HYPEREDGE2VEC: DISTRIBUTED REPRESENTATIONS FOR HYPEREDGES

Anonymous authors
Paper under double-blind review

ABSTRACT

Data structured in form of overlapping or non-overlapping sets is found in a variety of domains, sometimes explicitly but often subtly. For example, teams, which are of prime importance in social science studies are “sets of individuals”; “item sets” in pattern mining are sets; and for various types of analysis in language studies a sentence can be considered as a “set or bag of words”. Although building models and inference algorithms for structured data has been an important task in the fields of machine learning and statistics, research on “set-like” data still remains less explored. Relationships between pairs of elements can be modeled as edges in a graph. However, modeling relationships that involve all members of a set, a hyperedge is a more natural representation for the set. In this work, we focus on the problem of embedding hyperedges in a hypergraph (a network of overlapping sets) to a low dimensional vector space. We propose a probabilistic deep-learning based method as well as a tensor-based algebraic model, both of which capture the hypergraph structure in a principled manner without losing set-level information. Our central focus is to highlight the connection between hypergraphs (topology), tensors (algebra) and probabilistic models. We present a number of interesting baselines, some of which adapt existing node-level embedding models to the hyperedge-level, as well as sequence based language techniques which are adapted for set structured hypergraph topology. The performance is evaluated with a network of social groups and a network of word phrases. Our experiments show that accuracy wise our methods perform similar to those of baselines which are not designed for hypergraphs. Moreover, our tensor based method is quiet efficient as compared to deep-learning based auto-encoder method. We therefore, argue that we have proposed more general methods which are suited for hypergraphs (and therefore also for graphs) while maintaining accuracy and efficiency.

1 INTRODUCTION

In group structured data we have multiple entities related by some form of group relationships. In fact, such data is more abundantly found in the real world than has been usually studied (Estrada & Rodriguez-Velazquez, 2005). There are increasing number of fields where such data is being found more than ever before. In social networks domain: team data from massive online multiplayer games (Ahmed et al., 2011) such as World of Warcraft, group communication tools such as Skype and Google Docs and research collaborations (Subbian et al., 2013) [pub]. There are other fields in which structural relationships between entities is important as well, and large datasets capturing them exist. Examples include Natural Language Processing (Bengio & Bengio, 2000), Biology (Hwang et al., 2008; Klamt et al., 2009), e-commerce (Deshpande & Karypis, 2004; Chris- takopoulou & Karypis, 2014) and Chemistry (Bartholomay, 1960). Figure 1 shows such examples for networks of groups, sentences, and item sets.

Hypergraph (Berge, 1984), which is a generalization of graphs, is a popular model to naturally capture higher-order relationships between sets of objects (Figure 2) (Estrada & Rodriguez-Velazquez, 2005). Within machine learning, algorithms guided by the structure of such higher order networks (Zhou et al., 2006) have found applications in a variety of domains (Tian et al., 2009; Gao et al., 2013; Li & Li, 2013; Sharma et al., 2015; 2017).
Figure 1: Example illustrating “set-like” structures from various domains.

More recently, the interest of representation learning in NLP (Mikolov et al., 2013a) has stimulated its application in graph embedding (learning low dimensional representation for graph nodes) (Perozzi et al., 2014; Tang et al., 2015b; Grover & Leskovec, 2016; Cai et al., 2017). However, this new line of research is limited to simple graphs and we are unaware of any work that considers hyperedge embeddings (learning representations for hyperedges). In this paper we aim to develop methods which (1) learn hypergraph embeddings directly, (2) leverage the hypergraph topology and (3) not loose the hyperedge-level joint information. These learned embeddings can then be employed by a supervised or semi-supervised algorithm to perform various predictive tasks pertaining to hyperedges. For example performance prediction of a team (set of individuals) engaged in a collaborative task or sentiment analysis of a sentence (set of words).

Although, there have been a variety of attempts to learn node embeddings for hypergraphs by extending traditional graph embedding methods for hypergraph setting. These approaches differ in the extent they address the aims (2) and (3) as pointed previously. In first category, there are a number of methods that incorporate the hypergraph topology using proxy graphs (Zhou et al., 2006; Hwang et al., 2008), therefore, incur loss of information. Also, Agarwal et al. (2006) squarely criticize that such representations can be learned by constructing graphs, which are proxies for the hypergraph structure. Second, are methods which actually, take into account the hyperedge-level information, like the classical Neural Language Model which maintains the higher order sequence information (Bengio & Bengio, 2000). But their model is designed for sequences and also simply ignores the topology (sequences are linked by common words forming a network of sequences). However recently, there has been interest in modeling “set-like” structures within deep-learning community (Vinyals et al., 2016; Rezatofighi et al., 2016). But they do not consider the hypergraph structure between the sets and therefore, not model hypergraphs in a principled manner. Third category are those methods that do not ignore the topology completely and also retain the hyperedge-level information. Among this are k-way tensor based methods that explicitly do not work on hypergraph (Shashua et al., 2006; Bulò & Pelillo, 2009), but the connection exists in the form that k-way tensor represents a k-uniform hypergraph (Qi, 2005). But they are restricted to uniform hypergraphs.

We therefore, propose two methods, both of which directly learn hyperedge embeddings for general hypergraphs and capture the hypergraph structure in a principled manner. First, is an auto-encoder based deep-learning method and second, is a tensor-based algebraic method. We also highlight the connection between hypergraphs (topology), tensors (algebra) and probabilistic models. A number of interesting baselines are also developed, some of which adapt existing node-level embedding models to hypergraph setting and also sequence based NLP techniques are adapted for set structured hypergraphs. We have experimented with social group dataset (network of teams as a hypergraph) from an online game for team (hyperedge) performance prediction as well as language networks (sentence/phrase hypergraphs) for phrase-level sentiment analysis. Our experiments show that accuracy wise our methods perform similar to those of baselines which are not designed for hypergraphs. Moreover, our tensor based method is quiet efficient as compared to deep-learning based auto-encoder method. We therefore, argue that we have proposed more general methods which are suited for hypergraphs (and therefore also for graphs) while maintaining accuracy and efficiency. In summary the main contributions of our paper are:
We propose the novel concept of a dual tensor corresponding to the hypergraph dual and allows us to get hyperedge embedding directly.

We propose a general hypergraph tensor decomposition method designed for general hypergraphs (containing different cardinality hyperedges) unlike simple uniform hypergraph tensor decomposition which is restricted to fixed cardinality hyperedges (i.e. uniform hypergraph). We are unaware of any such works or applications employing this approach.

