Temporal Graph Embedding

Justin Mücke
justin.muecke@uni-ulm.com
University Ulm
Ulm, Germany

ABSTRACT

KEYWORDS

Temporal Graph, Embedding

ACM Reference Format:
Justin Mücke. 2021. Temporal Graph Embedding. In *University Ulm: Seminar Data Science.* ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION

ML finden Graphen ganz blöd! Drum machen wir jetzt vektoren *uwu*

2 PRELIMINARY

To start this paper out, let's first create a level playing field for all readers by laying out what exactly a Graph is, how it can change over time and what is meant when we talk about embedding.

2.1 Graphs

A graph is a mathematical construct which is used in a variety of tasks. It is often used to model relationships between entities, thus making it possible to operate on such structures to, for example, analyze them.

Mathematically it is consisting of two sets $G = (V, E)$, where

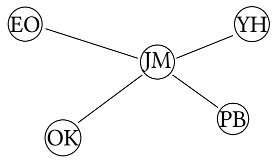

Figure 1: Part of the follower-relationship-graph of personal Instagram account.

V consists of all the vertices of the Graph, and E of all the edges, represented through a tuple of two vertices. Our graph above would thus look like following:
$G_i = (\{E, O, Y, P, J\}, \{(J, E), (J, O), (J, P), (J, Y)\})$.

2.2 Temporal Graphs

A temporal graph, on the other hand, is a graph that changes its structure over time. This happens when either one of our sets (V, E) changes. Our graph G is then represented through $G = \{g_1, g_2, \dots, g_t\}$ where g_i is the static graph after time $i * \Delta t$. [1]

2.3 Embedding

Embedding is a process in which the graph $G = (V_g, E_g)$ is transformed into a set of vectors V. These capture the graph topology, vertex-to-vertex relationship as well as other relevant information. This way, it is more accessible for analyzing the graph and comparing it to others. Overall we can divide graph embedding techniques into two categories; vertex embedding and graph embedding.
When using vertex embedding, there has to be one vector v for each node in G, so that $|V| = |V_g|$. This is used to make prediction on a node level.
When using graph embedding, there is one vector representing the whole graph. This method is most useful to analyze the data on a graph level e.g. comparing two graph structures with each other.
In order to achieve this transformation there are several methods available.

2.3.1 Word2Vec. The first method is word2vec which is the foundation for many other methods. It takes a text $T = (w_1, w_2, \dots, w_n)$ as input and returns a set of vectors$V$, where v_i is a distribution describing how likely it is for each word to be a direct neighbor to it. The neighbors of a word w_c are defined as $\{w_i | i \in (c - \epsilon, c + \epsilon)\}$ where ϵ describes a predefined window size. With this, the whole text is represented solely through vectors, where similar words have similar vectors. One way to achieve this, is to train a neural network to create the distributions. This neural network consists of one input-, one hidden- and one output layer. Each word in our text is assigned an ID through a one-hot coded vector, so that each word w_i can be represented like $(0_1, 0_2, \dots, 0_{i-1}, 1_i, 0_{i+1}, \dots 0_n)$. It then computes how likely it is for each word to be its neighbor and transforms it into a distribution using a softmax-function[2].

2.3.2 DeepWalk. This method is a continuation of the word2vec approach, but now the input is not a text, but a graph $G = (V, E)$ It consists of three steps. The first being the sampling, where random walks are performed from each node. Hereby, a random walk is a path in the graph from a starting point v_i of defined length λ. The following nodes are generated by:

$$P(v_i = x | v_{i-1} = y) = \begin{cases} \frac{\pi_{xy}}{Z} & \text{if}(x, y) \in E \\ 0 & \text{otherwise} \end{cases}$$

where π_{xy} is the probability between nodes x and y with$Z$ as the normalizing constant.
It is sufficient to perform $32 - 64$ random walks per Node. The resulting paths are of the same structure as sentences in a text, so

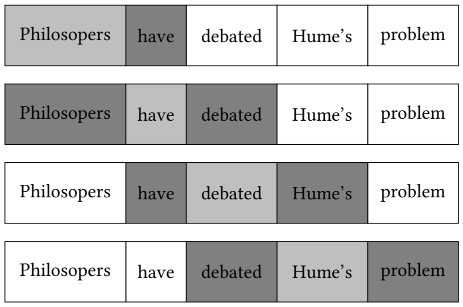
Philosopers have debated Hume's problem
Philosopers have debated Hume's problem
Philosopers have debated Hume's problem
Philosopers have debated Hume's problem

Figure 2: Light-gray word, with dark-gray neighbor and $\epsilon = 1$.

the word2vec method can now be used on it. The result is a set of vectors V where each vector v_i is a distribution of the probability that two nodes are next to each other.

2.3.3 Node2Vec. This is an optimized version of DeepWalk. Again, there is the sampling phase in the beginning. The difference in the methods is, that in node2vec the random walks are now biased and not completely random. An order for the random walk is defined through two parameters p and q. To evaluate which node to visit next the walk looks at the transition probability π_{xy} on the edge (x, y). This probability is now defined as $\pi_{xy} = \alpha_{pq}(x, y) * w_{xy}$ where w_{xy} is the weight assigned to the edge (x, y) and:

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

with d_{xy} being the shortest path between nodes x and y. Hereby, our parameter p defines how likely it is to revisit a node of the walk. Setting p to a higher value encourages the walk to go deeper into the graph, whereas a small value ensures the walk to stay local around the starting point. In contrast, q describes the preference of nearer or further nodes. For $q > 1$ the walk favors closer nodes, for $q < 1$ nodes that are further away.[3]

2.3.4 Structural Deep Network Embedding. In contrast to the methods used before, SDNE does not use random walks. It aims to preserve local pairwise similarity which characterizes the local structure, and as well as the global network structure. To achieve this, we use two autoencoder neural network. These get an adjacency vector as input and want to construct node adjacency as output. We then compute the distance between the two outputs and add it to the loss function of the network. The total loss function is then computed through summation of the distance loss plus the losses of the two encoders. At the end we remain with a collection of adjacency vectors which describe the graph structure.

