node2vec: Embeddings for Graph Data

Elior Cohen Apr 16, 2018 · 8 min read

Hotlinks:
Original article: node2vec: Scalable Feature Learning for Networks, Aditya Grover and Jure Leskovec
Algorithm implementation — By me: Github repo — Python3
Algorithm implementation — By the algo author: Github repo (by Aditya Grover) — Python2
Showcase code: https://github.com/eliorc/Medium/blob/master/Nod2Vec-FIFA17-Example.ipynb

Motivation

Embeddings… A word that every data scientist has heard by now, but mostly in the context of NLP.
So why do we even bother embedding stuff?
As I see it, creating quality embeddings and feeding it into models, is the exact opposite of the famous say "Garbage in, garbage out" .
When you feed low quality data into your models, you put the entire load of learning on your model, as it will have to learn all the necessary conclusions that could be derived

your data and thus make the task of learning the problem easier for your models.
Another point to think about is **information** vs **domain knowledge**.
For example, let's consider word embeddings (word2vec) and bag of words representations.
While both of them can have the entire **information** about which words are in a sentence, word embeddings also include **domain knowledge** like relationship between words and such.
In this post, I'm going to talk about a technique called **node2vec** which aims to create embeddings for nodes in a graph (in the G(V, E, W) sense of the word).

$$node2vec(G(V,E,W)) \rightarrow \mathbb{R}^n$$

I will explain **how it works** and finally supply my own **implementation** for Python 3, with some extras.

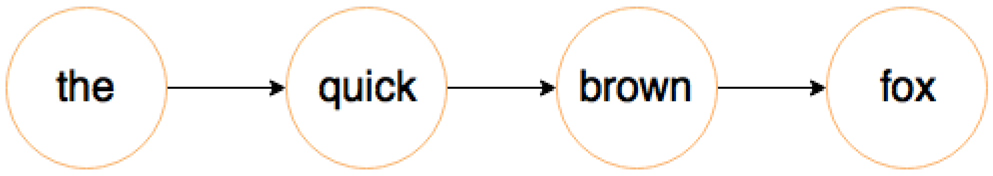
Embedding process

So how is done?
The embedding themselves, are learnt in the same way as word2vec's embeddings are learnt — using a skip-gram model.
If you are familiar with the word2vec skip-gram model, great, if not I recommend this great post which explains it in great detail as from this point forward I assume you are familiar with it.

The most natural way I can think about explaining node2vec is to explain how node2vec generates a "corpus" — and if we understand word2vec we already know how to embed a corpus.

So how do we generate this corpus from a graph? That's exactly the innovative part of node2vec and it does so in an intelligent way which is done using the **sampling strategy**.

this is a perfect representation for a text sentence, where each word in the sentence is a node and it points on the next word in the sentence.