We propose the novel idea to use de-noising auto-encoder in a hypergraph setting. Moreover, we also develop techniques of creating noise using random-walks over hasse diagram topology, which is original and unique.

We highlight and argue that embeddings from tensor based methods have a natural hypergraph theoretic interpretation unlike the deep learning based black-box approaches.

Our proposed methods learn hypergraph embeddings based on the unique approach of leveraging the existing structure present in network data as context, in contrast to context generation based existing graph embedding techniques.

Following is the outline for the rest of the paper. In section 2 we describe the problem definition and statement followed by Section 3 where we describe in detail the various methods proposed in this paper. Section 4 describes the datasets, experimental tasks & settings for the models. Section 5 provides an overview of the related literature followed by conclusion and appendix.

2 Preliminaries

In this paper we consider the scenario where we have a collection of elements. These elements can represent individual actors in case of social groups or words in sentences or items in item-sets within a transaction database. In other words a social group or a sentence or an item-set are sets which contain these elements. Let $V = \{v_1, v_2, \ldots, v_n\}$ represents $n$ elements and we have $m$ sets defined over these elements, denoted by $G = \{g_1, g_2, \ldots, g_m\}$, where $g_i \subseteq V$ represents the $i^{th}$ set. The cardinality $|g_i|$ represents the number of elements in the set. Also each set $g_i \in G$ has an occurrence number $R(g_i)$, which denotes the number of times it has occurred. Such overlapping or non-overlapping sets can be modeled as a hypergraph [Berge 1984], where the nodes and hyperedges, represent the elements and sets, respectively. This hypergraph is represented as $N_g = (V, G)$ with $G$ as the collection of hyperedges over the nodes $V$. The incidence matrix $H \in \{0, 1\}^{|G| \times |V|}$ for $N_g$ represents the presence of nodes in different hyperedges with $H(g_i, v) = 1$ if $v \in g_i$ else 0. A specialization of hypergraph model is the Simplicial complex [Munkres 1984] (Figure 2), in which additionally each hyperedge has the subset closure property, i.e., each subset of hyperedge is also a valid hyperedge.

**Problem Statement:** Given this setting, our goal is to learn the mapping $\phi : G \rightarrow \mathbb{R}^d$ from hyperedges to feature representations (i.e., embeddings) that can be used to build predictive models involving sets. Here $d$ is a parameter specifying the number of dimensions of the embedding vector.
Equivalently, \( \phi \) can be thought of as a look-up matrix of size \( |G| \times d \), where \( |G| \) is the total number of sets or hyperedges.

3 Methodology

As we had mentioned before there are several methods to learn representation of nodes in a graph. Given that a hyperedge is a set of nodes, a natural question arises is that, can we combine the node level embeddings (learned using existing methods) within a given hyperedge to find a suitable representation of that hyperedge? However, there are large number of possible ways one can combine the node embeddings. Therefore, we design an algorithm which is principally suited for hypergraph structured experiments, which approximate hyperedge or set-level information with dyadic edge-level information. Unlike the proxy graphs (corresponding to hypergraphs) based techniques (used as baselines in our experiments), where the auto-encoder based method harnesses the non-linearity of representation offered by deep learning to generate crisp and powerful embeddings. Tensor based method has a natural Hasse diagram based topological interpretation and can generate both node and hyperedge embeddings. Whereas deep learning model is more black-box based approach and can only generate hyperedge embeddings.

3.1 Hyperedge2vec using Hypergraph Tensor Decomposition

In this section, we develop tensor (higher-order matrices) based linear algebraic methods that learn node as well as hyperedge embedding by taking into account the joint probability over a hyperedge. The idea behind using tensors is that they retain the set-level information contained in a hypergraph, unlike the proxy graphs (corresponding to hypergraphs) based techniques (used as baselines in our experiments), which approximate hyperedge or set-level information with dyadic edge-level information. Therefore, we design an algorithm which is principally suited for hypergraph structured data. For a given hypergraph we can extract a sub-hypergraph that only consists of the hyperedges with cardinality \( k \). This sub-hypergraph is a \( k \)-uniform hypergraph or \( k \)-graph [Cooper & Dutle (2012)]. Corresponding to this \( k \)-uniform hypergraph, we can define a \( k \)-th order \( n \)-dimensional symmetric tensor \( [Q] \) (2005), \( \hat{A}_{\text{hyp}}^k = (a_{p_1, p_2, ..., p_k}) \in \mathbb{R}^{[k,n]} \) whose elements are initialized as follows:

\[
a_{p_1, p_2, ..., p_k} = R(g_i)
\]

where \( \{v_{p_1}, v_{p_2}, ..., v_{p_k}\} \in g_i \) and \( |g_i| = k, \forall i \in \{1, ..., m\} \). Note that symmetry here implies that value of element \( a_{p_1, p_2, ..., p_k} \) is invariable under any permutation of its indices \( (p_1, p_2, ..., p_k) \). Rest all the elements in the tensor are zeros.

In a similar manner we can also define a dual tensor, corresponding to hypergraph dual where the roles of nodes and hyperedges are interchanged. We consider all the hyperedges in the original hypergraph which have a degree of \( k \), i.e., they are part of exactly \( k \) hyperedges in the original hypergraph. Corresponding to this \( k \)-uniform hypergraph dual, we can define a \( k \)-th order \( m \)-dimensional symmetric dual tensor \( \hat{A}_{\text{dual}}^k = (a_{q_1, q_2, ..., q_k}) \in \mathbb{R}^{[k,m]} \) whose elements are initialized as follows:

\[
a_{q_1, q_2, ..., q_k} = 1
\]

where \( \{g_{q_1}, g_{q_2}, ..., g_{q_k}\} \ni v_j \) and \( |\{r|H(r,j) = 1\}| = k, \forall j \in \{1, ..., n\} \). Note that this tensor is also symmetric and rest all the elements in the tensor are zeros.

