Graph/Node embedding

In the context of graph embedding, embedding is a shortcut for embedding in low dimensions, and can be understood as assigning to some elements of the graph a vector (i.e., a list of numbers) composed of d elements. d is the number of dimensions in the embedding space, and d should be small.

Types of Network Embedding

According to the type of element which is embedded, we can differentiate:

- Node Embedding (one vector per node)
- Edge Embedding (one vector per edge)
- Substructure Embedding (e.g., one vector per community)
- Whole Graph Embedding (one vector per graph)

In this class, we will introduce only Node Embedding, which is the most popular approach. Whole graph embedding is also quite popular, for instance to classify types of networks.

Node Embedding

In node embedding, the vector assigned to each node should be a proxy, a sort of numeric summary of the position of the node in the graph, in term of topology. Several types of embeddings exist that capture different aspects of the network topology, in particular we will differentiate locational embedding from role embedding. Note that node embedding is sometimes called graph embedding in the literature.

Embedding distance

Since each node is represented by a vector, it is possible to compute a distance between nodes in the embedding. Intuitively, two nodes occupying similar positions in the network (according to what the chosen embedding capture) should have similar embedding vectors. The notion of distance to use (cosine, euclidean, etc.) also depends on the chosen embedding.

Adjacency matrix(A) as an embedding

A naive way to choose an embedding (in n dimensions) could be to consider each row of the adjacency matrix as the vector representation of the node it corresponds to. This embedding would capture what is called the structural equivalence, i.e. the fact that nodes share similar neighborhoods. Two nodes with the same neighborhoods would have the same vectors. If the Manhattan distance were used, the distance between nodes in the embedding would correspond to the number of different neighbors.

What is a good embedding?

What is a good embedding depends on the task that we want to achieve. In the perspective of this class, embeddings are mostly used as features for machine learning tasks. As such, they must be 1 ln as few dimensions as possible. Machine learning suffers from what is known as the curse of dimensionality, and tends to work better with lower dimensions, without talking about computational advantages. 2) As dense as possible. Sparsity--usually associated with high dimensions-- makes learning harder. Furthermore, the embedded properties must be meaningful for the task to achieve. For instance, the notion of distance captured by the adjacency matrix seems in contradiction with the intuition: in graphs, one usually use the number of common neighbors, and/or normalized fraction of neighbors (Jaccard, etc.) rather than a raw count of different neighbors.

Embedding and Dimensionality Reduction

In Machine Learning, when a dataset is composed of too many features, dimensionality reduction algorithms can be used to generate a smaller number of synthetic features, defined as combination of the original ones. Common algorithms to do so are for instance PCA (Principal Component Analysis) and T-SNE. A simple method to generate a better embedding from the adjacency matrix would be to apply Dimensionality Reduction on it to reduce its number of dimension. Its counter-intuitive definition of distance would nevertheless remain a problem.

![Cheatsheet](image)

**Node Embedding: LE**

Laplacian Eigenmaps (LE) is a method that can be used for node embedding, whose objective function is defined as follows:

\[ y = \min_{y} \sum_{i \neq j} \| y_i - y_j \|^2 S_{ij} \]

This can be read as follows: to find the embedding y of a graph, we need to assign an embedding y_i to each node i such as the sum (over all node pairs) of the equation \( \| y_i - y_j \|^2 S_{ij} \) is minimal. Said differently, its objective is to minimize the product between the euclidean distance in the embedding \( \| y_i - y_j \|^2 \) and the similarity in the graph S_{ij}. If two nodes are similar/close in the graph (high value), we need to make them as close as possible in the embedding (small value). Nodes dissimilar/distant in the graph can be distant in the embedding with a lesser penalty. To forbid a trivial solution of all nodes being on the same location, the sum of distance between points in the embedding must be equal to a constant.

**Notations**

- \( y \) Embedding of the graph
- \( y_i \) Vector corresponding to node i in the embedding y
- \( S \) Similarity matrix. For each pair of node i, j, S_{ij} represents the graph similarity that we want to preserve.

By default, \( S = A \) : two nodes have a maximal similarity of 1 if they are connected, and similarity 0 if they are not connected. But one can use a different notion, such as a random walk distance, a neighborhood similarity heuristic, etc.
**Word Embedding: Skipgram/word2vec**

In practice, a word is considered to be in the context of another if it is at a distance less than \( l \) in a sentence. From a corpus, one then extracts the probability \( p(w_j|w_i) \) for each word \( w_i \), that a word taken at random in its context is \( w_j \). The objective function of word2vec can be expressed as:

\[
y = \min_{i,j} p(w_j|w_i) - \sigma(y_i y_j^T)
\]

with \( \sigma \) the softmax function defined as \( \frac{e^x}{\sum e^x} \), a function commonly used in neural networks to add non-linearity while ensuring that the solution is a probability.

