MULTI-LEVEL APPROACH TO ACCURATE AND SCALABLE HYPERGRAPH EMBEDDING

Anonymous authors
Paper under double-blind review

ABSTRACT

Many problems such as node classification and link prediction in network data can be solved using graph embeddings, and a number of algorithms are known for constructing such embeddings. However, it is difficult to use graphs to capture non-binary relations such as communities of nodes. These kinds of complex relations are expressed more naturally as hypergraphs. While hypergraphs are a generalization of graphs, state-of-the-art graph embedding techniques are not adequate for solving prediction and classification tasks on large hypergraphs accurately in reasonable time. In this paper, we introduce NetVec, a novel multi-level framework for scalable unsupervised hypergraph embedding, which outperforms state-of-the-art hypergraph embedding systems by up to 15% in accuracy. We also show that NetVec is capable of generating high quality embeddings for real-world hypergraphs with millions of nodes and hyperedges in only a couple of minutes while existing hypergraph systems either fail for such large hypergraphs or may take days to produce the embeddings.

1 INTRODUCTION

A *hypergraph* is a generalization of a graph in which an edge can connect any number of nodes. Formally, a hypergraph H is a tuple (V, E) where V is the set of *nodes* and E is a set of nonempty subsets of V called *hyperedges*. Nodes and hyperedges may have weights. Graphs are a special case of hypergraphs in which each hyperedge connects exactly two nodes.

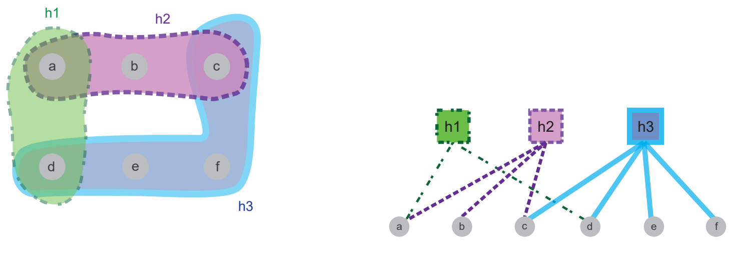
Figure 1a shows a hypergraph with 6 nodes and 3 hyperedges. The hyperedges are shown as colored shapes around nodes. The *degree* of a hyperedge is the number of nodes it connects. In the figure, hyperedge h2 connects nodes *a*, *b* and *c*, and it has a degree of three.

Hypergraphs arise in many application domains. For example, Giurgiu et al. (2019) model protein interaction networks as hypergraphs; nodes in the hypergraph represent the proteins and hyperedges represent *protein complexes* formed by interactions between multiple proteins. Piñero et al. (2019) represent a disease genomics dataset as a hypergraph in which nodes represents genes and hyperedges represent diseases associated with certain collections of genes. Algorithms for solving hypergraph problems are then used to predict new protein complexes, and to predict that a cluster of genes is associated with an as-yet undiscovered disease. Two hypergraph problems have been studied in the literature: *node classification* and *hyperedge prediction*.

In classification problems on nodes, labels are given for a subset of the nodes, and the task is to predict the most probable label for the unlabeled nodes. For example, in paper-authorship network, an author can write multiple papers. We can show this network by a hypergraph where nodes are papers and authors are hyperedges. Papers may be classified into different subjects. The task in this application is to predict the subject of the given paper. In hyperedge prediction, the task is to predict which set of nodes may form hyperedges. In many applications, hyperedges may have different numbers of nodes, so hyperedge prediction algorithms should be general enough to handle sets of nodes of different sizes.

1.1 HYPERGRAPH EMBEDDING

Bengio et al. (2013) show that one way to solve such prediction problems in graphs is to find an embedding of the graph using *representation learning*. There is a rich literature on graph embedding



(a) A hypergraph (b) Star expansion

Figure 1: Example hypergraph and the corresponding star expansion

methods that use a variety of techniques ranging from random walks [Perozzi et al. (2014); Tang et al. (2015); Grover & Leskovec (2016)] to matrix factorization [Qiu et al. (2018)] and graph neural networks [Hamilton et al. (2018); Xu et al. (2019)]. Graph embedding techniques can be extended to hypergraphs in two ways but neither of them is satisfactory.

One approach is to represent the hypergraph as a graph by replacing each hyperedge with a clique of edges connecting the vertices of that hyperedge, and then use graph embeddings to solve the prediction problem. This approach has been explored by Feng et al. (2019) and Yadati et al. (2019). However, the clique expansion is lossy because the hypergraph cannot be recovered from the clique expansion in general. Kirkland (2017) show that this information loss persists even if the dual of the hypergraph is considered.

Zien et al. (1999) show that another approach is to work with the *star expansion* of the hypergraph. Given a hypergraph $H{=}(V,E)$ where V is the set of nodes, E is the set of hyperedges, we create a bipartite graph $H^* = (V^*, E^*)$ by (i) introducing a node v_e for each hyperedge $e \in E$ so in final graph $V^* = V \cup E$, and (ii) introducing an edge between a node $u \in V$ and a hyperedge node $v_e \in E$ if $u \in e$ in the hypergraph. *i.e.,* $E^* = (u, v_e) : u \in e, e \in E$. Figure 1(b) shows an example. Unlike the clique expansion of a hypergraph, the star expansion is not lossy provided nodes representing hyperedges are distinguished from nodes representing hypergraph nodes. However, graph representation learning approaches do not distinguish between the two types of nodes in the bipartite graph, which lowers accuracy for prediction problems as we show in this paper.