To realize our aim of learning node and hyperedge embeddings we perform joint CP Tensor Decomposition [Kolda & Bader (2009)] (of the tensors we just described) across different cardinality hyperedges simultaneously. Specifically, for the node embeddings we solve the following optimization problem:

\[
\min_{Z} \sum_{k=c_{\text{min}}}^{c_{\text{max}}} \| \hat{A}_{\text{hyp}}^k - \hat{A}_{\text{hyp}}^k \| (3)
\]

where,

\[
\hat{A}_{\text{hyp}}^k = \sum_{r=1}^{d} \lambda_r z_{r} \otimes z_{r} \otimes \cdots \otimes z_{r} = (\lambda)^{k \text{ times}} \langle Z, Z, ..., Z \rangle
\]

\[
\hat{A}_{\text{dual}}^k = \sum_{r=1}^{d} \lambda_r z_{r} \otimes z_{r} \otimes \cdots \otimes z_{r} = (\lambda)^{k \text{ times}} \langle Z, Z, ..., Z \rangle
\]
with \( z_r \in \mathbb{R}^n \), \( \lambda_r \in \mathbb{R}_+ \), \( Z \in \mathbb{R}^{n \times d} \) and \( Z(:, r) = z_r \). Notice, that equation \( \ref{eq:cp-decomposition} \) is the standard symmetric CP decomposition but equation \( \ref{eq:non-symmetric-cp-decomposition} \) is summation of reconstruction errors in different tensor decompositions for different cardinality hyperedges. Each error term learns a common latent factor matrix \( Z \) using the empirical observed \( k \)-uniform sub-hypergraph stored in \( A_{\text{hyp}}^k \) tensor. We perform this joint decomposition by augmenting the standard CP Decomposition into the Hypergraph-CP-ALS algorithm \( \ref{alg:hypergraph-als} \). Lines (4-8) is the standard (non-symmetric) CP Decomposition (see reference \cite{kolda2009tensor} for details), which solves the following optimization:

\[
A_{\text{hyp}}^k = \sum_{r=1}^{d} \lambda_r k \cdot \mathbf{u}_r^{(1)} \odot k \cdot \mathbf{u}_r^{(2)} \ldots \odot k \cdot \mathbf{u}_r^{(k)} \equiv \langle \lambda |_{k} U^{(1)}, k U^{(2)}, \ldots, k U^{(k)} \rangle
\]  

with \( k \cdot \mathbf{u}_r^{(j)} \in \mathbb{R}^n \), \( \lambda_r \in \mathbb{R}_+ \), \( k \cdot U^{(j)} \in \mathbb{R}^{n \times d} \), \( k \cdot U^{(j)}(:, r) = k \cdot \mathbf{u}_r^{(j)} \) and \( j \in \{1, \ldots, k\} \). As we want to learn common representations \( (Z) \) for all the nodes, we add additional constraint that enforce that \( k \cdot U^{(j)} \) are same for all \( j \in \{1, \ldots, k\} \) and for all \( k \in \{c_{\min}, c_{\max}\} \). Lines (9-15) is where we force all representations for all cardinalities to be average of the \( k \cdot U^{(j)} \) achieved by the last decomposition iteration that occurred in lines (4-8). Averaging can be interpreted as equal contribution from the latent factors learned from different cardinality (uniform) sub-hypergraphs. We make repeated pass through the entire hypergraph (by learning via different \( k \)-uniform sub-hypergraph (line 3)) until the objective (equation \( \ref{eq:lp-decomposition} \)) converges. In our implementation we empirically observe that using the above mentioned unbiased averaging heuristic, our algorithm converges successfully. The same algorithm \( \ref{alg:hypergraph-als} \) is used to get the hyperedge embeddings, by just passing \( A_{\text{dual}} \) instead of \( A_{\text{hyp}} \). We shall jointly refer to the embeddings achieved for nodes and hyperedges via the above tensor decomposition techniques as \( \mathbf{t}_2 \mathbf{v} \).

**Algorithm 1 Hypergraph-CP-ALS \( (A_{\text{hyp}}, c_{\min}, c_{\max}) \)**

1. randomly initialize \( k \cdot U^{(j)}, \forall k \in [c_{\min}, c_{\max}], \forall j \in \{1, \ldots, k\} \)
2. repeat
3. for \( k = c_{\min} \) to \( c_{\max} \) do
4. for \( j = 1 \) to \( k \) do
5. \( V \leftarrow k \cdot U^{(1)} \odot \ldots \odot k \cdot U^{(j-1)} \odot k \cdot U^{(j+1)} \ldots \odot k \cdot U^{(k)} \)
6. \( k \cdot U^{(j)} \leftarrow A_{\text{hyp}}^k (\langle k \cdot U^{(1)} \odot \ldots \odot k \cdot U^{(j-1)} \odot k \cdot U^{(j+1)} \ldots \odot k \cdot U^{(k)} \rangle)^\top \)
7. normalize columns of \( k \cdot U^{(j)} \) (and store norms as \( \lambda \))
8. end for
9. \( Z \leftarrow \frac{1}{k} \sum_{j=1}^{k} k \cdot U^{(j)} \)
10. for \( p = c_{\min} \) to \( c_{\max} \) do
11. for \( j = 1 \) to \( p \) do
12. \( \hat{y}^{(j)} \leftarrow Z \)
13. end for
14. end for
15. end for
16. until fit criteria achieved or maximum number of iterations exceeded
17. return \( Z \)

We would like to highlight a few points regarding the tensor methods. The tensors that we have employed are super-symmetric and hence able to capture distribution over sets rather than sequence. But in general we can employ a \( k \)-way tensor which is not symmetric to even capture sequence. In this sense tensors are more general purpose. Another point one can observe that when we initialize the hypergraph tensor \( A_{\text{hyp}}^k \), we have initialized all the permutations of vertices corresponding to a given hyperedge (Eq. \( \ref{eq:hypergraph-als} \)). Moreover, we initialize it by the repetition counts.

### 3.2 Hyperedge2Vec Using Hasse De-noising Autoencoder

An autoencoder \cite{bengio2009learning}, takes an input vector \( x \in [0, 1]^n \) and maps it to a latent representation \( y \in [0, 1]^d \). This is typically done using an affine mapping followed by a non-linearity (more so when the input, like in our case, is binary \cite{vincent2010stacked})

\[
\hat{y}(x) = s(Wx + b),
\]

where \( W \) is a \( n \times d \) weight matrix and \( b \) is the offset.
Figure 3: For the given hypergraph between four nodes \((A,B,C,D)\) we consider the complete Hasse lattice. For a given hyperedge \(\{B,C\}\) (square box) we then construct the sub-lattice made of hyperedges with distance \(h = 2\) from \(\{B,C\}\). We perform random walk (with \(\tau = 0.2\)) starting from the node corresponding to hyperedge \(\{B,C\}\) and sample \(p = 3\) hyperedges (nodes visited by the random walk; shown with a check-mark \(\checkmark\)). Finally, we train the autoencoder to reconstruct the original hyperedge from these \(p\) noisy hyperedges.