**Word Embedding: word2vec - context**

The principle proposed in a famous method called word2vec is to use the context, i.e., the words encountered around a word in sentences of a corpus, to discover the semantic similarity. In summary, the more two words are encountered in a same context, the more they are considered similar. For instance, a corpus might contain sentences such as: "the dog eat dry food", and "the cat eat dry food". cat and dog are found in similar contexts, which should drive them closer in the embedding. In other sentences, their contexts differs, which should drive them away in the embedding.

**Matrix decomposition and Eigenvectors**

A diagonalizable matrix \( A \) can be factorized using eigenvectors as follows:

\[
A = Q \Lambda Q^{-1}
\]

where \( Q \) is the \( n \times n \) matrix whose \( i \)th column is the eigenvector \( q_i \) of \( A \), and \( \Lambda \) is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, \( \Lambda_{ii} = \lambda_i \). Keeping as embedding the eigenvectors associated with the largest eigenvalues means that we can reconstruct the original matrix with a good precision. This is the same method used by PCA (Principal Component Analysis), on a covariance (or correlation) matrix, which is also a form of similarity matrix.

**Random Walk NN based embedding**

In recent years, new approaches based on random walks and neural networks have encountered a large success and re-launched a large interest in graph embedding for various applications. They are transpositions of techniques developed for the embedding of words to the graph setting.

**Word Embedding**

Machine Learning on text suffers from a problem similar to Machine Learning on graphs: words are not numbers and cannot be naturally represented as meaningful vectors. Word embedding objective is to assign a low dimensional vector to each word such as two words with similar semantic have similar vectors.

**Node Embedding: HOPE**

Higher-Order Proximity preserved Embedding (HOPE) objective function is:

\[
y = \min \sum_{i,j} |S_{ij} - y_i y_j^T|
\]

Said differently, its objective is to minimize the difference between the graph similarity \( S_{ij} \) and the similarity in the embedding, computed as the product of embedding vectors. Vectors are imposed to be normalized, thus \( y_i y_j^T \) corresponds to the cosine similarity. Two nodes close (resp. far) in the graph should therefore be close (far) in the embedding. Relative distances should also be conserved.

**Skipgram: a neural network formulation**

The skipgram algorithm is solved, in practice, using tools and methods of neural networks, which make it scalable to large datasets. It can then be represented as follow:

https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614ed6e0

**Word2vec efficacy**

Word2Vec (and following word embedding approaches) have encountered an enormous success in the Natural Language Processing domain, and are nowadays used for most practical tasks such as automatic language translation, sentiment analysis, personal assistants, etc.

Various other fields, including network science, have therefore adapted the mechanism to embed other complex elements.

**DeepWalk**

DeepWalk is the direct transcription of Word2Vec to graphs. The principle is to generate random walks in the graph, playing the role of sentences in a corpus. The probability of finding a word in the vicinity of another therefore translates in the probability of encountering a node in a random walk from another.

To sum up, the objective function can now be expressed as:

\[
y = \min \sum_{(i,j)} p(n_j|n_i) - \sigma(y_i y_j^T)
\]

with \( p(n_j|n_i) \) the probability to encounter node \( n_j \) in a random walk of a chosen length starting from node \( n_i \). Its objective is therefore to make the distance in the embedding proportional to a random walk based distance in the graph.

\[^a\text{Perozzi, Al-Rfou, and Skiena 2014}\]
DeepWalk complexity
Contrary to matrix decomposition based approaches, DeepWalk does not require explicitly a similarity matrix $S$. All pairs $(i, j)$ are obtained by $k$ random walks of length $l$ starting from each of the $n$ nodes. The complexity of obtaining the input data is therefore in $O(n)$ ($l$ and $k$ being small constants).

Node2vec
Node2vec is a popular variant of DeepWalk, introducing biased random walks. Two parameters guide the random walks: $p$ decreases the probability to revisit the previous node, while $q$ decreases the probability to explore farther nodes, i.e., nodes that were not neighbors of the origin node. It allows to mimic breadth-first or depth-first like exploration of the graph, capturing more local or more global network structures.

 questões

DeepWalk complexity
Contrary to matrix decomposition based approaches, DeepWalk does not require explicitly a similarity matrix $S$. All pairs $(i, j)$ are obtained by $k$ random walks of length $l$ starting from each of the $n$ nodes. The complexity of obtaining the input data is therefore in $O(n)$ ($l$ and $k$ being small constants).

Node2vec
Node2vec is a popular variant of DeepWalk, introducing biased random walks. Two parameters guide the random walks: $p$ decreases the probability to revisit the previous node, while $q$ decreases the probability to explore farther nodes, i.e., nodes that were not neighbors of the origin node. It allows to mimic breadth-first or depth-first like exploration of the graph, capturing more local or more global network structures.

Node2vec
Illustration of random walk procedure in node2vec. (Figure from (Grover and Leskovec 2016)

The walk just transitioned from $t$ to $v$ and is now evaluating its next hop. Edge labels indicate the bias as a function of parameters $p$ and $q$.

