Universitat Politècnica de Catalunya

Semantic Data Management

Real use cases of property/knowledge graphs

Exploitation via data analysis

Enric Reverter
enric.reverter@estudiantat.upc.edu

Míriam Méndez
miriam.mendez.serrano@estudiantat.upc.edu

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Curs 2022/2023 Q2

Contents

1 Graph Embeddings

In many domains, from social networks to biological systems, complex interactions and relationships are pervasive. Traditional analytical methods often fall short in capturing the intricate dynamics and uncovering meaningful patterns within such complex systems. For that, graph embedding approach (Fig. 1) emerged as a powerful tool for studying and comprehending these intricate networks. Some applications are node classification, node clustering, node retrieval/recommendation, and link prediction, for example. Nonetheless, the high computation and space cost of such approach gave rise to an efficient strategy, graph embedding. An embedding is a numerical representation of data in a continuous vector space.It is a way to transform high-dimensional and discrete data to a lower-dimensional and continuous space. In terms of graphs, it is a way to capture the structural and relational information into a low-dimensional continuous space. There are different approaches depending on the available inputs and desired outputs. In a survey conducted by Cai et al. in 2018 [1] the possible input typings are classified within: homogeneous graph, heterogeneous graph, graph with auxiliary information, and graph constructed from non-relational data. Regarding the output, which is disconnected from the type of input, they argue it can be classified in: node embedding, edge embedding, hybrid embedding, and whole-graph embedding.

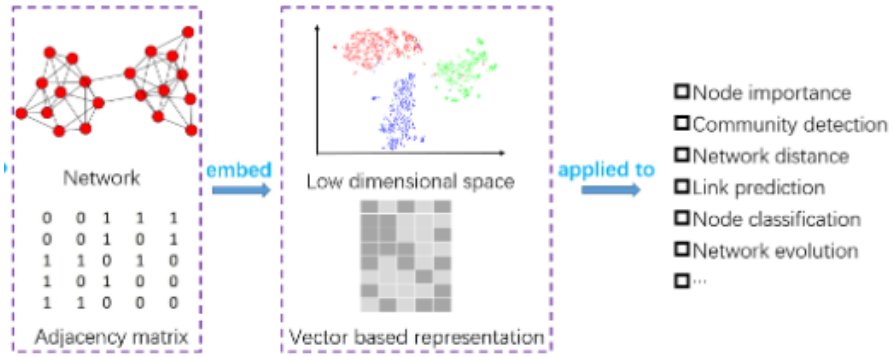


Figure 1: Pipeline of analytical analysis using graph embedding

1.1 Graph Embedding Input

The survey [1] discusses the following four categories of input graphs for graph embedding.

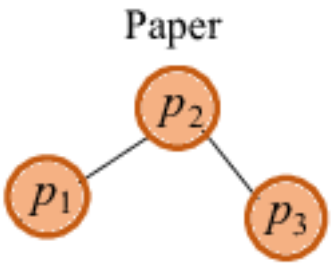
Homogeneous graph: (Fig. 2a) consists of nodes and edges of the same type. The most basic embedding input setting is the undirected and unweighted, since only structural information of the input graph is available. However, weighted and directed graphs provide additional information that can improve the accuracy of graph representation in the embedded space. In the weighted graph, nodes connected by higher-weighted edges are embedded closer to each other. In the directed graph, the direction information of the edges are preserved in the embedded space. Challenges in homogeneous graph embedding include capturing connectivity patterns observed in the graph and preserving them during embedding.

Heterogeneous graph: (Fig. 2b) involve different types of nodes and edges. Examples include community-based Question Answering (cQA) sites, multimedia networks, and knowledge graphs. Each type of object (node or edge) is embedded into the same space, requiring exploration of global consis-

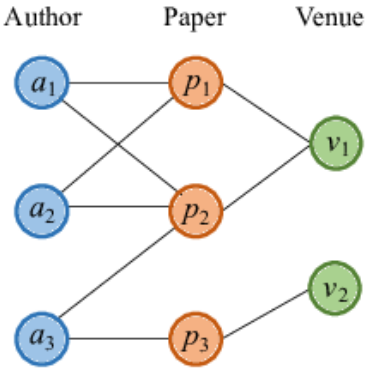
tency between different object types and addressing imbalances among object types. The challenge lies in combining different information sources to define node similarity and considering data skewness in embedding.