This latent representation is reconstructed into a vector \(z = f_{de}(y) = s(W'y + b')\), in the input space, \(z \in [0,1]^n\) and parameters \(\theta' = \{W', b'\}\). The mappings \(f_{en}\) and \(f_{de}\) are referred as the encoder and decoder, respectively. The representation \(y\) is learned by minimizing the following reconstruction error:

\[
\theta^*, \theta'^* = \arg\min_{\theta^*, \theta'^*} \frac{1}{m} \sum_{i=1}^{m} L(x_i, z_i) = \arg\min_{\theta^*, \theta'^*} \frac{1}{m} \sum_{i=1}^{m} L(x_i, g_{\theta'}(f_{\theta}(x_i))) \tag{6}
\]

where \(L\) is a loss function, which in case of binary or bit probabilities is often chosen as the cross-entropy loss:

\[
L(x, z) = \sum_{j=1}^{n} [x(j) \log z(j) + (1 - x(j)) \log(1 - z(j))] \tag{7}
\]

In their paper, Vincent et al. (2010) have shown that minimizing reconstruction amounts to maximizing the lower bound on the mutual information between input \((x)\) and the representation \(y\). However, they have further argued (Vincent et al., 2008) that \(y\) retaining information about input \(x\) is insufficient. They further, propose the idea that the learned representation should be able to recover (denoising) the original input even after being trained with corrupted input (adding noise). They generate the corrupted input \((\tilde{x})\), using a stochastic mapping \(q(\tilde{x}|x)\). Choice of noise is usually either Gaussian for real inputs and Salt-and-pepper noise for discrete inputs. The denoising autoencoder then learns the representation for each input, \(x\), same as Equation 5 but with the following modified loss function: \(L(x(i), g_{\theta'}(f_{\theta}(\tilde{x}(i))))\).

We leverage the denoising autoencoder for learning representation for \(j^{th}\) hyperedge, by treating each hyperedge as an input, \(x = H(j,:)\). The size of this input vector for each hyperedge is \(n\), which is the number of vertices in the hypergraph. In most natural hypergraphs, specially social networks, \(n\) can be quiet high ranging from thousands to millions or even billions (like Facebook for example). Therefore, randomly using a discrete noise like salt-and-pepper, might not be reasonable, as there are large number of possible permutations (as size \(n\) is large) and not all of them are related. Random addition of 1s or deletion of existing 1s from \(x\), amounts to randomly adding or deleting vertices to the hyperedge corresponding to \(x\). This might end up in new hyperedges that are completely unrelated to the given hyperedge \((x)\). For example, users (nodes) in a social network...
from completely different regions of the network suddenly form a group (hyperedge). Such anomalous scenarios rarely happen in practice and social groups evolve in a gradual fashion via simple processes (Sharma et al., 2017, 2015).

Rather, we take advantage of the hypergraph structure to systematically guide us in generating this noise. A hypergraph can be defined by its corresponding hasse lattice (Sharma et al., 2017). For a given hyperedge \( x \), we consider the sub-lattice consisting of only those hyperedges that are a distance \( h \) from it in the complete lattice. On this sub-lattice we sample \( p \) hyperedges (nodes in sub-lattice) by performing random walk starting at the given hyperedge’s node (see Figure 3). We assume that all the nodes in the sub-lattice which correspond to previously occurred hyperedge \( g_i \) have weight \( R(g_i) \) and rest all nodes have a constant weight \( \tau \). During random walk from a given node we choose a neighboring node (as the next node) in proportion to this neighboring node’s weight as compared to the other neighboring nodes. Our stochastic mapping \( q(\tilde{x}|x) \) is therefore, a random walk on the sub-lattice of hyperedge \( x \) containing hyperedges at distance \( h \) from it. Intuitively, the hyperedges coming within a reasonable distance will affect each others representations and will have more similar representations. We will refer to the hyperedge representations learned by the above autoencoder technique, as \textit{h2v-auto}.

4 Experiments

4.1 Dataset Description

As the first dataset, we use group interaction log-data of the Sony’s Online multi-player game EverQuest II (EQ II) (www.everquest2.com) for a time period of nine months. In this game, several players work together as a team to perform various tasks. Each team earn points after completion of each task, and as the teams progress by earning points, they are escalated to different levels of game play. The interestingness of the game increases with each level. The points earned by the teams are treated as a measure of group performance. Each set of players who played together is treated as a hyperedge. We treat the number of times same set of players play together again as hyperedge occurrence number \( R(g_i) \). Players can participate in several teams over time, therefore, resulting in a hypergraph with overlapping hyperedges. We consider hyperedges of cardinality \( \in [2,6] \) as almost 90\% of our hyperedges lie within this range. The resulting dataset contains a total of 5964 hyperedges (teams) among 6536 nodes (players).

Second dataset, is the fully labeled and publicly available sentiment analysis corpus of Stanford Sentiment Treebank (LangNet) (Socher et al., 2013). This dataset is based on the reviews from a movie review website (www.rottentomatoes.com) and contains 215,154 unique phrases. Each of the phrases are labeled with a sentiment score (a real number \( \in [0,1] \), larger value indicates positive sentiment) by human annotators. Each phrase is a set or hyperedge of words. As there is no occurrence information for a phrase hyperedge we consider \( R(g_i) = 1, \forall i \in \{1,...,m\} \). Given that words are shared across various phrases, these common words connect the phrase hyperedges, resulting in a phrase hypergraph with overlapping phrase hyperedges. Again, we only consider hyperedges of cardinality \( \in [2,6] \). After applying this cardinality filter we are left with 141,410 hyperedges (phrases) and 21,122 nodes (words).

4.2 Evaluation Methodology and Experimental Setup

4.2.1 Methods Compared

As mentioned before we refer to our proposed methods: tensor based hypergraph tensor decomposition and deep auto-encoder based hypergraph auto-encoder, as \textit{t2v} and \textit{h2v-auto}, respectively. We compare our proposed methods against six baselines which are listed below. Methods (1-2) are adapted from sequence based language models for set structured hypergraph data. Methods (3-6) are various kind of dyadic graph based embedding methods adapted for hypergraph setting.