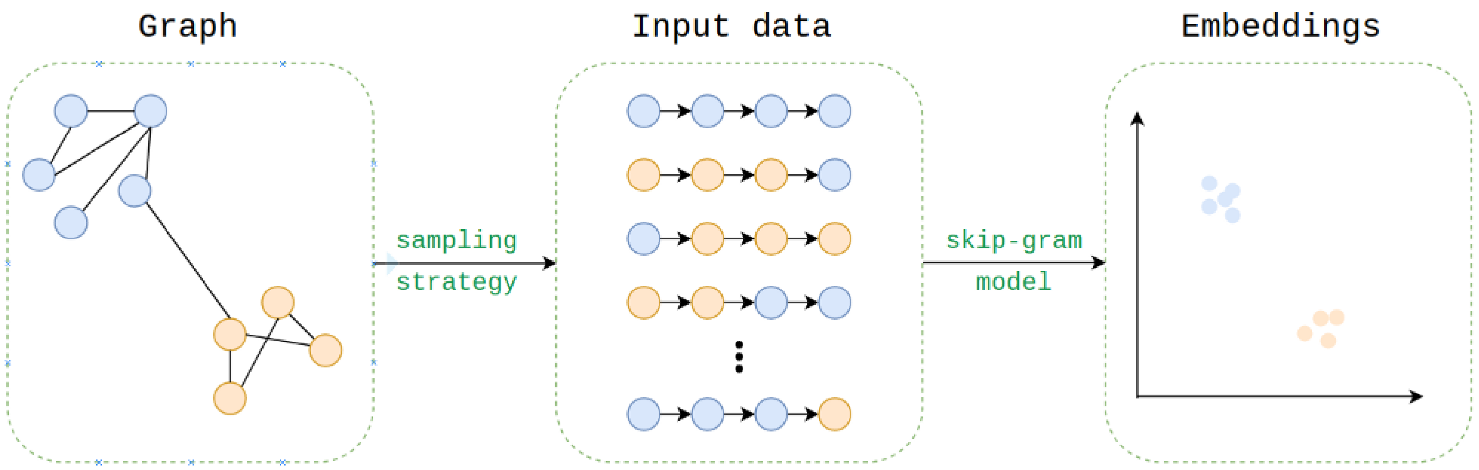


Sentence in a graph representation

In this way, we can see that word2vec can already embed graphs, but a very specific type of them.
Most graphs though, aren't that simple, they can be (un)directed, (un)weighted, (a)cyclic and are basically much more complex in structure than text.

In order to solve that, node2vec uses a tweakable (by hyperparameters) sampling strategy, to sample these directed acyclic subgraphs. This is done by generating random walks from each node of the graph. Quite simple right?

Before we delve how the sampling strategy uses the hyperparameters to generate these sub graphs, lets visualize the process:

