etc.

Concluding Remarks

Overall, I think the main takeaways from this article should be that node2vec generates embeddings associated with each node in a given network. These embeddings preserve the original structure of the network, thus similar nodes will have "similar" embeddings. This is an open source algorithm and scales very well to deal with larger networks (as long as you have the computing powers for it).

Even after reading through my article, I encourage you to read the original paper and try to apply it to conduct some analysis / solve some problems. The original paper can be found here.

If you want to see how you can use Node2Vec in the context of link prediction and