These problems motivated us to develop *NetVec*, a *parallel multi-level framework for constructing hypergraph embeddings*, which allows us to perform hypergraph embedding in a much faster and more scalable manner than existing methods. This multi-level framework differentiates between nodes and hyperedges of a hypergraph, which helps to maintain the structure of the hypergraph through successive levels of coarsening. We introduce a novel coarsening strategy that not only reduces the size of a hypergraph but also utilizes the feature vectors of the nodes of a hypergraph. This coarsening strategy can be combined with an iterative refinement algorithm described in this paper, which can be used on its own to improve the quality of given hypergraph embedding.

We evaluate NetVec on a number of data sets for tasks such as node classification and hyperedge prediction. Our experiments show that our multi-level framework can compute the embedding of hypergraphs with millions of nodes and hyperedges in just a few minutes without loss of accuracy in downstream tasks, while all existing hypergraph embedding techniques either fail to run on such large inputs and or take days to complete. These results show that NetVec reduces the computation time (and therefore the energy) required to generate embeddings of hypergraphs and graphs by orders of magnitude without compromising on accuracy.

Our main technical contributions are summarized below.

- **Unsupervised hypergraph embedding sytem:** To the best of our knowledge, NetVec is the first unsupervised hypergraph embedding system. The embeddings obtained from this framework can be used in various downstream tasks such as hyperedge prediction and node classification.

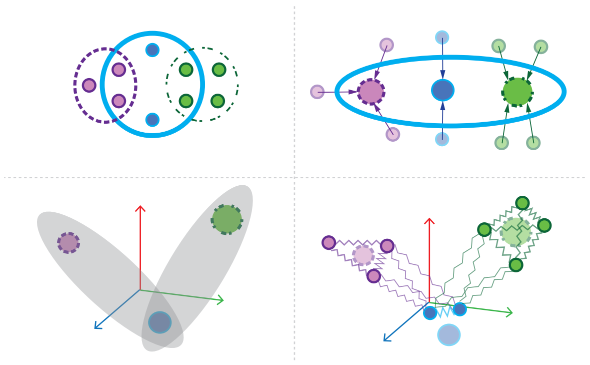


Figure 2: Multi-level embedding. **Top left**: original hypergraph. **Top right**: contracting nodes of a hypergraph to create a coarser hypergraph. **Bottom left**: initial coarse embedding. **Bottom right**: iterative refinement of node embeddings.

- **High-quality embeddings:** NetVec generates high-quality embeddings for hypergraphs. We propose a novel algorithm to approximate a hypergraph with a smaller hypergraph that utilizes the structure of the hypergraph as well as features of hypergraph nodes. We also propose a refinement algorithm that can further improve the quality of the embeddings. This results in a relative increase of the embedding accuracy over the prior works by up to 15%.
- **Scalability:** NetVec is the first hypergraph embedding approach that can generate embeddings of hypergraphs with millions of nodes and hyperedges. Our approach can significantly reduce run time while producing a better accuracy than state-of-the-art techniques.
- **Generality:** The techniques used in NetVec can be used to improve the performance of graph embedding systems as well.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 introduces the NetVec model in detail. Section 4 presents experimental results. We conclude in Section 5.

2 Related Work

There is a large body of work on graph and hypergraph embedding techniques so we discuss only the most closely related work.

2.1 Network Embedding

We divide these techniques into graph and hypergraph embedding techniques.

Graph embedding. There is a rich literature on graph embedding techniques. Approaches such as Grover & Leskovec (2016), and Perozzi et al. (2014) use random walks to generate a *corpus*, and apply a skip gram model on the corpus to generate the embedding. Edge reconstruction methods such as Tang et al. (2015) generate an embedding of a network by preserving first-order and second-order proximity. Graph neural network approaches such as Hamilton et al. (2018) collect feature information from a node's neighborhood, and the final embedding of a node is generated using intermediate embeddings from its neighbors. A more recent work is Veličković et al. (2018), which generates embeddings of a graph based on maximizing mutual information between local and global information of a graph. Scaling graph embedding to large graphs is still an open problem.

Hypergraph embedding. There are relatively few efforts on hypergraph embedding that treat hyperedges as first-class entities: in most existing techniques, the notion of a hyperedge is lost when the hypergraph is replaced by a graph.

As mentioned above, one popular approach to hypergraph embedding [Zhou et al. (2007); Tu et al. (2018b); Zhang et al. (2018); Feng et al. (2019)] is to convert the hypergraph to a graph and then use a graph embedding technique. For example, each hyperedge can be replaced with a clique connecting the nodes of that hyperedge to produce a graph representation. Other approaches such as Yadati et al. (2019) use a graph convolution on a modified clique expansion technique where they *choose* what edges to keep in the graph representation. While this method keeps more structure than methods based on the clique expansion of a hypergraph, existing methods fail to scale to large networks.

2.2 MULTI-LEVEL EMBEDDING

Multilevel approaches attempt to improve the runtime and quality of existing embedding techniques. This approach was first introduced for graphs in Bui & Jones (1993); Barnard & Simon (1993) and later extended to graph embedding in Chen et al. (2017); Liang et al. (2020); Deng et al. (2020). Multi-level graph embedding consists of three phases: coarsening, initial embedding, and refinement. *Coarsening*: A coarsened graph G' is created by merging pairs of nodes in input graph G. This process is applied recursively to the coarser graph, creating a sequence of graphs. in which the final graph is the coarsest graph that meets some termination criterion (*e.g.*, its size is below some threshold). *Initial embedding*: Any of the techniques discussed in Sections 2.1 can be used to generate an initial embedding. *Refinement*: For graphs G' and G, the embedding of G' is projected onto G and then refined, starting from the coarsest graph and finishing with the original graph. In principle, the same approach can be adopted for hypergraphs. However, multilevel approaches for hypergraphs have considered only hypergraph partitioning [Karypis et al. (1999); Devine et al. (2006); Maleki et al. (2021)].