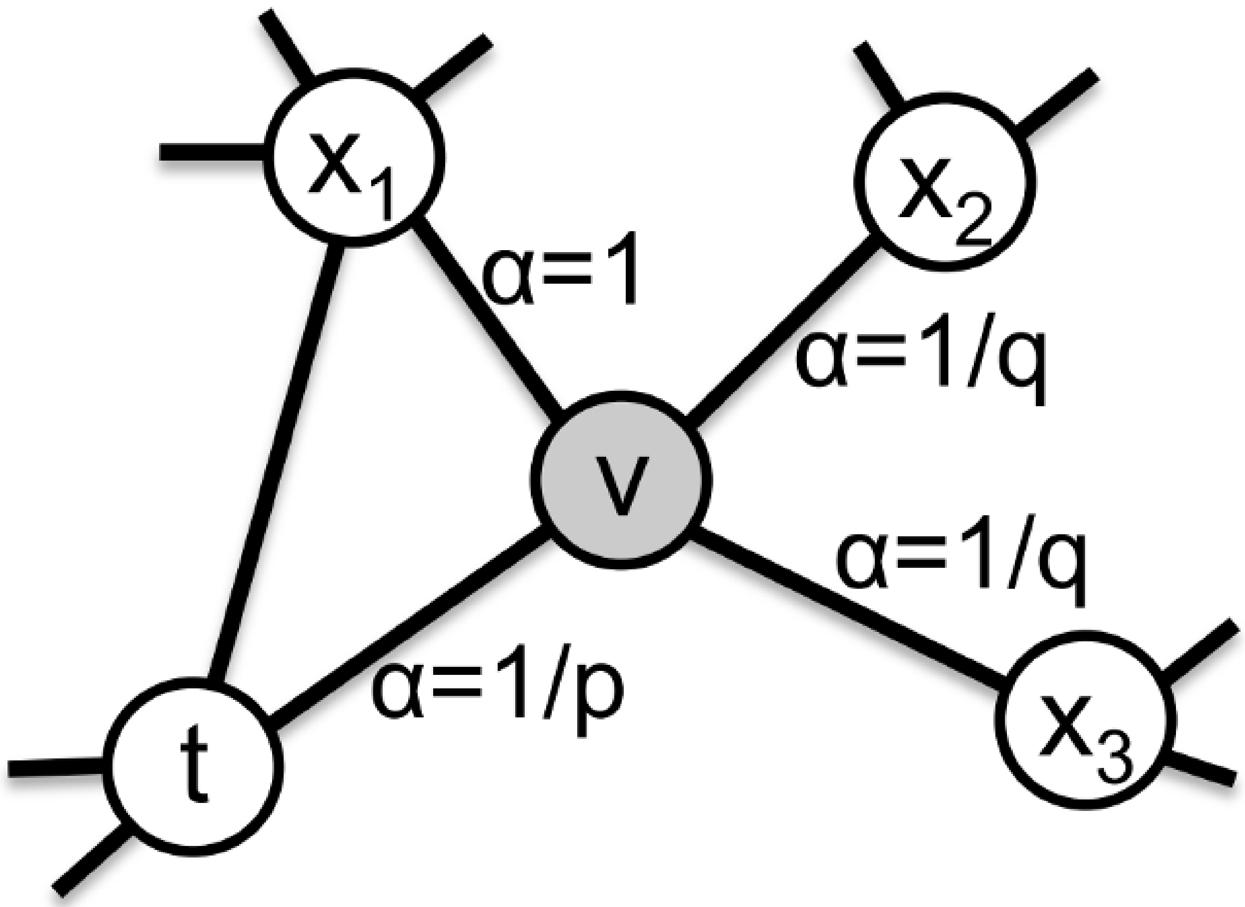


Node2vec embedding process

Sampling strategy

— **Number of walks**: Number of random walks to be generated from each node in the graph
— **Walk length**: How many nodes are in each random walk
— **P**: Return hyperparameter
— **Q**: Inout hyperaprameter
and also the standard skip-gram parameters (context window size, number of iterations etc.)

The first two hyperparameters are pretty self explanatory.
The algorithm for the random walk generation will go over each node in the graph and will generate <**number of walks**> random walks, of length <**walk length**>.
Q and **P**, are better explained with a visualization.
Consider you are on the random walk, and have just transitioned from node <*t*> to node <*v*> in the following diagram (taken from the article).



P controls the probability to go back to <**t**> after visiting <**v**>.
Q controls the probability to go explore undiscovered parts of the graphs.
In a intuitive way, this is somewhat like the perplexity parameter in tSNE, it allows you to emphasize the local/global structure of the graph.
Do not forget that the weight is also taken into consideration, so the final travel probability is a function of:
1. The previous node in the walk
2. P and Q
3. Edge weight

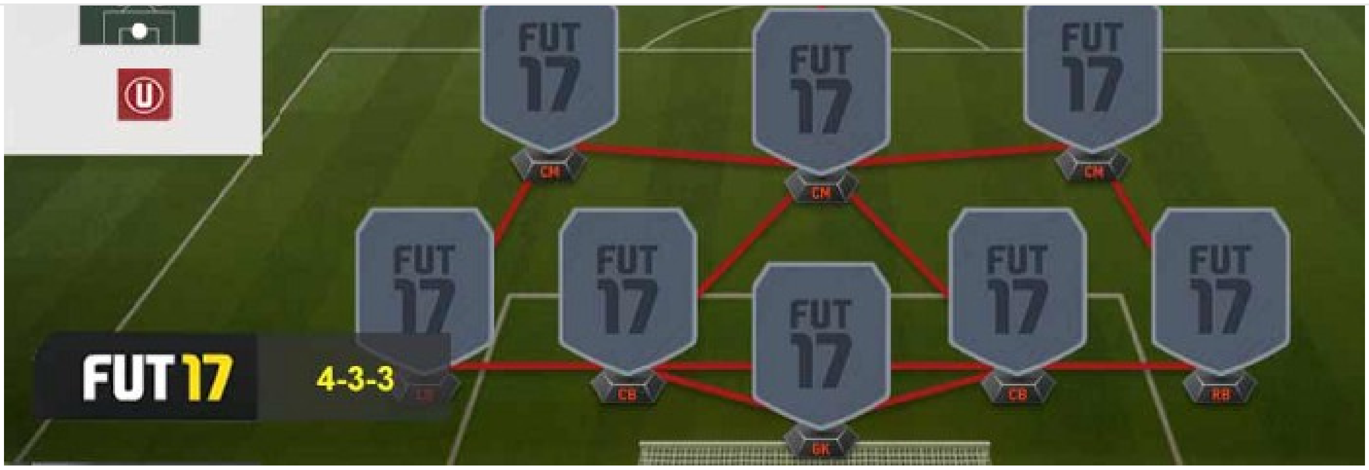
This part is important to understand as it is the essence of node2vec. If you did not fully comprehend the idea behind the sampling strategy I strongly advise you to read this part again.