1. \textbf{h2v-DM:} Hyperedge and node embeddings obtained using sentence embedding using distributed memory (DM) model (refer section A.1)

2. \textbf{h2v-DBOW:} Hyperedge and node embeddings obtained using sentence embedding using distributed memory (DBOW) model (refer section A.1)
3. **h2v-inv**: Hyperedge and node embeddings obtained using node2vec on inverted hypergraph (refer section A.2)
4. **h2v-dual**: Hyperedge and node embeddings obtained using node2vec on hypergraph dual (refer section A.2)
5. **e2v**: Hyperedge and node spectral embeddings obtained using eigen decomposition of inverted hypergraph laplacian and graph laplacian, respectively (refer section A.3)
6. **e2v-hyp**: Hyperedge and node spectral embeddings obtained using eigen decomposition of dual laplacian and hypergraph laplacian, respectively (refer section A.3)

The details of these baseline can be found in Appendix A. Except for **h2v-auto**, each of these methods result in node as well as hyperedge embeddings of dimension $d = 128$. We further combine the node and hyperedge embedding using five different strategies: i) node embedding summation (dimension $d = 128$), ii) node embedding summation and concatenation with hyperedge embedding (dimension $2 \times d = 256$), iii) node embedding averaging (dimension $d = 128$), iv) node embedding averaging and concatenation with hyperedge embedding (dimension $2 \times d = 256$), and v) only hyperedge embedding (dimension $d = 128$). **h2v-auto** only produces hyperedge embeddings of dimension $d = 128$. But it builds the embeddings using three different scenarios as mentioned in next section. Therefore, in total we have $38 (= 35 + 3)$ different scenarios each resulting in a different hyperedge embedding.

### 4.2.2 Evaluation Tasks and Setup

We perform two regression based tasks for the two datasets. In **EQII** dataset each team (hyperedge) has a team performance score associated with it. This team performance score is a real number, equal to the number of points earned by the team while performing one or more tasks within a gaming session. We treat the embedding learned for a given team (hyperedge) as its feature vector which is associated with a real number (team performance). We therefore, perform on regression over all the hyperedges (teams) with team performance as the dependent variable.

Similarly, in **LangNet** dataset each phrase (hyperedge) has a sentiment score associated with it, which again is a real number. Similar to the team dataset above, we treat the embedding learned for a given phrase (hyperedge) as its feature vector which is associated with a real number (sentiment score). We therefore, treat this as a regression task with sentiment score as the dependent variable and perform regression using the feature matrix containing embeddings of all the phrases.

For both the tasks we just described, we perform several evaluation runs. In each run we randomly choose 30% of hyperedges (teams or phrases) as the test set and learn ridge regression parameters using the remaining 70% training hyperedges for each of the 38 different embedding scenarios. Root mean squared error (RMSE) was chosen as the evaluation metric (the lower, the better). RMSE was calculated for each of the 38 scenarios and for each run. Final RMSE score was taken as the average RMSE score across five runs. Ridge regression’s hyper-parameter was chosen by 5-fold cross-validation.

For the autoencoder method (**h2v-auto**) we consider three scenarios: (1) single hidden layer ($L_1$) of $d = 128$; (2) two hidden layers ($L_1 \& L_2$) with size of $L_1 : d = 96$ and of $L_2 : d = 32$. We concatenate these embedding to get a single $d = 128$ size embedding; and (3) two hidden layers ($L_1 \& L_2$) with size of $L_1 : d = 512$ and of $L_2 : d = 128$. We use the output of $L_2$, which is of dimension $d = 128$, as the embedding. For sampling, we use the distance parameter $h = 2$ for generating the sub-lattice and $\tau = 0.2$ for both datasets. Also, $p = 10$ & $p = 5$ number of hyperedges are sampled (corresponding to each hyperedge) from **EQII** and **LangNet**, respectively.

### 4.3 Results and Discussion

Tables 1 & 2 show the RMSE scores for the tasks of team performance prediction and sentiment score prediction, respectively. These tables contain scores for all the 35 different scenarios: columns represent 7 (6 baselines and the proposed t2v) different models while rows represent combination strategies. The scores for **Hypergraph Auto-encoder** are shown in the separate Table 3 as autoencoder only generates hyperedge embeddings.
Table 1: RMSE Scores for **EQ II** team performance

<table>
<thead>
<tr>
<th>Embed Combination</th>
<th>Baselines</th>
<th>Hypergraph</th>
<th>Tensor Decomp.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sentence Embed based</td>
<td>Node2Vec based</td>
<td>Spectral methods</td>
</tr>
<tr>
<td></td>
<td>h2v-DM</td>
<td>h2v-DBOW</td>
<td>h2v-inv</td>
</tr>
<tr>
<td>Node Embed Sum</td>
<td>0.79308</td>
<td>0.79567</td>
<td>0.80418</td>
</tr>
<tr>
<td>Node Embed Sum + Hyperedge Embed</td>
<td>0.79651</td>
<td>0.80241</td>
<td>0.81362</td>
</tr>
<tr>
<td>Node Embed Average</td>
<td>0.81584</td>
<td>0.81733</td>
<td>0.82407</td>
</tr>
<tr>
<td>Node Embed Avg + Hyperedge Embed</td>
<td>0.8182</td>
<td>0.82077</td>
<td>0.83378</td>
</tr>
<tr>
<td>Only Hyperedge Embed</td>
<td>0.81203</td>
<td>0.81522</td>
<td>0.82189</td>
</tr>
</tbody>
</table>

Table 2: RMSE Scores for **LangNet** Sentiment Analysis

<table>
<thead>
<tr>
<th>Embed Combination</th>
<th>Baselines</th>
<th>Hypergraph</th>
<th>Tensor Decomp.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sentence Embed based</td>
<td>Node2Vec based</td>
<td>Spectral methods</td>
</tr>
<tr>
<td></td>
<td>h2v-DM</td>
<td>h2v-DBOW</td>
<td>h2v-inv</td>
</tr>
<tr>
<td>Node Embed Sum</td>
<td>0.14081</td>
<td>0.14029</td>
<td>N/A</td>
</tr>
<tr>
<td>Node Embed Sum + Hyperedge Embed</td>
<td>0.14028</td>
<td>0.13883</td>
<td>N/A</td>
</tr>
<tr>
<td>Node Embed Average</td>
<td>0.14245</td>
<td>0.14115</td>
<td>N/A</td>
</tr>
<tr>
<td>Node Embed Avg + Hyperedge Embed</td>
<td>0.14178</td>
<td>0.14007</td>
<td>N/A</td>
</tr>
<tr>
<td>Only Hyperedge Embed</td>
<td>0.14194</td>
<td>0.14147</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3: Average Runtime (seconds) of various methods across datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Baselines</th>
<th>Hypergraph</th>
<th>Tensor Decomp.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sentence Embed based</td>
<td>Node2Vec based</td>
<td>Spectral methods</td>
</tr>
<tr>
<td></td>
<td>h2v-DM</td>
<td>h2v-DBOW</td>
<td>h2v-inv</td>
</tr>
<tr>
<td>EQ2</td>
<td>455.84</td>
<td>103.47</td>
<td>90.05</td>
</tr>
<tr>
<td>LangNet</td>
<td>80.61</td>
<td>62.31</td>
<td>211.97</td>
</tr>
</tbody>
</table>