Graph with Auxiliary Information: (Fig. 2c) this type includes graphs with additional information such as labels, attributes, node features, information propagation, and knowledge bases. Labels and attributes help define node similarity, while node features (e.g., text or image features) provide unstructured information. Information propagation refers to the paths through which information spreads in a graph. Knowledge bases provide background knowledge and facts. Challenges in embedding graphs with auxiliary information involve incorporating rich and unstructured information while preserving topological structure and discriminative characteristics based on the auxiliary information.

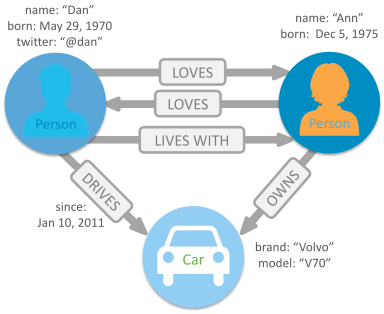
Graph Constructed from Non-relational Data: these types are constructed from non-relational input data using various strategies. One approach involves constructing a similarity matrix based on the features of the input data and treating it as the adjacency matrix of the graph. Another approach is to establish edges between nodes based on co-occurrence or other relationships. Challenges in embedding graphs constructed from non-relational data include computing relations between instances and preserving the generated node proximity matrix in the embedded space.



(a) Homogeneous graph



(b) Heterogenous graph



(c) Graph with auxiliary information

1.2 Graph Embedding Output

The output of graph embedding can be categorized into four following types. Each type of embedding serves different purposes and facilitates various applications.

Node Embedding: each node in the graph is represented as a vector in a low-dimensional space. Nodes that are close in the graph are embedded to have similar vector representations. The challenge lies in defining pairwise node proximity and encoding it in the learned embeddings. Node embedding is useful for tasks such as node clustering and classification.

Edge Embedding: aims to represent edges in the graph as low-dimensional vectors. This type of embedding is beneficial for tasks like link prediction and knowledge graph entity/relation prediction. The challenge in edge embedding is defining edge-level similarity and modeling the asymmetric property of edges, if any.

Hybrid Embedding: involves embedding a combination of different graph components, such as node + edge or node + community. It can be used to embed substructures or communities within the

graph. Challenges in hybrid embedding include generating the target substructure and embedding different types of graph components in one common space.

Whole-Graph Embedding: represents an entire graph as a single vector. This type of embedding is suitable for small graphs and aids in tasks like graph classification. The challenge in whole-graph embedding is capturing the properties of the entire graph while balancing between the expressiveness of the embedding and the efficiency of the embedding algorithm.

1.3 Graph Embedding Techniques

Graph embedding techniques offer effective ways to represent graphs mentioned in Section 1.1 in the lower-dimensional spaces mentioned in Section 1.2, facilitating analysis and interpretation. Moreover, we can employ hybrid techniques by combining multiple of this methods, such as combining edge-based and attribute-based embeddings, optimizing embeddings using landmark nodes, and considering both structural and attribute-based losses. Additionally, methods based on distances to prototype graphs and approximation techniques are employed for efficient embeddings. The choice of technique depends on specific requirements, data characteristics, and computational considerations.

Matrix factorization: This technique is used in graphs constructed from non-relational data. Represents graph properties (e.g. node pairwise similarity) as matrix and factorize this matrix to obtain node embedding output by *graph Laplacian eignmaps* or *the node proximity matrix*. The first one, aim to preserve pairwise node similarities by penalizing larger dissimilarities between nodes that should be closer together. The second one, It approximates node proximity in a lower-dimensional space through matrix factorization, aiming to minimize the loss of approximation. However, this technique it inefficient for large graphs.

Deep Learning: This technique utilizes deep learning models on graphs and it embeds homogeneous and heterogeneous graph and graph with auxiliary information. It can output all four types mentioned in Section 1.2. This technique can be categorized into these two groups:

The *random walk* approach (e.g. DeepWalk, node2Vec),represents graphs as sets of random walk paths and can automatically exploit the neighbourhood structure through sampled paths on the graph. However,it only consider local context within a path and finding an optimal sampling strategy is challenging.

The *non-random walk* approach (e.g. GCN, struc2vec), applied directly deep learning methods to the entire graph or a proximity matrix. Autoencoders minimize reconstruction errors and capture neighborhood information. Deep neural networks, such as CNNs, are used either directly or adapted for graph data. Other types, like DUIF, HNE, and ProjE, utilize deep learning techniques for specific purposes. However, it is computationally expensive due to traditional deep learning architectures assume input data on a 1D or 2D grid structure to leverage GPU acceleration, which is not applicable to graph structures.