Using the sampling strategy, node2vec will generate “sentences” (the directed subgraphs) which are will be used for embedding just like text sentences are used in word2vec. Why change something if it works right?

Code (showcase)

Now its time to put node2vec into action.
You can find the entire code for this node2vec test drive here.
I am using for the example my implementation of the node2vec algorithm, which adds support for assigning node specific parameters (q, p, num_walks and walk length).

What we are going to do, using formation of European football teams, is to embed the teams, players and positions of 7 different clubs.
The data I’m going to be using is taken from the FIFA 17 dataset on Kaggle.
In FIFA (by EASports) each team can be represented as a graph, see picture below.



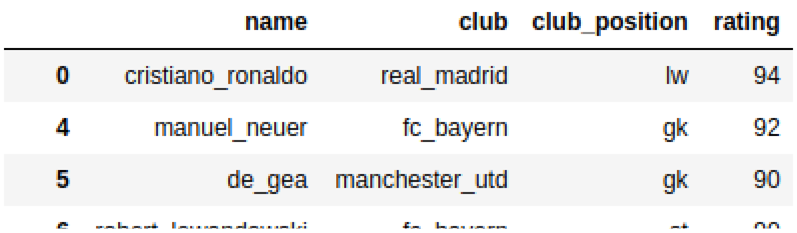
Formation example from FIFA17, easily interpreted as a graph

As we can see, each position is connected to other positions and when playing each position is assigned a player.
There dozens of different formations, and the connectivity between them differs. Also there are type of positions that are in some formations but are non existent in others, for example the 'LM' position is not existent in this formation but is in others.

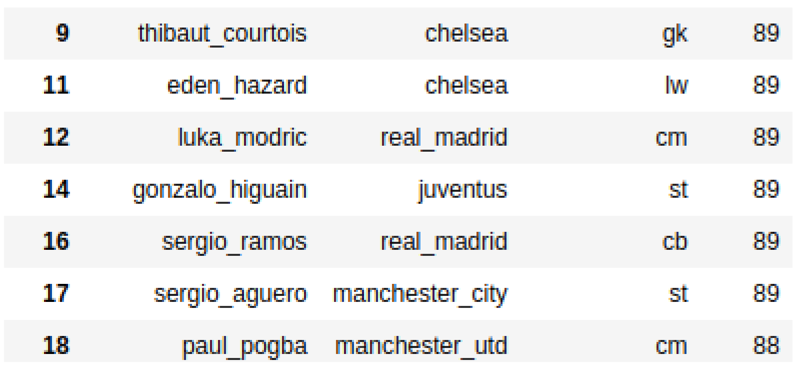
This is how we are going to do this:
1. Nodes will be players, team names and positions
2. For each team, create a separate graph where each player node is connected to his team name node, connected to his teammates nodes and connected to his teammate position nodes.
3. Apply node2vec to the resulting graphs

**Notice: In order to create separate nodes for each position inside and between teams, I added suffixes to similar nodes and after the walk generation I have removed them. This is a technicality, inspect code in repo for better understanding*

First rows of the input data looks like this (after some permutations):



	name	club	club_position	rating
0	cristiano_ronaldo	real_madrid	lw	94
4	manuel_neuer	fc_bayern	gk	92
5	de_gea	manchester_utd	gk	90