These limitations led us to design NetVec, which preserves the notion of hyperedges in the representation of the data, and scales to hypergraphs with millions of nodes and hyperedges.

3 METHODOLOGY

3.1 MULTI-LEVEL FRAMEWORK

Algorithm 1: NetVec

```
Input: Hypergraph $H = (V, E)$; node feature matrix $X \in \mathcal{R}^{V \times k}$; coarsening depth $c$; refinement depth $r$; base embedding function $f$
Output: Vector representation $h_u, \forall u \in (V \cup E)$
$H_1^* = (V^*, E^*) \leftarrow \mathsf{StarExpansion}(H)$
for $i = 1$ to $c - 1$ do
    $H_{i+1}^* = \mathsf{Coarsening}(H_i^*)$
end for
$h_c \leftarrow f(H_c^*)$  $\{h$ is the embedding$\}$
for $i = c - 1$ to $1$ do
    $h_i(u) \leftarrow h_{i+1}(\tilde{u}), \forall u \in \tilde{u}$
    for $j = 1$ to $r$ do
        $h_i \leftarrow \mathsf{Refinement}(H_i^*, h_i)$
    end for
end for
```

Given a hypergraph H=(V,E), the algorithms described in this paper use the star expansion of the hypergraph and assign a vector representation h_u for each u∈$(V$∪$E)$. Intuitively, these embeddings attempt to preserve *structural similarity* in the hypergraph: if two hyperedges have many nodes in common or if two nodes are in many of the same hyperedges, the algorithm attempts to assign the two hyperedges/nodes to points that are close in the vector space. Embedding should also exploit the *transitivity* property of similarity: if a and b are similar, and b and c are similar, we want the embedding of a and c to be close to each other as well. Finally, if nodes have features, the embeddings should also exploit *functional similarity* between nodes.

Figure 2 illustrates the high-level idea of multi-level hypergraph embedding. Pseudocode for NetVec is given in Algorithm 1. This framework consists of three phases: (i) *Coarsening*, which iteratively merges nodes of the hypergraph to shrink the size of the hypergraph until the hypergraph is small enough that any network embedding algorithm can quickly obtain the embedding of the smallest hypergraph; (ii) *Initial embedding*, in which a network embedding algorithm is used on the coarsest hypergraph to generate the embedding, and (iii) *Refinement*, in which the embedding vectors of the coarser hypergraph are projected onto a finer hypergraph and a refinement algorithm is used to refine these embedding vectors. In the rest of this section, we describe these phases in more detail. NetVec is a parallel implementation of the multilevel approach. For lack of space, we do not discuss the issues that arise in implementing these phases in parallel.

3.1.1 COARSENING

Intuitively, coarsening finds nodes that are *similar* to each other and merges them to obtain a coarser hypergraph. To obtain a high quality embedding, we need to explore both structural similarity and functional similarity. The connectivity of nodes and hyperedges of a hypergraph determines structural similarity while node features determine functional similarity.

The first step in coarsening a hypergraph is to find nodes that are similar to each other and merge them. This is accomplished by "assigning" each node to one of its hyperedges, and then merging all nodes $\{n_1, n_2, ..., n_k\}$ assigned to a given hyperedge to produce a node m of the coarser hypergraph. We refer to m as the *representative* of node n_i in the coarse hypergraph, and denote it as rep(n_i). If all nodes of a hyperedge h_j are merged, we remove that hyperedge from the hypergraph. Otherwise, we add the hyperedge to the next level and refer to it as rep(h_j). If a node n_i is contained in hyperedge h_j in the finer hypergraph and h_j is present in the coarse hypergraph, then rep(n_i) is made a member of rep(h_j).

If nodes of a hypergraph have features, this information can be used to find similar nodes and merge them together. In a hypergraph with node features, the feature vector of a hyperedge is the mean aggregation of the features of its nodes. In this scenario, the cosine similarity between a feature vector of a hyperedge and a node is a good measure for assigning nodes to hyperedges. However, if the hypergraph has no features, NetVec uses other measures such as weights or degrees of a hyperedge to assign nodes to hyperedges. For example, if a node belongs to a hyperedge h_j with weight w_j and degree d_j, the "importance" of that node for that hyperedge can be estimated from the value of w_j/d_j, and the node can be assigned to the hyperedge with the largest importance score.

