SDM Graph Embedding Project

XIMENA MOURE ALASSIO and THOMAS GUENTHER SCHMIDT (Erasmus)

Task 1: Summary about Graph Embeddings

Introduction

Graph databases are a data management technology that is designed to store and manage data about entities and their relationships. They are able to handle large and interconnected datasets for example in the domains of social networks. Graphs can be analyzed by means of graph analytics to extract insights and discover patterns, for example by performing node classification, node recommendation or link prediction. Graph embeddings are used in order to facilitate and speed up graph analytics tasks compared to traditional distributed graph analytics (Cai et al., 2018). They are representing the graph in a lower-dimensional vector space while preserving its structure and semantics and thereby improving the computational cost of graph analytics. This summary will provide a short overview over graph embeddings. It will summarize parts of the literature on types, techniques, applications, and evaluation of graph embeddings as well as open questions about them.

Types of Graph Embeddings

Graph embedding encompases various types, leading to diverse input and output representations. Depending on the specific graph type we aim to generate, the output can consist of specific elements such as edges, nodes or even the entire graph. This variability arises from the different types of graph embedding available.
The choice of embedding input type depends on the specific problem being addressed. We may consider different types of embedding inputs:

- Homogeneous graph: all nodes and edges share the same type. The simplest form of graph embedding involves an undirected and unweighted homogeneous graph as the input. However, incorporating weights and directions in the graph provides additional information and improves accuracy.
- Heterogeneous graph: involves graphs with diverse types of nodes and/or edges. They allow for rich and meaningful representations of the underlying complex relationships within the graph.
- Graph with auxiliary information: incorporates additional attributes or features associated with nodes or edges in the graph during the embedding process. The auxiliary information provides supplementary context or characteristics that can enhance the quality of the resulting embeddings.
- Graph constructed from Non-relational Data: the data may not have explicit relationships or connections, but through appropriate embedding techniques, valuable insights and representations can be derived.

The resulting embedding graph is derived from the input graph and can take one of four different forms:

- Node embedding: each node is assigned a unique vector representation. It aims to maintain the graph structure, ensuring that nodes that are proximate in the original graph are also close to each other in the embedding space. They are widely used in tasks such as node classification, recommendation systems, etc.
- Edge embedding: each edge is represented as a low-dimensional vector. They offer significant value across diverse graph analysis tasks, such as edge classification, link prediction, etc. They facilitate a deeper understanding of the relationship between nodes. Moreover, they can be used to infer missing edges and predict future connections within the graph.
- Hybrid embedding: combination of different types of graph components like node and edge.
- Whole graph embedding: as its name indicates it is the embedding of the whole graph. It is used for small graphs. It aims to capture the global structure, patterns and characteristics of the entire graph in a compact representation. Graph classification problems are a typical application where graph embedding proves useful.

Graph Embedding Techniques

Graph embedding methods can be categorized into three broad groups: matrix factorization-based methods, random-walk-based methods and deep learning-based methods.

1) Matrix factorization-based methods for graph embedding can be categorized into two types. The first type is Graph Laplacian Eigenmaps, which represents a graph using a Laplacian matrix and aims to minimize the error between assigned values and the Laplacian matrix. It excels at capturing the local neighborhood structure, is unsupervised, and produces easily interpretable embeddings. However, it is sensitive to changes in the graph's topology and has increasing computational complexity with graph size. The second type is Node Proximity Matrix Factorization, which aims to accurately capture the structure of the original adjacency matrix. However, the choice of proximity measure can impact the quality of the embeddings.

2) Random walk methods: the goal is to estimate the embedding of each node while capturing the underlying connectivity patterns of the graph.
   In a random walk, the process involves randomly selecting a neighbor of a node and moving to that neighbor. This procedure is repeated by randomly selecting neighbors at each step until a maximum length is reached.
   The similarity between two nodes is determined by their co-occurrence in random walks. The co-occurence is quantified using conditional probability, specifically the probability of visiting node v given that we started from node u.

3) Deep Learning: primarily rely on neural networks, which learn a mapping function that transforms a graph represented in numerical form into a low-dimensional embedding. This process involves optimizing over a diverse set of expressive neural networks functions. Deep learning models provide an efficient means of extracting embeddings by introducing novel techniques for approximating convolutions and kernels in traditional graph representations.
Deep learning models provide an efficient means of extracting embeddings by introducing novel techniques for approximating convolutions and kernels in traditional graph representations.