Edge reconstruction: This technique establish edges in the embedded graph that resemble the edges. It suits all the input graphs except for constructed from non-relational data and all the embedding outputs except the whole graph embedding. This is due to the complexity of reconstructing manual edges and the focus on local edges. That resemble of the edges of the input graph can be

done by *maximizing edge reconstruction probability*, *minimizing distance-based loss*, or *minimizing margin-based ranking loss*. This technique is more efficient than the techniques mentioned above but it sacrifice the ability to capture the global graph structure and, relying only on directly observed local information.

In *maximizing edge reconstruction probability* the objective is to maximize the likelihood of observing first-order and second-order proximities between nodes are calculated based on their embeddings.

In *minimizing distance-based loss* the objective is to preserve first-order and second-order proximity by minimizing the KL divergence between the proximity calculated from embeddings and the empirical proximity based on observed edges.

In *minimizing margin-based ranking loss* the objective is to ensure that a node's embedding is more similar to its relevant nodes than to irrelevant nodes. This is achieved by optimizing a margin-based ranking loss, considering the relevance between node pairs indicated by the edges in the input graph.

Graph Kernel: represents a graph as a single vector by decomposing it into substructures. Three types of substructures are commonly used: graphlets (induced subgraphs), subtree patterns (compressed labels based on relabeling iterations), and random walks. Graph kernels are primarily used for whole-graph embedding and are applied to homogeneous graphs or graphs with auxiliary information. They capture the global properties of the graph and compare graphs based on the inner product of their vector representations. This technique is the most efficient compared to the others as it only counting the desired atomic substructures in a graph. However, the substructures are not independent and also it faces challenges in dealing with redundant information and high-dimensional sparse embeddings as the substructure size increases.

Generative Model: are used for node and edge embedding tasks, particularly with heterogeneous graphs or graphs with auxiliary information, as they consider node semantics during embedding. They can either directly *embed the graph in a latent semantic space* or *incorporate latent semantics for graph embedding.*In the first approach, nodes are represented as vectors of latent variables, treating the graph as generated by a model (e.g. LDA) and the resulting embedding can be interpretable. The second approach leverages latent semantics to influence the embedding, considering both the graph structure and additional node information, offering the advantage of capturing information from different sources. This is useful, for example, to discover latent semantics in knowledge graphs or textual descriptions of entities and relations. However, the generative models make assumptions about the data distribution, which can be challenging to justify in practice. Additionally, the need for large amount of data for training can limit their applicability to smaller graphs or datasets with a limited number of instances.

1.4 Applications

There are many applications we can use in graph embeddings. These applications can be processed efficiently in terms of both time and space, thanks to the vector representations provided. In brackets is specified in the domain that are related.

Node classification (node): use the traditional classifiers such as Support Vector Machines (SVM), logistic regression, and k-nearest, to be trained on the embedded node features, in order to accurately

predict the class labels for unlabeled nodes. Moreover, there are advance frameworks that integrate graph embedding and node classification, in one step, optimizing and enhancing classification-specific representations for each node.

Node clustering (node): is an unsupervised algorithm that aims to group similar nodes together based on their characteristics. It can be applied the traditional clustering algorithms such as k-means. Moreover, there are advance frameworks that integrate graph embedding and node clustering, in one step, optimizing and enhancing clustering-specific node representation.

Node recommendation/retrieval/ranking (node): suggest the most relevant K nodes on specific criteria, such as similarity. This task is applicable in various domains, including research interests, customer item recommendations, etc. It is also useful in community-based question answering and proximity search, where nodes are ranked based on their relevance to a given query node. Additionally, cross-modal retrieval allows keyword-based searches for images or videos. In knowledge graph embedding, entity ranking is a popular application for prioritizing missing entities in triplets $< h, r, t >$.

Link prediction (edge): is a key application of graph embedding, where the goal is to infer missing links in an incomplete graph. Graph embedding techniques, such as DeepWalk, LINE, GCN, and struc2vec, encode network proximity and structural similarity in low-dimensional vectors. These vectors capture valuable information about the graph structure and can be leveraged for link prediction. While most research focuses on link prediction in homogeneous graphs, some approaches also address link prediction in heterogeneous graphs.

Triple classification (edge): It is an specific application in knowledge graphs that focuses on determining the correctness of unseen triplets $< h, r, t >$. The goal is to classify whether the relation between entities h and t matches the given relation r.