Algorithm 3 in the appendix lists the pseudo-code of coarsening. K is a hyperparameter that determines the number of levels of coarsening. At each level of coarsening, *ComputeEdgeFeature* computes a feature vector for a hyperedge by finding the mean aggregation of the feature vectors of its nodes. *AssignHyperedge* assigns each node v in the current hypergraph to a hyperedge $c(v)$ as defined as $c(v) = \text{argmax}_{e \in \mathcal{N}(v)} \dfrac{f(e) \cdot f(v)}{|f(e)| \cdot |f(v)|}$ where $\mathcal{N}(v)$ is the set of hyperedges that node v belongs to, $f(e)$ is the feature vector of hyperedge e and $f(v)$ is the feature vector of node v. Nodes that are assigned to the same hyperedge are merged together and the resulting node is added to the coarse hypergraph. In case of a tie, NetVec chooses a hyperedge randomly. Figure 2 top left shows the assignment of nodes to hyperedges. The figure in the top right shows the hypergraph after one level of coarsening. Nodes with the same color are merged together and form a single node in the coarser hypergraph. Hyperedges purple and green are removed from the hypergraph since all nodes in these hypergraphs are merged together. Hyperedge blue, however, is not removed since it has nodes that are merged in other hyperedges. A straightforward implementation has a time complexity of $O(|E^*| \cdot |f|)$ for assigning nodes to hyperedges and $O(|E^*|)$ for merging nodes and adding them to the coarser hypergraph where $|E^*|$ is the number of edges in the star expansion of the hypergraph, and $|f|$ is the size of the feature vector. So, the overall time complexity for each level of coarsening is linear in the size of the bipartite representation of the hypergraph. A detailed algorithm for coarsening is presented in the Appendix.

One important point about the coarsening algorithm is that it is cognizant of the fact that the bipartite graph is the representation of a hypergraph, and it merges only nodes of the hypergraph to create the coarser graph so that a coarser graph represents the original hypergraph. In particular, it does not merge nodes in the star expansion that represent nodes and hyperedges in the hypergraph, since this will destroy the structure of the hypergraph and the resulting structure will not be a good approximation of the original hypergraph. The experiments in Section 4 show the importance of this.

Algorithm 2: Refinement

```
Input: Bipartite graph representation $H^* = (V^*, E^*, W)$ of hypergraph $H = (V, E, W)$,
vector representation $z_u$ for all $u \in (V^*)$, neighborhood function $\mathcal{N}(u)$, parameter $\omega$, parameter
$k$ for max iteration
Output: Refined vector representation $h_u, \forall u \in (V^*)$
$z_u^0 \leftarrow z_u, \forall u \in (V^*)$
$iter = 0$
while $iter < k$ do
    for $u \in V^*$ do
        $\tilde{z}_u^i \leftarrow \sum_{v \in \mathcal{N}(u)} w_{uv} z_v^{i-1} / \sum_{v \in \mathcal{N}(u)} w_{uv}$
        $z_u^i \leftarrow (1-\omega) z_u^{i-1} + \omega \tilde{z}_u^i$
    end for
    $iter$ += 1
end while
$h_u \leftarrow z_u^k, \forall u \in (V^*)$
```

3.1.2 Initial Embedding

We coarsen the hypergraph until it is small enough that *any* unsupervised embedding method can generate the embedding of the coarsest hypergraph in just a few seconds. We use the edgelist of the coarsest bipartite graph as the input to this embedding method. The running time for computing this initial embedding is reduced if the size of the coarsest graph is reduced.

3.1.3 Refinement

The goal of this phase is to improve embeddings by performing a variation of Laplacian smoothing [Taubin (1995)] that we call the *refinement* algorithm. The basic idea is to update the embedding of each node u using a weighted average of its own embedding and the embeddings of its immediate neighbors $\mathcal{N}(u)$. Intuitively, smoothing eliminates high-frequency noise in the embeddings and improves the accuracy of downstream inference tasks. A simple iterative scheme for smoothing is shown below:

$$\tilde{z}_u^i = \sum_{v \in \mathcal{N}(u)} \left(\frac{w_{uv}}{\sum_{v \in \mathcal{N}(u)} w_{uv}}\right) z_v^{i-1} \tag{1}$$

In this formula, z_u^i is the embedding of node u in iteration i, and w_{uv} is the weight on the outgoing edge from u to v; if there no weights in the input hypergraph, a value of 1 is used and the denominator is the degree of node u. This iterative scheme can be improved by introducing a hyper-parameter ω that determines the relative importance of the embeddings of the neighboring nodes versus the embedding of the node itself, to obtain the following iterative scheme:

$$z_u^i = (1-\omega) z_u^{i-1} + \omega \tilde{z}_u^i \tag{2}$$

The initial embeddings for the iterative scheme are generated as follows. For the coarsest graph, they are generated as described in Section 3.1.2. For the other hypergraphs, if a set of nodes S in hypergraph H_{i-1} was merged to form a node n in the coarser hypergraph H_i, the embedding of n in H_i is assigned to all the nodes of S in H_{i-1}.

Abstractly, this iterative scheme uses *successive over-relaxation* (SOR) with a parameter ω to solve the linear system $Lz = 0$ where L is the Laplacian matrix of H^*, defined as $(D - A)$ where D is the diagonal matrix with diagonal elements d_{uu} equal to the degree of node u for unweighted graphs (for weighted graphs, the sum of weights of outgoing edges), and A is the adjacency matrix of H^*. To avoid oversmoothing, we do not compute the exact solution of this linear system but if we start with a good initial embedding z^0, a few iterations of the iterative scheme lead to significant gains in the quality of the embedding, as we show experimentally in Section 4.

Algorithm 2 shows the psuedocode for refinement. The input to this algorithm is H^*, the bipartite (star) representation of the hypergraph, z_u, the initial embedding for each node and hyperedge, and a relaxation parameter ω between 0 and 1. Embeddings of the hyperedges are updated using the embeddings of the nodes, and the embeddings of nodes are updated using the embeddings of hyperedges. Note that if u represents a hyperedge, $\mathcal{N}(u)$ is the set of nodes in that hyperedge, and if u represents a node in the hypergraph, $\mathcal{N}(u)$ represents the set of hyperedges that u is contained in. Each iteration of the refinement algorithm has a linear time complexity in the size of the bipartite representation of the hypergraph.