* these are average time taken for learning vertex embeddings only

Table 4: RMSE Scores & Run-times for **LangNet** (Sentiment Analysis) & **EQ II** (Team Performance) using Hypergraph Auto-encoder (**h2v-auto**)

<table>
<thead>
<tr>
<th>Layer Sizes</th>
<th>EQ II</th>
<th>LangNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:128</td>
<td>L1:128</td>
<td>L1:512/L2:128</td>
</tr>
<tr>
<td>L1:96/L2:32</td>
<td>L1:96/L2:32</td>
<td>L1:512/L2:128</td>
</tr>
<tr>
<td>L1:512/L2:128</td>
<td>L1:512/L2:128</td>
<td>L1:512/L2:128</td>
</tr>
</tbody>
</table>

| RMSE         | 0.81104 | 0.14568 | 0.14479 |
| Run Time     | 52 min | 40 min | 1 hr 20 min | 2 hr 10 min | 3 hr 20 min | 6 hr |

As we can observe that for our datasets and for both the regression tasks, almost all the embeddings are performing very similarly in terms of accuracy. However, we can observe in Table 3 that tensor based method (**t2v**) have significant less time than the expensive auto-encoder technique (**h2v-auto**) (Table 4). This makes tensor based well understood and interpretable decomposition techniques more attractive than the black-box type deep learning based auto-encoder method.

Over all, we observe that **e2v**, **e2v-hgraph** and **t2v**, all of which are matrix or tensor based algebraic techniques, perform comparable to state of the art embedding techniques from NLP. Therefore, it provides motivation to further explore algebraic techniques, which are easily interpretable in graph theoretic terms, as a reasonable alternative for learning representations.

Another thing we notice, is that in case of **LangNet** dataset, that Node2Vec based **h2v-inv** and **h2v-dual** methods are simply unable to run, as the **A_inv** and **A_dual** are quiet large (see N/A in Table 2). It seems that in **LangNet** and possibly in text data in general the hypergraph dual, which contain
phrase to phrase edges turns out be containing significantly more edges than the number of node to node edges in $A_{hyp}$. Performing context generation over $A_{inv}$ and $A_{dual}$ graphs turns out be very costly and we were unable to get hyperedge embeddings for $h2v-inv$ and $h2v-dual$ methods. Note, however we do get vertex embedding (as mentioned with a * mark below Table 3). But as you can see that even the time time taken for vertex embedding is similar or more than the total time taken by spectral methods for learning both hyperedge as well as vertex embedding, together. This indicates the robustness of the Spectral methods, specially the $e2v-hyp$ for different kinds of hypergraph data and edge densities (sparsity).

**Scalability** Main scalability in the tensor approach arises with the increase in range of cardinality $k$. For a hyperedge of cardinality $k$ there are $k!$ entries to be filled in the tensor $A_{hyp}^k$. Per iteration complexity of the algorithm is $O(\sum_{k=\text{min}}^{\text{max}} \text{nnz}(A_{hyp}^k))$. Similarly, for a vertex of degree (i.e. number of hyperedges it is part of) there are $\delta!$ entries to be filled in the tensor $A_{dual}^\delta$. Per iteration complexity of the algorithm is $O(\sum_{\delta=\text{min}}^{\text{max}} \text{nnz}(A_{dual}^\delta))$. Now, as the $k$ increases, the number of $k$ cardinality hyperedge decrease, however the $A_{hyp}^k$ tensor grows exponentially making it increasingly sparse and less informative (in information theoretic sense). Therefore, depending upon the cardinality distribution of a given dataset, we can choose a threshold value ($\alpha$) for $k$ beyond which we don’t enumerate the permutations. Moreover, for enumeration till $k \leq \alpha$ in a scalable manner we propose to use hyperedge-centric distributed computation libraries like MESH [Heintz & Chanstra, 2015], HyperX [Heintz et al., 2017] and HyperX [Huang et al., 2015], for distributed enumeration per hyperedge. Similar argument goes for $A_{dual}^\delta$ where we ignore vertices with degree beyond a certain cut-off ($\beta$). These high degree vertices can be interpreted as highly popular person participating in a huge number of teams, but should the embeddings of these teams be related to each other is difficult to infer. In text data these high degree nodes correspond to words occurring in a large number of phrases, but this doesn’t mean the sentences are related, rather its just that this word is quiet common. We can again employ MESH or HyperX for scalability till $\delta \leq \beta$.

In case of auto-encoder method the main scalability challenge lies in generating the noisy hyperedges. For a given hyperedge the intermediate sub-lattice from which $p$ samples are drawn is of size $O(n^h)$, where $h$ is max distance from the hyperedge considered. Once this sampling is performed we have total $mp$ hyperedges as input for the auto-encoder, which is linear in the number of hyperedge as $p$ will be a constant. The main challenge therefore, is to generate the noisy samples where we have to tackle exponential size sub-lattice per hyperedge. Again, as this sampling is done for each hyperedge separately (hyperedge-centric), we can therefore, use distributed computing for hypergraphs using MESH and HyperX, as mentioned before.

Furthermore, we also observe that even simple element-wise summation or averaging of node embeddings for the nodes (in a given hyperedge) also perform comparably when compared to hyperedge embedding alone. From this we can infer that depending upon the dataset, if we have less hyperedges and more nodes, than we would rather prefer to simply learn the hyperedge embedding directly rather than learning node embeddings and then performing aggregating operation over them. Aggregation might turn out be costly specially if average hyperedge size is large and the choice of aggregation function is an issue. Therefore, learning hyperedge embeddings directly seems to be escape the problem of choosing the aggregation function all together.

**Choice of Method** Both tensor based ($t2v$ using $A_{dual}^k$) as well as auto-encoder ($h2v-auto$) methods proposed in this paper are able to generate embeddings for all hyperedges of various cardinalities present in data. However, the auto-encoder method cannot generate node embeddings unlike tensor method (using $A_{hyp}^k$). On contrary, in auto-encoder technique we don’t need to figure out $\alpha$ and $\beta$ parameters, and can work for all cardinality hyperedges. Another, thing to note is that tensors are inherently capable of retaining joint information because of their higher-order structure. Auto-encoder method implicitly (possibly lossy manner) leverage the higher order information embedded in network in form of node weights ($R(g_i)$) while performing the random walk sampling. Main advantage of auto-encoders is the crisp embedding achieved via multiple levels of non-linearity offered by deep neural networks. Although this non-linearity improves the accuracy of auto-encoder methods very slightly (Table 4) as compared to tensor method (bottom right corner of (Table 1 & 2), the computation cost of tensor is far less than the auto-encoder method (Table 3 & 4). Intuitively, it seems that depending upon the task at hand, in some task the retention of joint hyperedge-level information is more important for modeling and lesser presence of non-linear structure in the data; or vice versa.
But the low computational cost (as reflected in our tasks) makes tensor method definitely more lucrative. Together with the natural graph theoretic interpretation, tensor methods in comparison to black-box approach of deep learning based auto-encoder, makes them even more viable.