Graph classification (graph): assigns a class label to an entire graph, which is commonly used when graphs themselves are the unit of data, such as chemical compounds or protein structures. Graph embedding techniques are employed to calculate graph-level similarity, either by comparing sets of node embeddings or by decomposing graphs into substructures and assessing similarities between them. These approaches enable effective graph classification and similarity analysis, facilitating tasks such as chemical compound identification or protein structure recognition.

Visualization (graph): generates visual representations of graphs in a low-dimensional space. Usually, nodes are mapped to 2D vectors and plotted in a 2D space, allowing for a visual representation of node categories and their proximity. Fig. 3 depicts an example of graph visualization using LINE, extracted from survey[2].

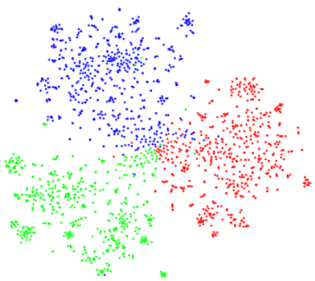


Figure 3: Graph visualization

2 Graph Embedding in Practice

Getting real network data sets in academic research is always far from trivial. In this task the objective is to use the well-known *DBLP* data set to test graph embedding. Alternative datasets, such as *Cora* (from McCallum et al. [6]), were considered but the DBLP dataset was selected for its heterogeneous graph structure, adding a layer of complexity to the task.

This section aims to apply state-of-the-art (SOTA) methodologies to the task of node classification on the heterogeneous graph learning context. This process involves first encoding entity properties and subsequently using Graph Neural Networks (GNNs) to derive graph embeddings. While various embedding techniques exist, in this case, Skip-gram [7] and sentence transformers [8] were utilized. Concerning the GNN, two distinct layers are tested for the heterogeneous GNN: GraphSAGE Convolution (SAGEConv) and Graph Attention Convolution (GATConv).

SAGEConv, rooted in GraphSAGE (Hamilton et al., 2017 [5]), which stands for Graph Sampled Aggregations, focuses on learning a function to generate embeddings by sampling and aggregating features from a node's local vicinity. The GraphSAGE convolution operation takes the features of neighboring nodes, aggregates them, and then combines them with the node's own features. This aggregation can be done in several ways, such as mean, LSTM, or pooling. The key advantage of SAGEConv is that it supports inductive learning, which means it can generate embeddings for new nodes or entire unseen graphs, assuming they share the same feature space with the training graphs.

The GATConv method is based on the Graph Attention Network (GAT) proposed in the paper "Graph Attention Networks" by Veličković et al. in 2018 [10]. The idea behind GAT is to incorporate the attention mechanism into the graph neural network. It is worth to mention this mechanism gained traction after "Attention is All You Need" by Vaswani et al. in 2017 [9]. In GATConv, every node aggregates the feature information from its neighboring nodes with the attention mechanism, which assigns different importance to different nodes. This attention mechanism allows the model to focus more on the important nodes and less on the less relevant ones, and this importance is learned through the training process. The GATConv layer's main advantage is that it captures the most relevant features from the neighbors by assigning different weights (attention coefficients), which are determined by the feature interactions between connected nodes.

In order to evaluate this approach, a subset of the DBLP dataset has been curated in two different ways. The former is provided in the `pytorch-geometric` (PyG) [3] library, which is the framework used to construct and train the above-mentioned methods. The DBLP subset available in PyG is curated as in Fu et al. [4]. That is, it only consists of nodes from authors, papers, terms, and conferences; and its relationships. Additionaly, each node contains already encoded features. For instance, authorsare described by a bag-of-words representation of their paper keywords. Finally, the authors are divided into four research areas (database, data mining, artificial intelligence, information retrieval), which are meant to be used for the node classification task. Detailed information on the dataset is depicted in Table 1.

The alternative dataset was preprocessed similarly, but the source is raw, enabling to work on a different property encoding. The only nodes curated are those from authors and articles, along with their relationships. Citations, which were synthetically generated, are ignored. To evaluate the node

Node/Edge Type	#nodes/#edges	#features	#classes
Author	4,057	334	4
Paper	14,328	4,231	
Term	7,723	50	
Conference	20	0	
Author-Paper	196,425		
Paper-Term	85,810		
Conference-Paper	14,328		

Table 1: Description of the PyG DBLP dataset.