Role2Vec — Struc2Vec
Two popular methods for role embedding are Struc2Vec and Role2Vec. They are based on a similar principle: as DeepWalk, they use random walks and SkipGram to generate embedding from contexts. But instead of generating sequences composed of the labels of encountered nodes, it generate contexts based on the attributes/features of encountered nodes. Nodes with similar vectors thus corresponds to nodes that tend to encounter nodes with similar properties in random walks starting from them.

Examples of properties could be node features (age, genre, etc.) or structural properties (degree, clustering coefficient, graphlet belonging, etc.).

Link Prediction with embeddings: supervised
In the second approach, we consider each dimension of the embedding as a node feature. For each pair of nodes, we compute a vector by combining nodes’ vectors.

As with heuristics, a machine learning algorithm is then trained to predict, from the combined vector, how likely it is to have an edge between nodes.

Combining node vectors
There are several methods to combine node vectors. Although it has been observed empirically that the Hadamard product often gives the best results, this choice is often considered a hyper-parameter. Two variants are tested and the most efficient is used for the final prediction.

The most used operators are:

<table>
<thead>
<tr>
<th>Average</th>
<th>$(a+b)/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concat</td>
<td>$(a_1, a_2, ..., a_d, b_1, b_2, ..., b_d)$</td>
</tr>
<tr>
<td>Hadamard</td>
<td>$[a_1 \cdot b_1, a_2 \cdot b_2, ..., a_d \cdot b_d]$</td>
</tr>
<tr>
<td>Weighted L1</td>
<td>$</td>
</tr>
<tr>
<td>Weighted L2</td>
<td>$(a_1 - b_1)^2, (a_2 - b_2)^2, ..., (a_d - b_d)^2$</td>
</tr>
</tbody>
</table>

where $a = [a_1, a_2, ..., a_d]$ and $b = [b_1, b_2, ..., b_d]$

How many dimensions?
There is no universal method to choose a number of dimensions for the embedding. In the literature, for large graphs, a common value is $d = 128$ dimensions. As a general rule, $d < n$.

Too few dimensions limit the capacity to embed complex information, but too many dimensions limit cross-learning, generalization, and make learning from embeddings harder. More dimensions also require (usually) more computation.

Visualization and embeddings
Network visualization is a domain in itself. Its objective is to assign positions to nodes in a two dimensional space in order to plot the network in a meaningful way.

Algorithms such as HOPE or node2vec are not well adapted to generate visually interpretable 2-dimensional spaces, in part because the distance in the embedding is based on the cosine distance, while humans naturally assume euclidean distance. When embeddings are used for visualization, the first step consists in embedding in a moderate number of dimensions (e.g., 128), and in a second step, a dimensionality reduction algorithm more adapted for visualization such as T-SNE is used to reduce this number to 2 dimensions.

Node Classification with embeddings
Machine Learning algorithms such as Logistic Regression or Decision Tree can be trained to predict a property of a node from a vector of features representing the node property. We have seen in a previous class that these features could be manually chosen heuristics such as node centralities.

Vectors yielded by embedding algorithms can naturally be used in the same way. Locational embeddings could be used, for instance, to attribute category to objects or political opinions to social media accounts, while role embedding could be used to identify suspicious accounts in social media.

Link Prediction with embeddings: unsupervised
If we consider that the property captured by the embedding is correlated with the probability of being connected by an edge, then the distance in the embedding can be used as a heuristic for link prediction.

For instance, with LE and HOPE with $S = A$ or with random walks based approach, the embedding tries to put pairs of nodes connected by an edge closer than unconnected ones. As a consequence, we can assume that the closer two nodes are in the embedding, the more likely it is that they should be connected by an edge.

Visualization and embeddings
Network visualization is a domain in itself. Its objective is to assign positions to nodes in a two dimensional space in order to plot the network in a meaningful way.

Algorithms such as HOPE or node2vec are not well adapted to generate visually interpretable 2-dimensional spaces, in part because the distance in the embedding is based on the cosine distance, while humans naturally assume euclidean distance. When embeddings are used for visualization, the first step consists in embedding in a moderate number of dimensions (e.g., 128), and in a second step, a dimensionality reduction algorithm more adapted for visualization such as T-SNE is used to reduce this number to 2 dimensions.

$\alpha$Van der Maaten and Hinton 2008

Role Embedding
Node2Vec and DeepWalk are location embedding: nodes with similar vectors tend to be close in the graph, in term of graph distance.

Another notion of graph similarity is role similarity. Two nodes have similar roles in the graph if their neighborhood is similar, ignoring node labels.
Community detection with embeddings
Community detection in graphs is equivalent to the clustering task in non-network data. Intuitively, clustering methods try to group elements with similar features, and separate those that are different. Applying a clustering algorithm such as k-means on an embedding will therefore yield clusters of nodes, that can be considered as communities. In practice, it has been observed that communities detected by this approach are often similar to those found by modularity maximization. Note that unlike with modularity, it is often required to provide the desired number of clusters—or a distance scale—to clustering methods.

Going Further

References