Intuitively, the updates to the embeddings of nodes and hyperedges made by the relaxation algorithm in each iteration exploit *structural* similarity. For example, if two hyperedges e_1 and e_2 have many nodes in common, these nodes will bring the embeddings of e_1 and e_2 closer. Performing these updates iteratively exploits *transitivity* of similarity. For instance, hyperedges a and c may not have nodes in common but if each of them has many nodes in common with hyperedge b, the embeddings of a and c will become closer after a few iterations.

4 EXPERIMENTS

NetVec provides an unsupervised method for representation learning for hypergraph. We show these representations perform well for both node classification and hyperedge prediction. In contrast, prior works such as HyperGCN and Hyper-SAGNN have been evaluated for one or the other of these tasks but not both.

All experiments are done on a machine running CentOS 7 with 4 sockets of 14-core Intel Xeon Gold 5120 CPUs at 2.2 GHz, and 187 GB of RAM. All the methods used in this study are parallel implementations and we use the maximum number of cores available on the machine to run the experiments. The embedding dimension is 128. For the multi-level approaches (MILE, GraphZoom, and NetVec), the execution time is the sum of the CPU time for coarsening, initial embedding, and refinement. For the rest of the baselines, we use the CPU time for hypergraph embedding.

Table 1: Datasets used for node classification

DATA SET	NODES	HYPEREDGES	EDGES	CLASSES	FEATURES
CORA	1,434	2,708	9,572	7	1,433
CITESEER	1458	1,079	6,906	6	3703
PUBMED	3,840	7,963	69,258	3	500
DBLP	41,302	22,363	19,122	6	1425

4.1 NODE CLASSIFICATION

Given a hypergraph and node labels on a small subset of nodes, this task is to predict labels on the remaining nodes. We used the standard hypergraph datasets from prior works, and these are listed in Table 1. We observe 4% of node labels and predict the remaining 96%. To demonstrate that NetVec produces high-quality results, we compare NetVec with state-of-the-art hypergraph embedding approaches as well as a number of popular methods for multi-level graph embedding approaches. In particular, we compare our results with HyperGCN[Yadati et al. (2019)], GraphZoom[Deng et al. (2020)], MILE[Liang et al. (2020)], and node2vec[Grover & Leskovec (2016)]. For the multi-level approaches, we use node2vec as the initial embedding method. We report the mean test accuracy and standard deviation over 100 different train-test splits. More details of datasets and baselines are available in Appendix. We optimize hyperparameters of all the baselines to achieve a better accuracy.

These are the main takeaways from Table 2:

- NetVec with two levels of coarsening generates the highest quality embeddings for the node classification task.
- NetVec outperforms HyperGCN in terms of quality for all datasets by up to 15%.
- The refinement algorithm improves the quality of embeddings for all the datasets by up to 23%. This can be seen by comparing the statistics for NetVec without coarsening ($l = 0$) with those for node2vec. The initial embedding for NetVec is obtained from node2vec, so differences in the statistics arise entirely from the fact that NetVec performs refinement.

Table 2: Node classification. Accuracy in % and time in seconds. l is the number of coarsening levels. 0 means without coarsening.

	CORA		**CITESEER**		**PUBMED**		**DBLP**	
	ACCURACY	TIME	ACCURACY	TIME	ACCURACY	TIME	ACCURACY	TIME
NETVEC(l=0)	67.5 ±3.	5.2	59.1 ±1.	6.5	72.2 ±1.	15.1	72.4 ± .4	126.1
NETVEC(l=1)	67.0 ±3.	6.1	60.5 ± 1.	4.1	79.8±1.	15.9	77.7 ±.4	66.1
NETVEC(l=2)	**67.7** ±3.	8.1	**60.6** ± 1.	4.2	**80.3**±1.	17.1	**78.5** ±.4	56.1
NODE2VEC	44.5 ±3.	3.1	51.1 ±1.	3.9	64.5 ± .2	13.1	67.1 ± .4	60.5
HYPERGCN	57.6 ±6.	15.1	54.1 ±10	12.7	64.3 ± 10	60.0	63.3 ± 10	480.7
MILE(l=2)	48.1 ±3.	10.2	21.2 ±1.	14.1	68.7 ± .2	30.0	-	-
GRAPHZOOM(l=2)	60.1 ±3.	3.3	46.8 ±3.	3.2	74.9 ± .1	13.3	71.6 ± .5	60.0

- NetVec outperforms prior multi-level graph embedding approaches (MILE and GraphZoom) for all the datasets by up to 39% for MILE and up to 14% for GraphZoom.

The coarsening algorithm in NetVec plays an important role in producing high-quality embeddings. First, NetVec utilizes node features during coarsening. Second, by merging nodes that have at least one hyperedge in common, NetVec ensures that nodes that are structurally similar end up having similar embeddings. Finally, it allows the refinement algorithm to improve the quality of embeddings successively in multiple coarse graphs. MILE was unable to generate embeddings for DBLP. For the other hypergraphs, we believe that one reason that MILE does not perform as well is that it does not utilize node features.

4.2 HYPEREDGE PREDICTION

In hyperedge prediction, we are given a hypergraph with a certain fraction of hyperedges removed, and given a proposed hyperedge (i.e. a set of nodes) our goal is to predict if this is likely to be a hyperedge or not. Formally, given a k-tuple of nodes $(v_1, v_2, ..., v_k)$, our goal is to predict if this tuple is likely to be a hyperedge or not. We compare our method with the state-of-the-art hypergraph method Hyper-SAGNN [Zhang et al. (2020)] on four datasets listed in Table 3, and with the graph methods node2vec. We did not compare our method with other hypergraph methods such as DHNE [Tu et al. (2018a)] since Hyper-SAGNN has shown its superior performance over these methods. Finally, we study the scalability of NetVec on a large hypergraph (Friendster) and compare NetVec's accuracy and running time with that of MILE (Hyper-SAGNN, GraphZoom and node2vec failed to generate results for Friendster).