Node/Edge Type	#nodes/#edges	#features	#classes
Author	3,736	0	
Paper	998	768	4
Author-Paper	4357		

Table 2: Description of the custom DBLP dataset.

classification task in this setting, the articles are clustered into four groups based on their titles. Each article is encoded using the **all-mpnet-base-v2** model from the `sentence-transformers` library in a natural language processing (NLP) context, resulting in a 768-dimensional continuous element vector. Once each article is encoded, **Spectral Clustering** from `scikit-learn` is applied to obtain four clusters, mirroring the previously curated subset. It is important to note that all the data is considered during the clustering process. Subsequently, the data is split into train, validation, and test sets. Detailed information on the dataset is depicted in Table 2.

Thus, two datasets are prepared for the task of node classification, where each node already possesses its encoded information. The selected model architecture is illustrated below:

```python
class HeteroGNN(torch.nn.Module):
    def __init__(self, hidden_channels, out_channels, num_layers):
        super().__init__()
        self.convs = torch.nn.ModuleList()
        for _ in range(num_layers):
            conv = HeteroConv({
                ('paper', 'to', 'term'): GATConv((-1, -1), hidden_channels,
    add_self_loops=False),
                ('author', 'to', 'paper'): SAGEConv((-1, -1), hidden_channels),
                ('paper', 'rev_to', 'author'): SAGEConv((-1, -1), hidden_channels),
            }, aggr='sum')
            self.convs.append(conv)
        self.lin = Linear(hidden_channels, out_channels)

    def forward(self, x_dict, edge_index_dict):
        for conv in self.convs:
            x_dict = conv(x_dict, edge_index_dict)
            x_dict = {key: F.relu(x) for key, x in x_dict.items()}
        return self.lin(x_dict['author'])
```

This architecture is designed specifically for the PyG DBLP dataset, as the convolutional layers require appropriately named edges. Similarly, the value to be returned during the forward pass should follow the same naming convention. It is important to highlight that the implemented block already handles classification by incorporating a final linear layer after the convolutions. Alternatively,

another approach could involve obtaining the embeddings and applying a different model on top of them, similar to the approach taken by `Neo4J` in its **HashGNN** implementation. Also, it is worth mentioning that if citations were to be considered, the GATCOnv layer with its `add_self_loops` set to `True` would be enough to handle it.

The training process involves iterating over batches of data within each epoch. In each iteration, the optimizer's gradients are reset to avoid gradient accumulation. The chosen optimizer is Adam. Cross-entropy loss is calculated between the model's predictions and the actual author labels, considering the batch size. Subsequently, the backward pass is performed to compute the gradients, which are then utilized by the optimizer to update the model's parameters. The training loss is recorded at the end of each epoch.

During evaluation, the model is switched to the evaluation mode, deactivating features like dropout and batch normalization. The validation data is fed through the model to obtain predictions. These predictions are transformed into class labels by selecting the class with the highest predicted probability. The predicted labels are compared against the ground truth for the corresponding nodes, enabling the calculation of the number of correct predictions. This number is then divided by the total number of validation nodes to compute the validation accuracy.

For the testing phase, a similar evaluation process is employed, but additional classification metrics such as precision and recall are computed. Once again, the model is set to evaluation mode, and predictions are generated for the test data. The final metrics are then calculated based on these predictions.

As indicated in Figure 4, the training loss and evaluation accuracy of the model on the DBLP PyG dataset show convergence after approximately 8 epochs. This indicates that the model has effectively learned to predict the data's structure and labels up to this point. Consequently, the model after this 8th epoch is used to test the data.

Table 3 presents the performance metrics for the test dataset. For each class, precision, recall, F1-score, and the number of samples (support) are shown. Precision is a measure of the model's accuracy in terms of the positive predictions it makes, while recall measures how many of the actual positives our model captures through labeling it as positive. The F1-score is a balanced harmonic mean of precision and recall.

Class 0 shows a balanced precision, recall, and F1-score of 0.86, indicating a high level of correct predictions with a good balance between precision and recall. Class 1 has slightly lower scores, with a precision of 0.70 and recall of 0.75, resulting in an F1-score of 0.72. Classes 2 and 3 present balanced scores of around 0.80.

The model's overall accuracy is reported as 0.81, which signifies that, on average, 81% of the model's predictions are correct across all classes. The macro-average (calculated as the mean of scores for each class) and the weighted average (which takes class imbalance into account) of precision, recall, and F1-score are around 0.80 and 0.81, respectively. These averages demonstrate that the model has a fairly consistent performance across all classes.