9	thibaut_courtois	chelsea	gk	89
11	eden_hazard	chelsea	lw	89
12	luka_modric	real_madrid	cm	89
14	gonzalo_higuain	juventus	st	89
16	sergio_ramos	real_madrid	cb	89
17	sergio_aguero	manchester_city	st	89
18	paul_pogba	manchester_utd	cm	88

Sample rows from the input data

Then we construct the graph, using the FIFA17 formations.
Using my node2vec package the graph must be an instance of `networkx.Graph`.
Inspecting the graph edges after this, we will get the following

```
for edge in graph.edges:
    print(edge)

>>> ('james_rodriguez', 'real_madrid')
>>> ('james_rodriguez', 'cm_1_real_madrid')
>>> ('james_rodriguez', 'toni_kroos')
>>> ('james_rodriguez', 'cm_2_real_madrid')
>>> ('james_rodriguez', 'luka_modric')
>>> ('lw_real_madrid', 'cm_1_real_madrid')
>>> ('lw_real_madrid', 'lb_real_madrid')
>>> ('lw_real_madrid', 'toni_kroos')
>>> ('lw_real_madrid', 'marcelo')
...
```

As we can see, each player is connected to his team, the positions and teammates according to the formation.
All of the suffixes attached to the positions will be returned to their original string after the walks are computed (`lw_real_madrid → lw`).

So now that we have the graph, we execute node2vec

```
# pip install node2vec
```

```
node2vec = Node2Vec(graph, dimensions=20, walk_length=16,
num_walks=100)

# Reformat position nodes
fix_formatted_positions = lambda x: x.split('_')[0] if x in
formatted_positions else x

reformatted_walks = [list(map(fix_formatted_positions, walk)) for
walk in node2vec.walks]

node2vec.walks = reformatted_walks

# Learn embeddings
model = node2vec.fit(window=10, min_count=1)
```

We give `node2vec.Node2Vec` a `networkx.Graph` instance, and after using `.fit()` (which accepts any parameter accepted by we get a `gensim.models.Word2Vec`) we get in return a `gensim.models.Word2Vec` instance.

First we will inspect the similarity between different nodes.
We expect the most similar nodes to a team, would be its teammates:

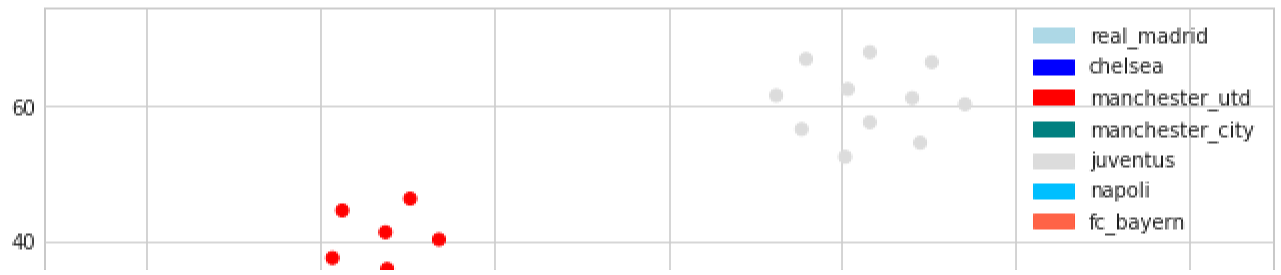
```
for node, _ in model.most_similar('real_madrid'):
    print(node)

>>> james_rodriguez
>>> luka_modric
>>> marcelo
>>> karim_benzema
>>> cristiano_ronaldo
>>> pepe
>>> gareth_bale
>>> sergio_ramos
>>> carvajal
>>> toni_kroos
```