To make a fair comparison, we used the same training and test data setups as Hyper-SAGNN (except of course for Friendster, which Hyper-SAGNN could not run). For NetVec, we use two levels of coarsening and two levels of refinement, and use node2vec for the initial embedding. We then use the vector of the variances of each dimension of the embedding for hyperedge prediction. The intuition is that if nodes are spread out (high variance in the embedding), then they probably do not form a hyperedge whereas nodes that are close to each other are likely to constitute a hyperedge. We used the same setting for node2vec. Hyper-SAGNN is based on a supervised method in which it learns a function to map from embeddings of nodes to hyperedges for the hyperedge prediction task. We ran it for 300 epochs and report the AUC.

Experimental results. Table 4 summarizes the hyperedge prediction results for NetVec, node2vec, MILE, and Hyper-SAGNN. NetVec achieves the best AUC and running time compared to Hyper-SAGNN. Hyper-SAGNN took almost a day for Wordnet whereas NetVec completed the task in less than a minute. NetVec achieves better AUC compared to node2vec on all datasets except Drug while it is always the fastest. A more detailed discussion of hyperedge prediction is in Appendix.

We also applied NetVec on the large hypergraph Friendster which has millions of nodes and hyperedge. Since the hypergraph is large, we used five levels of coarsening and ten levels of refinement. The only other baseline that was able to run Friendster was MILE with 15 levels of coarsening, and it failed for smaller numbers of coarsening levels. It took MILE 8 hours to generate embeddings for Friendster with the accuracy of 90.4 while it took NetVec only fifteen minutes to do the same task with better accuracy (92.3%). Figure 3 compares MILE and NetVec in terms of accuracy for different levels of coarsening for NetVec (for MILE, we used fifteen levels of coarsening). One reason that MILE is slower than NetVec is that it uses GCN as refinement method. However, this requires training a GCN model, which is very time consuming for large graphs or hypergraphs. The main takeaway from Figure 3 is that, for a faster hypergraph embedding we have to use more levels of coarsening. However, a large number of coarsening may reduce the accuracy. While this is a fact in most multi-level approaches, Figure 3 shows that the loss of accuracy for NetVec is negligible and we are able to get more than 13x speed up by using 5 levels of coarsening instead of 3 while loosing less than 3% in accuracy. We perform a detailed ablation study on Friendster in the Appendix.

Table 3: Datasets used for hyperedge prediction.

	NODES	HYPEREDGES	EDGES
GPS	221	437	1,436
MOVIELENS	17,100	46,413	47,957
DRUG	7,486	171,757	171,756
WORDNET	81,073	146,433	145,966
FRIENDSTER	7,458,099	1,616,918	37,783,346

Table 4: Area Under Curve (AUC) scores for hyperedge prediction. Time in seconds.

	GPS		**MOVIELENS**		**DRUG**		**WORDNET**		**FRIENDSTER**	
	AUC	TIME	AUC	TIME	AUC	TIME	AUC	TIME	AUC	TIME
NETVEC	**94.5**	1	**94.8**	6.4	96.5	295	**93.0**	43.4	**93.2**	897
HYPER-SAGNN	90.6	1800	90.8	11,160	95.9	39,540	87.7	82,800	-	-
NODE2VEC	94.0	10	79.8	19	**97.4**	895	89.0	940	-	-

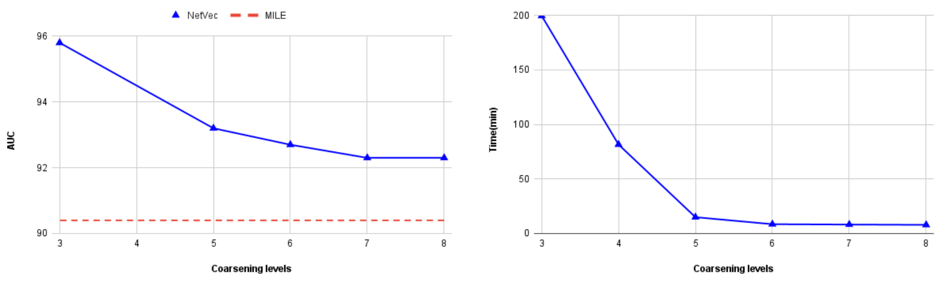


Figure 3: Performance on Friendster

5 CONCLUSION

We describe NetVec, a multi-level hypergraph embedding framework, which can process hypergraphs with millions of nodes and hyperedges in just a few minutes, producing high-quality embeddings for node classification and hyperedge prediction. Our experimental results show that it significantly outperforms multi-level graph embedding approaches such as MILE and GraphZoom in both accuracy and running time (some of these approaches fail for the large datasets in our studies). We also showed that our refinement algorithm can be used on its own to improve the quality of embeddings for hypergraphs by up to 23%. Even though NetVec is an unsupervised method, it outperforms supervised hyperedge prediction methods such as Hyper-SAGNN. In future work, we want to extend NetVec to multi-relation hypergraphs.

REFERENCES

Stephen Barnard and Horst Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. pp. 711–718, 01 1993.

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013. doi: 10.1109/TPAMI.2013.50.