Applications of Graph Embeddings

Graph embedding has experienced significant growth in its applications and it continues to grow. This powerful modeling tool finds utility in various domains, including but not limited to:
e9

- Node classification: it is used to predict the labels or classes of nodes in a graph. The node embeddings can be used as input features for machine learning algorithms to perform node classification. The following are some examples of node classification tasks where graph embedding techniques have been used:
    - Social Network Analysis: it can be used to predict user interests, identify communities or groups and detect influential users within the network.
    - E-commerce networks: to predict customer preferences, segment customers into specific groups or detect fraudulent behavior by identifying anomalous nodes.
    - Biological networks: graph embedding techniques have been applied to biological networks such as protein-protein interaction networks or gene co-expression networks. Node classification in these networks involves predicting protein functions or identifying disease-related genes.

- Link detection: it aims to predict or identify missing or future connections between nodes. The following are some examples of how graph embedding is applied in link detection:
    - Social Networks: to predict social connections between users. By representing the nodes of the network in a lower-dimensional space, the algorithm captures the similarity or proximity between nodes and identifies probable links that have not yet been established.
    - Recommender Systems: by learning the representations of users and items in a graph, the algorithm can identify potential associations and recommend items to users.
    - Brain networks: embedding brain networks into a lower-dimensional space facilitates their visualization which allows researchers to gain insights into the spatial organization and connectivity patterns. Also, graph embedding techniques can be applied to extract discriminative features from brain networks, enabling the classification and diagnosis of neurological disorders.

Evaluation of Graph Embeddings

Currently, there is no established framework or baseline to consistently evaluate graph embeddings for generalizable results. They are evaluated on an ad-hoc basis because each method stipulates different assumptions about the input data. This means that there is no widely accepted benchmark dataset or agreed upon metrics. Moreover, performance is usually evaluated on a small number of graphs which leads to low generalizability of the results (Goyal et al., 2019). Oftentimes available graphs such as the Twitter Graph or Wikidata are used to evaluate embedding techniques. This is a problem because it has been shown that the performance of graph embeddings is dependent on several factors such as graph domain, size of the graph, graph density, embedding dimension, and evaluation metric. A selection of knowledge graphs can be found in the Stanford Network Analysis Project (SNAP).

Looking at the ad-hoc evaluation it can be noted that it is done in a supervised manner comparable to other data mining techniques. Depending on the task, a subpart of the dataset is selected and split into a train and test set. For the test set, the task specific information is removed. Then, e.g. a classifier is trained on the train set and evaluated (via a performance metric) on the test set. Depending on the task different performance metrics are used (Cui et al., 2019):

- Node classification: Micro-F1 and Macro-F1
- Link prediction: Precision @k and Mean Average Precision (MAP)
- Clustering: Accuracy (AC) and normalized mutual information (NMI)

In an effort to promote a more generalizable evaluation of graph embeddings (Goyal et al., 2019) propose an evaluation framework that evaluates the performance over a wide range of graphs with different properties. However, they only focus on the use case of link prediction.

In summary, benchmark datasets, evaluation metrics, and evaluation procedures have to be established and standardized in order to be able to generalize and compare the results of different graph embedding techniques. However, one should keep the downstream use case of the embeddings in mind. If they are only used to predict fraudulent transactions in a payment graph, then an evaluation focusing on only this task and the specific domain is usually sufficient.

Future Directions

Computation and dynamic embeddings: First there is research towards more efficient and scalable techniques for the computation of graph embeddings. Methods like GPU optimization do not work well on graphs because of their structural information that makes it hard to present them in matrix form (Cai et al., 2018). Current techniques however, fail to scale to large graphs. In order to address this problem new techniques need to be developed. Moreover, current embedding techniques often assume static graphs, i.e. graphs that are not modified. In real world scenarios however, nodes and relationships are added and modified frequently which renders the previous embedding less relevant. As

outlined before, the computation of embeddings is costly. Thus, one research stream is focusing on scalable and incremental techniques to compute graph embeddings (Cai et al., 2018; Cui et al., 2019; Goyal & Ferrara, 2018).

Graph structure: Current embedding techniques often focus on pairs of nodes to learn embeddings and thereby omit relevant structural information. Capturing non-pairwise structure could bring information about the centrality of a node into the embedding (Cai et al., 2018). Deep-learning based methods already consider more complex structures, however they are not very computationally efficient. Additionally, a graph might contain hyperedges whose embeddings are an avenue for future research. Lastly, addressing the challenge of learning effective embedding for nodes that only participate in a few relationships is being addressed (Cui et al., 2019).