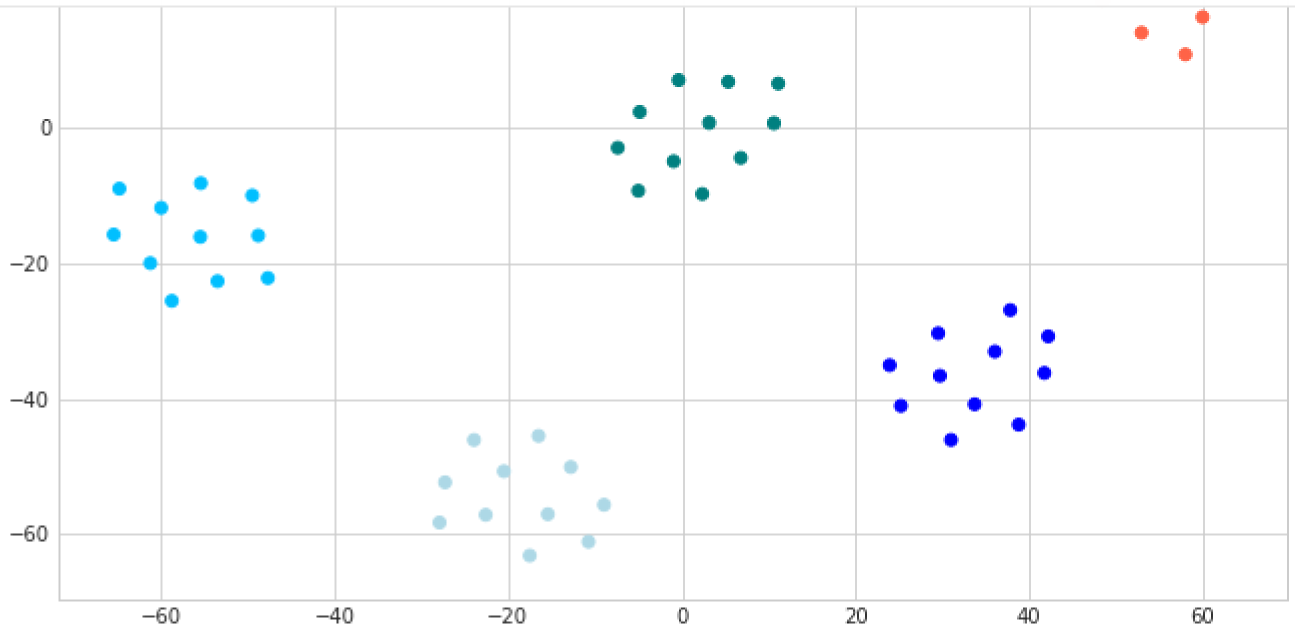
For those who are not familiar with European football, these are all indeed Real Madrid’s players!

```
# Right Wingers
for node, _ in model.most_similar('rw'):
    # Show only players
    if len(node) > 3:
        print(node)

>>> pedro
>>> jose_callejon
>>> raheem_sterling
>>> henrikh_mkhitaryan
>>> gareth_bale
>>> dries_mertens

# Goal keepers
for node, _ in model.most_similar('gk'):
    # Show only players
    if len(node) > 3:
        print(node)

>>> thibaut_courtois
>>> gianluigi_buffon
>>> keylor_navas
>>> azpilicueta
>>> manuel_neuer
```

In the first try (right wingers) we indeed get different right wingers from different clubs, again a perfect match.
In the second try though, we get all goalkeepers except Azpilicueta which is actually a defender — this could be due to the fact that goalkeepers are not very connected to the team, only to central backs usually.

Works pretty good right? Just before we finish, lets use tSNE to reduce dimensionality and visualize the player nodes.





Visualization of player nodes (tSNE reduced dimensionality)

Check it out, we get beautiful clusters based on the different clubs.

Final Words

Graph data is almost everywhere, and where its not you can usually put it on a graph yet the node2vec algorithm is not so popular.
The algorithm also grants great flexibility with its hyperparameters so you can decide which kind of information you wish to embed, and if you have the the option to construct the graph yourself (and is not a given) your options are limitless.
Hopefully you will find use in this article and have added a new tool to your machine learning arsenal.

If someone wants to contribute to my node2vec implementation, please contact me.