T N Bui and C Jones. A heuristic for reducing fill-in in sparse matrix factorization. 12 1993.

Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. Harp: Hierarchical representation learning for networks, 2017.

Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. Graphzoom: A multi-level spectral approach for accurate and scalable graph embedding, 2020.

Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pp. 124–124, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL `http://dl.acm.org/citation.cfm?id=1898953.1899056`.

Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks, 2019.

M Giurgiu, J Reinhard, B Brauner, I Dunger-Kaltenbach, G Fobo, G Frishman, C Montrone, and A. Ruepp. CORUM: the comprehensive resource of mammalian protein complexes-2019. *Nucleic Acids Research*, 2019.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):69–79, March 1999. ISSN 1063-8210.

Steve Kirkland. Two-mode networks exhibiting data loss. *Journal of Complex Networks*, 6(2): 297–316, 08 2017. ISSN 2051-1329. doi: 10.1093/comnet/cnx039. URL `https://doi.org/10.1093/comnet/cnx039`.

Jiongqian Liang, Saket Gurukar, and Srinivasan Parthasarathy. Mile: A multi-level framework for scalable graph embedding, 2020.

Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. Bipart: A parallel and deterministic hypergraph partitioner. *SIGPLAN Not.*, 2021. doi: 10.1145/3437801.3441611.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 701–710, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623732. URL `http://doi.acm.org/10.1145/2623330.2623732`.

Janet Piñero, Juan Manuel Ramírez-Anguita, Josep Saüch-Pitarch, Francesco Ronzano, Emilio Centeno, Ferran Sanz, and Laura I Furlong. The DisGeNET knowledge platform for disease genomics: 2019 update. *Nucleic Acids Research*, 48(D1):D845–D855, 11 2019. ISSN 0305-1048. doi: 10.1093/nar/gkz1021. URL `https://doi.org/10.1093/nar/gkz1021`.

Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, Feb 2018. doi: 10.1145/3159652.3159706. URL `http://dx.doi.org/10.1145/3159652.3159706`.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line. *Proceedings of the 24th International Conference on World Wide Web*, May 2015. doi: 10.1145/2736277.2741093. URL `http://dx.doi.org/10.1145/2736277.2741093`.

Gabriel Taubin. A signal processing approach to fair surface design. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pp. 351–358, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917014. doi: 10.1145/218380.218473. URL `https://doi.org/10.1145/218380.218473`.

Ke Tu, Peng Cui, Xiao Wang, Fei Wang, and Wenwu Zhu. Structural deep embedding for hyper-networks, 2018a.

Ke Tu, Peng Cui, Xiao Wang, Fei Wang, and Wenwu Zhu. Structural deep embedding for hyper-networks, 2018b.

Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax, 2018.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. Hypergcn: A new method of training graph convolutional networks on hypergraphs, 2019.

Muhan Zhang, Zhicheng Cui, Shali Jiang, and Yixin Chen. Beyond link prediction: Predicting hyperlinks in adjacency space. In *AAAI*, pp. 4430–4437, 2018.

Ruochi Zhang, Yuesong Zou, and Jian Ma. Hyper-SAGNN: a self-attention based graph neural network for hypergraphs. In *International Conference on Learning Representations (ICLR)*, 2020.

Denny Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, pp. 1601–1608. MIT Press, September 2007.

J. Y. Zien, M. D. F. Schlag, and P. K. Chan. Multilevel spectral hypergraph partitioning with arbitrary vertex sizes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18 (9):1389–1399, 1999. doi: 10.1109/43.784130.

A Appendix

A.1 Details of Datasets

- **Cora**: a computer science publication citation network dataset. All documents cited by a document are connected by a hyperedge. Each document is classified into one of seven classes based on topic. Nodes not connected to any hyperedge, as well as hyperedges containing only one node, were removed.
- **Citeseer**: scientific publications classified into six classes. All documents cited by a document are connected by a hyperedge. Nodes not connected to any hyperedge, as well as hyperedges containing only one node, were removed.
- **Pubmed**: scientific publications classified into three classes. All documents cited by a document are connected by a hyperedge. Nodes not connected to any hyperedge, as well as hyperedges containing only one node, were removed.
- **DBLP**: scientific publications classified into six classes. All documents co-authored by an author are in one hyperedge.
- **Friendster**: a community network obtained from https://snap.stanford.edu/data/. It is an on-line gaming network. Users can form a group on Friendster social network which other members can then join. These user-defined groups are considered as communities. For the social network, the induced subgraph of the nodes that either belong to at least one community or are connected to other nodes that belong to at least one community are considered too. Communities larger than 500 were removed.

A.2 Details of Baselines

- **Node2Vec**: random-walk based method. It generates random walks by using a return parameter p and an in-out parameter q. We set these parameters 1.0 and 1.0 respectively. The length of random walk in this paper is 20, number of walks per node is 10, and the context window size is 10.
- **MILE**: multi-level graph embedding framework. We use node2vec as its initial embedding method. We used the default refinement technique, MD-gcn.
- **GraphZoom**: multi-level graph embedding framework. For the coarsening, we used *simple*. We used node2vec for the initial embedding.
- **HyperGCN**: Given a hypergraph, HyperGCN approximates the hypergraph by a graph where each hyperedge is approximated by a subgraph. A graph convolutional network (GCN) is then run on the resulting graph. Since this method is not proposed for hyperedge prediction, we do not consider them for this task. We used 200 epochs and learning rate of 0.01.
- **Hyper-SAGNN**: A self-attentionbased approach for hyperedge prediction. Since this method is not proposed for node classification, we do not consider them for this task. We used learning rate of 0.001 and 300 epochs.