2.3.5 Graph2Vec. Now we don't want to represent the nodes as vectors, but the whole graph. For that we have once again three steps. In the first step we create sub-graphs for each node and encode then once again in a one-hot code. We then use these sub-graphs to train the network used in word2vec to maximize the probability that a predicted sup-graph exists in the input graph. The embedding is then the result of the network. [4]

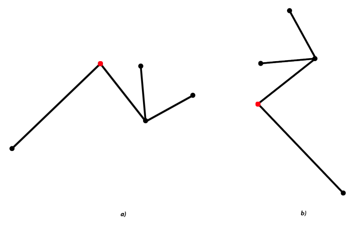

Figure 3: a) graph desrcibed by first embedding. b) graph described by second embedding. The red node do not align even though the graph structure is the same.

3 EMBEDDING OF TEMPORAL GRAPHS

Let $G = (V, E)$ be a temporal graph with $G_t = \{G_{t_1}, \ldots, G_{t_T}\}$ as its evolution over the time steps T. The goal is now, to create a vector space which can be used to analyze the graph to find anomalies in it, compare it to other graphs, classify nodes or predict if certain links are going to exist. Depending on which outcome is wanted, different methods have to be used to optimize it. For predictions on the node level like node classification and link prediction a node-level algorithm should be used. On the other hand, a graph-level algorithm should be used to compare graphs with each other and to find anomalies, as shown in 3.3.

3.1 tNodeEmbed

As aforementioned, a node-level algorithm should be used to perform e.g. link predictions and node classification. One proposed algorithm of that form is *tNodeEmbed.* Here, the embedding can be split into three parts:

3.1.1 Initialization. To start out, the algorithm initializes a representative vector $Q_t \in \mathbb{R}^{T \times d}$ for each node v in all graphs $G_{t_1}, \ldots, G_{t_T}$, where d is the embedding size, and T the number of time steps. These vectors are created using the node2vec algorithm discussed in 2.3.3.

3.1.2 Node alignment. A downside to node2vec is that, when running on a graph, it aims to minimize the word embedding distances but not consistency over multiple trainings. The resulting embedding could be in a two-dimensional space for example. It lays in a x-y-plane. Now imagine a rotation around the unused z-axis. The resulting graphs have the same structure but different orientation in the plan, thus not having an alignment, as seen in figure 3. Similarly, when embedding two graphs G_{t_i} and G_{t_j} it is not guaranteed that, even if the graphs are identical, their axes align. To adjust those embeddings, tNodeEmbed uses an orthogonal transformation between embeddings at two time points t_i and t_j[5]. Used in this algorithm, it takes the matrix $Q_t \in \mathbb{R}^{d \times |V|}$ of node embeddings at time t. Then the matrices get aligned iteratively starting from the earliest timestep. For the alignment an orthogonal matrix R between Q_t and Q_{t+1} is needed to result in the final embedding $Q'_t = RQ_t$. The matrix are is approximated by :

$$R_{t+1} = \arg\min_{R s.t. R^T R = I} \|RQ_{t+1} - Q_t\|$$

where $R_{t+1} \in \mathbb{R}^{d \times d}$ is the transformation which fits the time steps the most.

3.1.3 *Finalization.*

IGNORE FOR NOW. The algorithm works in such a way, that the only difference in the algorithm, between the two outcomes, is the loss function. For the node classification task it uses a categorical cross-entropy loss:

$$L_{task} = -\sum_{v \in V} \log Pr(class(v)|f_T(v))$$

For the link prediction task it considers a binary classification loss:

$$L_{task} = -\sum_{v_1, v_2 \in V} \log Pr((v_1, v_2) \in E|g(f_T(v_1), f_T(v_2)))$$

[6]

3.2 tdGraphEmbed

Starting paper [7]

4 COMPARISON

How do Methods differ -> nodelevel / Graphlevel?

5 APPLICATION

Why do we use Embedding

5.1 Similarity

Differences Between graphs (exp - googletrends)

5.2 Anomaly

Where does it differ

6 CONCLUSION

REFERENCES

[1] G. B. Mertzios, H. Molter, R. Niedermeier, V. Zamaraev, and P. Zschoche, "Computing maximum matchings in temporal graphs," *CoRR*, vol. abs/1905.05304, 2019.
[2] B. Gao and L. Pavel, "On the properties of the softmax function with application in game theory and reinforcement learning," 2018.
[3] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks."
[4] P. Godec, "Graph embeddings — the summary - towards data science," *Towards Data Science*, 31.12.2018.
[5] P. H. Schönemann, "A generalized solution of the orthogonal procrustes problem," *Psychometrika*, vol. 31, no. 1, pp. 1–10, 1966.
[6] U. Singer, I. Guy, and K. Radinsky, "Node embedding over temporal graphs," *IJCAI*, pp. 4605–4612, 2019.
[7] M. Beladev, L. Rokach, G. Katz, I. Guy, and K. Radinsky, "tdgraphembed: Temporal dynamic graph-level embedding," ACM, 2020.