Applications: Future research is and should be addressing new applications for graph embeddings. For example, making use of embeddings for cross modal retrieval is promising (Cai et al., 2018). By learning embeddings for e.g. images and texts at the same time keyword retrieval tasks or image/video classification could be greatly improved. Moreover, research should explore application specific embedding techniques that possibly outperform general purpose embedding methods by including domain knowledge (Cui et al., 2019). The challenge here is to model and include the domain knowledge in the embedding.

Interpretability & Explainability: Being able to interpret and explain graph embeddings is of high importance. If embeddings contain social biases these could greatly affect the downstream decision that is being made upon the graph analytics tasks, raising social and ethical issues. Additionally, researchers and practitioners might benefit from knowing how a characteristic of a graph is represented in the embedding. While this is possible for some methods, especially deep-learning-based methods currently lack this explainability (Goyal & Ferrara, 2018).

Task 2: Application of Graph Embeddings

Graph Description and Use Case

For our use case we selected the CORA dataset. The dataset describes a citation network of scientific papers. Papers belong to one of 7 classes (topic classification) and have a feature vector indicating which of 1433 different keywords are contained in the paper. We used a subset of the data which we got from https://graphsandnetworks.com/the-cora-dataset/.
The goal of the use case is to perform node classification, i.e., classifying each paper to one of the seven topics. This is a multi-class classification problem, meaning that a paper is assigned to one out of the 7 classes.

Embedding Techniques

We decided to use two different embedding techniques with different features to compare their results (both create node embeddings):

- Graph Convolutional Network: A graph convolutional network can take graph structure as well as node features of the neighboring nodes into account for the embedding
- Node2Vec: Another popular embedding technique based on random walks. Takes graph structure into account, but can't use node features.

Implementation:
We used python to create the embeddings and apply them to our use case of node classification.

GCN

The GCN was implemented using the pytorch.geometric package. We decided on the following Structure 1433 - 16 - 7. The number of input nodes is determined by the shape of our feature vector, which in our case has 1433 elements which represent the keywords present in the paper. This is then mapped down to 16 neurons. The number of output neurons is determined by the number of classes we have. The following table presents more details about the convolutional layers.

Input Layer	#Neurons: 1433 Aggregation function: add Activation function: Relu
Layer 2 (Hidden Layer)	#Neurons: 16 Aggregation function: add Activation function: Relu
Output Layer	#Neurons: 7 Aggregation function: add Activation function: log softmax

Loss function	Cross-entropy loss

We trained the GCN with a cross entropy loss function and a batch size of 70 in the gradient descent. We had a total of 280 train samples (40 per class).

The embeddings of the GCN is hard to describe because the model outputs a probability for the node being from a class in the output layer. The embeddings are basically “contained” in the weights of the GCN and the neighboring node information. The activation of the hidden layer can also be accessed to be used as embedding for a node.

Node2Vec:

To implement the solution, we utilized NetworkX and Karate. The former is a Python package that enables the creation and manipulation of intricate network structures and the latter is an unsupervised machine learning extension designed for NetworkX.

We ran the node2vec algorithm with a breadth-first approach, because we considered closer nodes (i.e. closer in the citation chain) more important for the classification than the citations further down the stream.

Node2Vec algorithm requires tuning hyperparameters such as the length of the random walks, the number of random walks per node, and the context window size.

The following are the parameters that we choose:

‘p’: it controls the likelihood of the random walk to stay in the same neighborhood (breadth-first search) or to explore further (depth-first-search). We decided to use p=5 because of the reason mentioned at the beginning of the section.

‘q’: it balances the likelihood of the random walk to visit nodes that are already visited or nodes that are unvisited. We set q=2 to bias the random walk towards local nodes.

‘window_size’: it determines the size of the sliding window used to generate node context pairs for the skip-gram model. We chose window_size=5 so that the algorithm will consider the five nodes before and the five nodes after the target node as its context for generating node context pairs during the training.

‘walk_length’: it represents the length of each random walk performed on the graph. Longer random walks capture more global structure but may sacrifice local information. We chose walk_length = 25.