A.3 Coarsening Algorithm

Algorithm 3 shows the detailed algorithm for coarsening.

A.4 Ablation Study

In this section we perform an ablation study to understand the effectiveness of the major NetVec kernels.

Table 5 shows the effect of different levels of refinement algorithm on node classification datasets. Performing just 5 levels of refinement improves the quality of the initial embedding by up to 12%. This table also shows that after about 100 iterations of refinement, the improvement in the quality of the embedding stops.

Algorithm 3: Coarsening

```
Input: fineGraph $G = (V, E)$, node feature matrix $X \in \mathcal{R}^{V \times k}$,neighborhood function $\mathcal{N}(u)$,
depth $K$
Output: coarseGraph $G' = (V', E')$, node feature matrix $X' \in \mathcal{R}^{V' \times k}$
for $i = 1$ to $K$ do
  for $e \in E$ do
    ComputeEdgeFeature($e$)
  end for
  for $v \in V$ do
    AssignHyperedge($v$)
  end for
  for $e \in E$ do
    $M \leftarrow$ FindAssignedNodes($e$)
    $m \leftarrow$ Merge($M$)
    coarseGraph.addNode($m$) # m is representative of each node in M
    if $|M| \neq |e|$ then
      coarseGraph.addHedge($e$) # representative of e
    end if
  end for
  for $v \in V$ do
    for $e \in \mathcal{N}(v)$ do
      if rep($e$) exists then
        coarseGraph.addEdge(rep($e$),rep($v$))
      end if
    end for
  end for
end for
```

Table 5: Accuracy of NetVec at different numbers of refinement iterations for node classification (REF-N is the number of iterations of refinement)

REF-N	0	5	30	40	50	60	70	80	90	100
CORA	44.5	58.9	65.6	66.5	66.9	67.2	67.4	67.6	67.7	67.7
CITESEER	51.1	56.1	59.4	59.9	60.1	60.2	60.4	60.5	60.6	60.6
PUBMED	64.5	76.6	79.5	79.9	80.1	80.2	80.4	80.5	80.5	80.5
DBLP	67.1	74.5	77.3	77.7	78.0	78.1	78.3	78.3	78.4	78.5

We study the behaviour of NetVec for the largest hypergraph Friendster. In Table 8 we can see the effect of the coarsening on the size of the hypergraph as well as accuracy and running time. Coarsening improves the running time by up to 10x while the accuracy roughly stays the same.

Table 6: Behavior of NetVec at different levels of coarsening on Friendster. (COARSE-N is the number of levels of coarsening)

COARSE-N	3	5	6	7	8
HYPEREDGES	564,262	419,588	404,857	399,194	396,333
NODES	436,099	85,371	67,682	61,669	59,359
ACCURACY	95.8	93.2	92.7	92.3	92.3
TIME (SEC.)	11,972	897	512	492	475

Table7 shows the breakdown of time in different NetVec's kernel. If the coarsest hypergraph is small, most of the time spends in refinement while for large coarsest hypergraphs, the time is mostly spend in initial embedding.

Table 7: Time break down for Friendster dataset for different levels of coarsening. Time in seconds.

COARSE-N	3	5	6	7	8
COARSENING	31	36	43	41	39
INITIAL EMBEDDING	11,760	600	120	51	29
REFINEMENT	181	261	261	349	406

Table 8: Hypergraph size at different coarsening level for dataset Friendster. (COARSE-N is the number of levels of coarsening)

COARSE-N	1	2	3	4	5	6	7	8
HYPEREDGES	1,602,472	817,450	564,262	460,830	419,588	404,857	399,194	396,333
NODES	7,458,099	1,214,343	436,099	154,418	85,371	67,682	61,669	59,359

A.5 HYPEREDGE PREDICTION

For hyperedge prediction, we compare our result with the state-of-the-art hyperedge prediction Hyper-SAGNN. We used the same datasets used in their paper. For this task, they randomly hide 20 percentage of existing hyperedges and use the rest of the hypergraph for training. The negative samples are 5 times the amount of positive samples. We downloaded their code and datasets from their GitHub repository. We used the encoder-based approach to generate the features. We ran Hyper-SAGNN for 300 epochs. One noteworthy aspect of Hyper-SAGNN is that they use a different set of negative samples for each epoch.

For NetVec, we first obtain the embedding of the hypergraphs with node2vec as the initial embedding technique. We find the embedding of the hypergraph without seeing the hidden hyperedges. To train our classifier, we used the same positive samples as Hyper-SAGNN. For negative samples however, we used only the negative samples of a *single* epoch of Hyper-SAGNN. *We note that NetVec is able to obtain a high-quality embedding of the hypergraph with a smaller pool of negative samples and still obtain good accuracy compared to Hyper-SAGNN.* This reduces the time of training from days to minutes as we showed in the main paper.

The results for node2vec in the main paper are also better than what was reported in the Hyper-SAGNN paper. The reason is that in their paper, they use node2vec for predicting pairwise edges. Then for each triplet, they have three probability scores. They used *average* operator to aggregate the scores into one and calculate AUC based on this one score. We realized that if we find the *variance* of each dimension and use that for training a classifier, the AUC would be much higher for node2vec and that is what we have reported in the main paper.

For the dataset Friendster, we randomly hide 20% of existing hyperedges and use the rest of the hypergraph to generate the embeddings for the nodes of the hypergraph and finally, use the variance operator to report the AUC.