**Information Loss** Many methods that we have developed so far make use of only pair-wise information. For example, the methods developed using Node2Vec in Section A.2 are based on the skip-gram model, which learns embedding of nodes while maximizing the conditional probability of a node given another node in a context (Eq. 14). Similarly, the spectral methods (Section A.3) are inherently two dimensional as they are based on matrix. Same is the case with skip-gram based sentence embedding (h2v-dbow). However, sentence embedding based on the DM architecture (h2v-dm) maximizes the conditional probability of a word given the previous set of words (context) without breaking this context (by concatenating or averaging the embedding of the previous words). However, the tensors principally capture the joint distribution over various cardinality hyperedges unlike conditional distribution like sentence embedding (h2v-dm) (which are more appropriate for sequences).

**Improvements** There are a few possible improvements which we can see in various approaches. In tensor approach when we extract a sub-hypergraph of a given \( k \) size hyperedges, it is possible that a significant part of nodes are left out (as they might not be a part of any \( k \) size hyperedge). This can possibly cause a cold start problem in the corresponding tensor decomposition. However, this does not mean that we will not get embedding for those nodes. It simply means that we might not achieve embeddings for these nodes from the decomposition of \( k \)-way tensors. But we can get embedding for these nodes from other tensors for a different \( k \). However, we can mitigate the cold start problem in a principled manner using a graph regularization term (\( \mathbf{U}^T L_k \mathbf{U}^j \)) in our tensor decomposition objective function. Here, \( \mathbf{L} \) is the graph laplacian of our choice. For example we can use \( \mathbf{L}_{hyp} \) or \( \mathbf{L}_{graph} \) when we perform tensor decomposition for node embeddings and use \( \mathbf{L}_{inv} \) or \( \mathbf{L}_{dual} \) for hyperedge embeddings. Lastly, the hypergraph autoencoder (h2v-auto) can only produce hyperedge embeddings and not the node embeddings. In this sense it is feeble to other methods.

**Highlight** Another distinguishing feature of our work is that we leverage the existing structure (group structure) within the data. Recent attempts along these lines in network literature (Grover & Leskovec 2016; Tang et al. 2015b; Perozzi et al. 2014) argue that language models have ready-made context in the form of sentences or paragraphs to train the model, which are not available in networks, and therefore, they propose different ways to generate this context. In contrast, we focus on networks where the context is already present, e.g. collaboration networks where collaborative teams are hyperedges, or language hyper-networks where sentences are hyperedges.

5 **RELATED WORKS**

Hypergraphs were studied rigorously by Berge (1976; 1984) as a generalization of graphs and directed hypergraphs have been introduced by Bretto (2013). Hypergraph were argued for the first time as a model to naturally capture higher-order relationships between sets of objects across variety of domains by Estrada & Rodriguez-Velazquez (2005). Hypergraphs have been used to model complex networks in different fields including biology (Klatt et al. 2009), databases (Fagin 1983) and data mining (Han et al. 1997; Zhou et al. 2007). Within machine learning, algorithms guided by the structure of hypergraph were introduced by Zhou et al. (2006) and have found applications in a variety of domains (Tian et al. 2009; Gao et al. 2013; Li & Li 2013; Sharma et al. 2015; 2017). Simplicial complex (Munkres 1984) based view of hypergraph using *hasse lattice* (Skiena 1990) within machine learning has recently been proposed by Sharma et al. (2017).

**Representation learning (RL)** (Bengio et al. 2013) focuses on learning features (geometry) of the data (topology). Machine learning algorithms make use of these features for prediction. Traditionally these features are readily available within the data-set or are engineered manually. However, this is often a tedious and human labor-intensive process. RL addresses this issue by learning features automatically in a task-dependent supervised or a task-independent unsupervised manner.

**Node Representations in Graph:** Traditionally unsupervised node embedding learning has been done using latent models like matrix factorization (Ahmed et al. 2013) or by community detection (Tang & Liu 2011) based techniques for networks (Roweis & Saul 2000; Belkin & Niyogi 2001; Tenenbaum et al. 2000; Cox & Cox 2000). In each case there is a vector of features learned for
a node, each of whose entries reflects node’s association with some latent dimension or a network community. More recently, there has been a revived interest in graph embedding in form of context oriented techniques (Perozzi et al., 2014; Tang et al., 2015b; Grover & Leskovec, 2016). These techniques are inspired by recent unsupervised RL methods in NLP (Mikolov et al., 2013b; Le & Mikolov, 2014) where word embeddings are learned that are similar to words in a given neighborhood or context. These techniques differ in the manner they generate this context as well as in the objective which they optimize. Also there are supervised algorithms learn embeddings which are optimal for the specific task at hand. This results in high accuracy but incurs significant computational cost for training. Recently, several supervised learning algorithms have been proposed for network analysis (Tian et al., 2014; Xiaoyi et al., 2013) and for text networks in a semi-supervised setting (Tang et al., 2015a). Finally, we refer readers to a very recent and comprehensive survey on graph embedding methods by Cai et al. (2017).

Node Representations in Hypergraph: Learning embeddings for nodes within a hypergraph while incorporating the hypergraph topology using proxy graphs is introduced by Zhou et al. (2006). Using graph proxy destroys the hypredge-level joint information and thus, incur loss of information. Also, Agarwal et al. (2006) squarely criticize that such representations can be learned by constructing graphs, which are proxies for the hypergraph structure.

Set Representations: RL for sets using neural networks has been proposed recently (Vinyals et al., 2016), where a memory network is used to compose features sequentially but in an order invariant manner. In their very recent paper, Rezatofighi et al. (2016) have tried to answer this set ordering issue by the use of random set theory. However, they do not consider embedding but focus on learning set-level probabilities. More importantly, both of these works, do not consider the hypergraph structure of overlapping sets which is the main focus of this paper.

Tensors For comprehensive view of tensors, tensor decomposition as well as applications we refer to the survey by Kolda & Bader (2009). The connection between $k$-way tensor and $k$-uniform hypergraph eigen values was established by Qi (2005). The use of $k$-way symmetric tensor and their non-negative decomposition for uniform hypergraph partitioning was first introduced by Shashua et al. (2006). But they are again restricted to uniform hypergraphs.