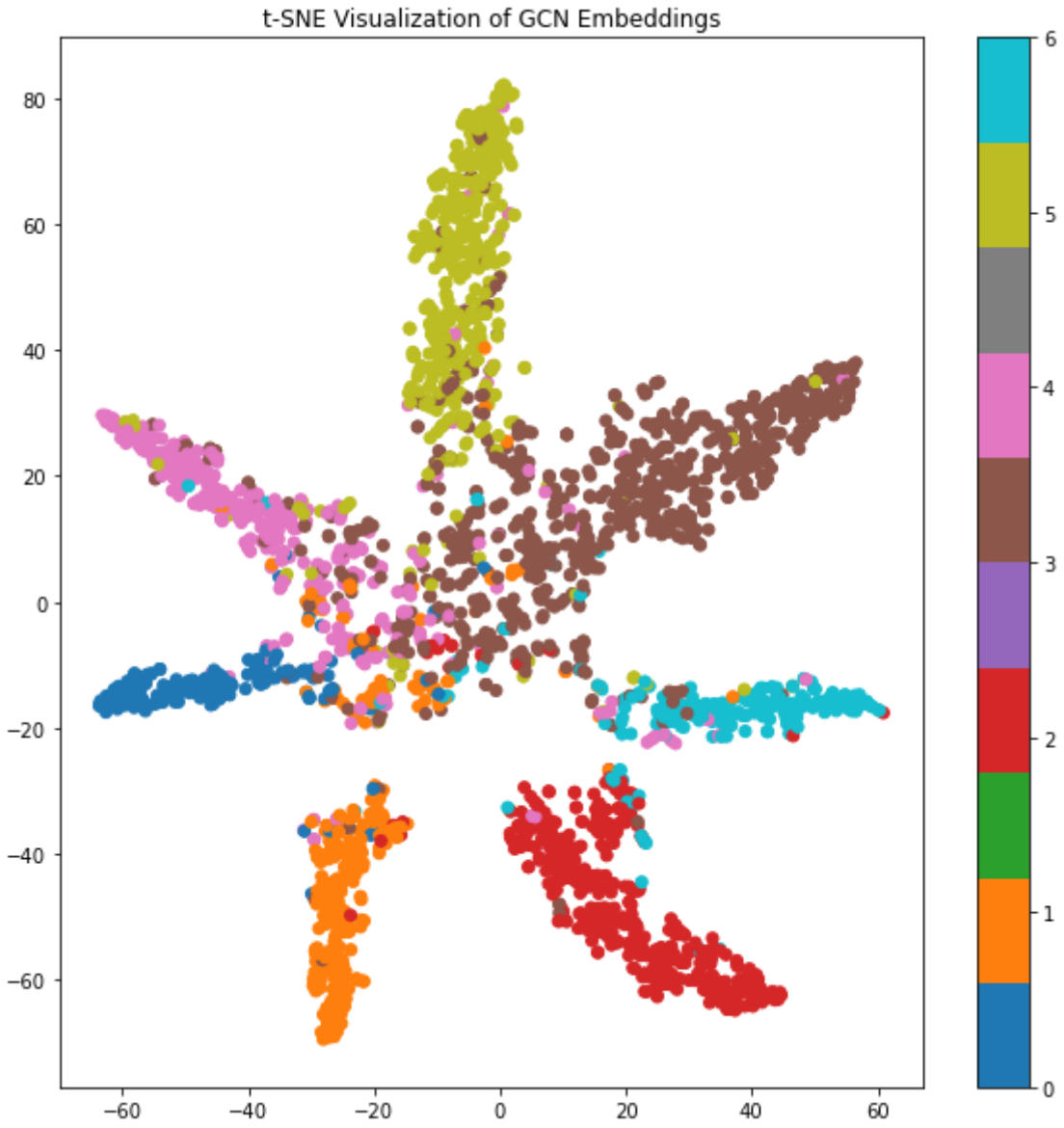
Comparison of techniques

Without further hyperparameter optimization we achieved the following results:

	GCN	Node2Vec
Micro-F1	0,80	0,69

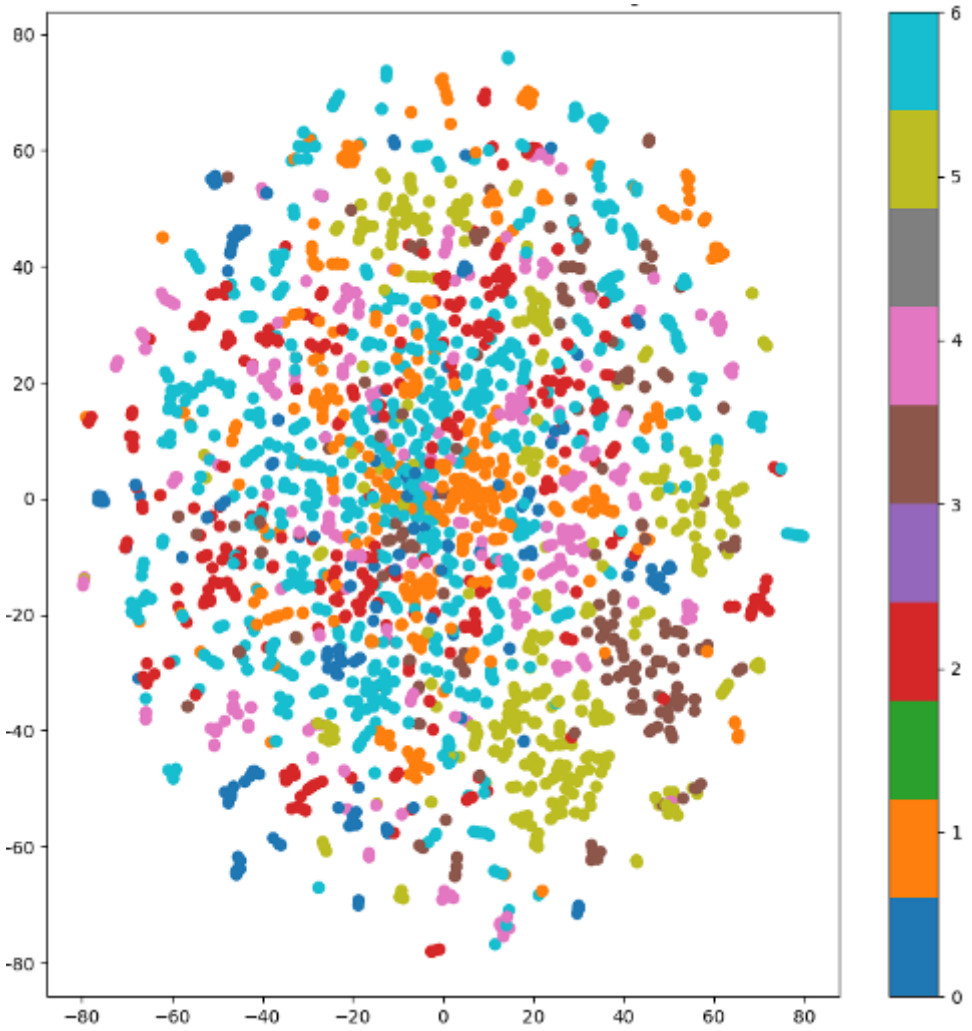
We can see that the GCN performed better than Node2Vec on our downstream task. This might be because Node2Vec only takes the graph structure into account, while the GCN also has access to the node features. In our case the node features were already present, but usually this would also come with a feature engineering effort which however likely yields better results.

We also performed a t-SNE dimensionality reduction to two dimensions in order to visualize the result. For the GCN (output layer embeddings) we see that similar nodes are in clusters, however there is some overlap into other clusters.



For Node2Vec we can see that there are no clear clusters. This may be due to the fact that the performance of node2vec is highly sensitive to its hyperparameters, thus, suboptimal parameter settings may result in less effective capturing of cluster separability. Furthermore, the reliance on random walks in Node2Vec introduces an element of randomness, which can result in different outcomes for each run. This variability can affect the formation and separability of clusters, leading to less distinct visual patterns. Additionally, GCN typically generates node embeddings of higher dimensionality compared to Node2Vec. Higher-dimensional embeddings may have more expressive power to capture finer-grained details and cluster separability.

t-SNE Visualization of Node2Vec embeddings



Summary and conclusions

Throughout this research-oriented project, we had the opportunity to extensively review and provide an explanation of Graph Embeddings. We discussed the underlying mechanisms and workings of this technique and explored its various variants. Additionally, we implemented Graph Embeddings and gained hands-on experience using them.

Bibliography

Cai, H., Zheng, V. W., & Chang, K. C.-C. (2018). *A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications* (arXiv:1709.07604). arXiv. http://arxiv.org/abs/1709.07604

Cui, P., Wang, X., Pei, J., & Zhu, W. (2019). A Survey on Network Embedding. *IEEE Transactions on Knowledge and Data Engineering*, *31*(5), 833–852. https://doi.org/10.1109/TKDE.2018.2849727

Goyal, P., & Ferrara, E. (2018). Graph Embedding Techniques, Applications, and Performance: A Survey. *Knowledge-Based Systems*, *151*, 78–94. https://doi.org/10.1016/j.knosys.2018.03.022

Goyal, P., Huang, D., Goswami, A., Chhetri, S. R., Canedo, A., & Ferrara, E. (2019). *Benchmarks for Graph Embedding Evaluation* (arXiv:1908.06543). arXiv. http://arxiv.org/abs/1908.06543

Further References

https://www.tigergraph.com/blog/understanding-graph-embeddings/
https://arxiv.org/abs/2012.08019
https://towardsdatascience.com/graph-embeddings-explained-f0d8d1c49ec
https://medium.com/@st3llasia/graph-embedding-techniques-7d5386c88c5
https://arxiv.org/pdf/1705.02801.pdf