The results indicate that the model has learned to classify nodes effectively on the DBLP PyG dataset, providing valuable insights for node classification in heterogeneous graphs. However, further

	Precision	Recall	F1-score	Support
Class 0	0.86	0.86	0.86	977
Class 1	0.70	0.75	0.72	584
Class 2	0.83	0.78	0.81	892
Class 3	0.81	0.84	0.82	804
Accuracy	0.81	3257		
Macro avg	0.80	0.80	0.80	3,257
Weighted avg	0.81	0.81	0.81	3,257

Table 3: Final results for the DBLP PyG test subset.

optimizations may be explored to improve the model's performance on certain classes, such as Class 1.

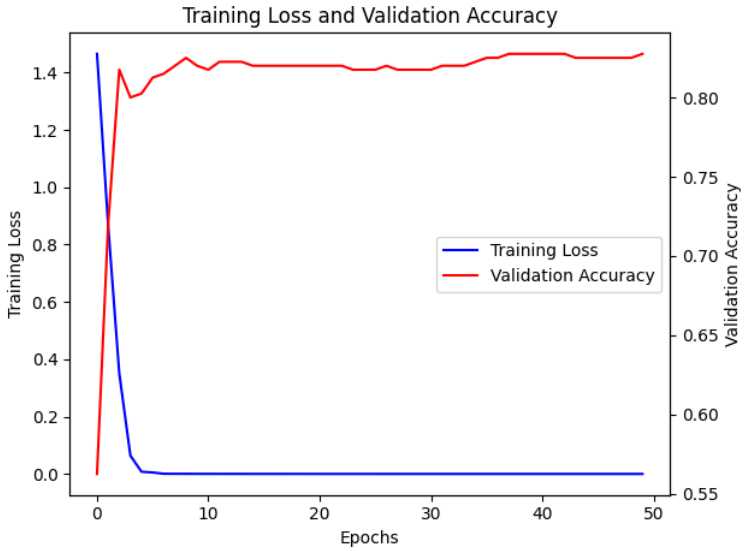


Figure 4: Training loss and validation accuracy through 50 epochs.

The results obtained from the other dataset demonstrate exceptionally high performance metrics. Notably, both the macro average and weighted F1 score achieved a remarkable value of 0.98. Such exceptional scores might be attributed to the model effectively capturing the underlying patterns and relationships within the dataset, which may have been facilitated by the clustering process utilized to learn from the synthetically generated classes. Hence, considering the meaningless nature of the results, no further exploration or analysis is pursued.

2.1 Conclusions and Limitations

The DBLP database was utilized in this project to implement a successful Heterogeneous Graph Neural Network (HeteroGNN) for node classification. By incorporating Graph Attention Convolution (GATConv) and GraphSAGE Convolution (SAGEConv) layers, the model demonstrated notable performance when learning from diverse graph data.

The initial phase of this research extensively examined graph embeddings, laying the groundwork for subsequent stages of model development and implementation.

For one of the DBLP datasets used in model training, a distinctive approach to encoding was em-

ployed, employing Natural Language Processing (NLP) sentence transformers. This approach effectively transformed unstructured textual data into meaningful numerical representations that could be processed by the model.

However, the model's performance fell short of state-of-the-art models due to limited hyperparameter tuning and the narrow scope of the data considered. Nevertheless, these results validate the viability of the applied approach and indicate a promising direction for a future extension.

Several limitations should be acknowledged when interpreting the results. Firstly, more systematic hyperparameter optimization could potentially enhance the model's performance. Secondly, the study focused on a limited subset of the DBLP database, and incorporating more diverse data, such as citation relationships, could provide a deeper understanding of the graph structure. Lastly, while the model performed well on synthetically clustered datasets, its performance on more complex or noisy data requires further exploration. Nonetheless, this project establishes a strong foundation for node classification understanding in heterogeneous graphs.

References

[1] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. *A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications*. 2018. arXiv: 1709.07604 [cs.AI].

[2] Peng Cui et al. *A Survey on Network Embedding*. 2017. arXiv: 1711.08752 [cs.SI].

[3] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. arXiv: 1903.02428 [cs.LG].

[4] Xinyu Fu et al. "MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding". In: *Proceedings of The Web Conference 2020*. ACM, Apr. 2020. DOI: 10.1145/3366423.3380297. URL: https://doi.org/10.1145%5C%2F3366423.3380297.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI].

[6] Andrew Kachites McCallum et al. *Automating the Construction of Internet Portals with Machine Learning*. July 2000. DOI: 10.1023/A:1009953814988. URL: https://doi.org/10.1023/A:1009953814988.

[7] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[8] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: http://arxiv.org/abs/1908.10084.

[9] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[10] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].