6 Conclusion

In this paper we have proposed two methods to generate higher-order representations for both hyperedges (representing sets of nodes) and hypergraph nodes (that also take into account the hypergraph structure). First, is an auto-encoder based deep-learning method and second, is a tensor-based algebraic method. Both learning models are unique in the manner they leverage the existing structure present in network data as context. While introducing a new idea of a dual tensors corresponding to the hypergraph dual, we develop a novel approach of using factors from joint decomposition of $k$-way tensors corresponding to $k$-uniform sub-hypergraphs, as generic node & hyperedge representations. We show that that both methods perform comparably with several other baselines in terms of accuracy. We also observe that the proposed tensor based methods are more efficient and also have a natural hypergraph theoretic interpretation; unlike deep learning based black-box approach. We therefore, argue that we have proposed more general methods which are principally suited for hypergraphs (and therefore also for graphs) while maintaining accuracy and efficiency.
REFERENCES

Pubmed data. [https://www.ncbi.nlm.nih.gov/pubmed/]


E.H. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering based on association rule hypergraphs. University of Minnesota, Department of Computer Science, 1997.


Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher Manning, Andrew Ng, and Christopher Potts. Parsing With Compositional Vector Grammars. In EMNLP. 2013.


As mentioned before, the most commonly used model for studying complex interactions in networks is graphs, where each edge represents a dyadic interaction between nodes (Strogatz, 2001). Therefore, even if the original interaction is not dyadic like a set of researchers (3 or more) collaborating on a publication, we shall break down this joint interaction into dyadic interactions. Even though a large number of complex network data naturally occurs as hypergraphs (Estrada & Rodriguez-Velazquez, 2005), the popularity of “think like dyadic edges” and not like sets seems to us as a hindrance, specifically when the end aim is to model the joint distribution at the level of sets. In this paper, as it is obvious, we go by “think like set” paradigm, specifically, when the data is naturally occurring as network of sets (i.e., a hypergraph). Therefore, if we think of hyperedge as a set, a natural question arises is that are there any existing techniques that learn embeddings at set level? We, henceforth, explore the most popular discrete data on which representation learning has been applied is text data.

Recently, Le & Mikolov (2014) proposed two representation learning methods for sentences as shown in Figure 4: a) a distributed memory (DM) model, and b) a distributed bag of words (DBOW) model. In the DM model, every sentence in the dataset $G$ is represented by a $d$ dimensional vector in a shared lookup matrix $S \in \mathbb{R}^{\lvert G \rvert \times d}$. Similarly, every word in the vocabulary $\Omega$ is represented by a $d$ dimensional vector in another shared lookup matrix $L \in \mathbb{R}^{\lvert \Omega \rvert \times d}$. Given an input sentence $v = (v_1, v_2 \cdots v_m)$, the corresponding sentence vector from $S$ and the word vectors from $L$ are averaged to predict the next word in a context. More formally, let $\phi$ denote the mapping from sentence and word ids to their respective vectors in $S$ and $L$, the DM model minimizes the following objective:

$$J(\phi) = -\sum_{t=k}^{m-k} \log P(v_t|v_{t-k+1}, \cdots, v_{t-1})$$  \hspace{1cm} (8)

$$= -\sum_{t=k}^{m-k} \log \frac{\exp(\omega(v_t)^T z)}{\sum_i \exp(\omega(v_i)^T z)}$$  \hspace{1cm} (9)

where $z$ is the average of $\phi(v)$, $\phi(v_{t-k+1}), \cdots, \phi(v_{t-1})$ input vectors, and $\omega(v_t)$ is the output vector representation of word $v_t \in \Omega$. The sentence vector $\phi(v)$ is shared across all (sliding window) contexts extracted from the same sentence, thus acts as a distributed memory.

Instead of predicting the next word in the context, the DBOW model predicts the words in the context independently given the sentence id as input. More formally, DBOW minimizes the following objective:
Both the methods take sentences as input and return embeddings for words as well as the sentence. To apply the sentence embedding models directly to hyperedges, as our first method we generate a proxy sentence for each hyperedge by leveraging the contextual information in the hyperedge.

A proxy sentence $v_i$ is formed for each hyperedge $g_i \in G$ as a sequence made by concatenating all the permutations of the nodes (as words) in the hyperedge and further repeating this sequence as many time this hyperedge occurred. For example, for a three node hyperedge $g_i = \{1, 4, 7\}$ which has occurred two times we make the following sentence $v_i$: 
\{1,4,7,1,4,7,4,1,7,4,7,1,4,7,4,1,7,4,7,1,4\}

A proxy sentence $v_i$ is formed for each hyperedge $g_i \in G$ as a sequence made by concatenating all the permutations of the nodes (as words) in the hyperedge and further repeating this sequence as many time this hyperedge occurred. For example, for a three node hyperedge $g_i = \{1, 4, 7\}$ which has occurred two times we make the following sentence $v_i$: 
\{1,4,7,1,4,7,4,1,7,4,7,1,4,7,4,1,7,4,7,1,4\}

A.2 HYPEREDGE2VEC USING NODE2VEC

Grover & Leskovec [2016] propose a representation method for nodes in a graph called Node2Vec, which uses the skip-gram model [Mikolov et al. [2013b]] with the intuition that neighborhood nodes within a graph should have similar representations. The objective of the skip-gram model for graphs can be defined as:

$$J(\phi) = - \sum_{v \in V} \log P(N(v)|\phi(v))$$

$$= - \sum_{v \in V} \sum_{n_i \in N(v)} \log P(n_i|\phi(v))$$

$$= - \sum_{v \in V} \sum_{n_i \in N(v)} \log \frac{\exp(\omega(n_i)^T \phi(v))}{\sum_x \exp(\omega(x)^T \phi(v))}$$

where as before, $\phi$ and $\omega$ denote the input and the output vector representations of the nodes. The neighboring nodes $N(v)$ form the context for node $v$. Node2Vec uses a biased random walk which adaptively combines breadth first search (BFS) and depth first search (DFS) to find the neighborhood of a node. The walk attempts to capture two properties of a graph often used for prediction tasks in networks: i) homophily and ii) structural equivalence. According to homophily, nodes in the same group or community should have similar representations. Structural equivalence suggests that nodes with similar structural roles (hub, bridge) should have similar representations. In a real-world network, nodes exhibit mixed properties.

Our aim is to find both node and hyperedge level embeddings by taking into account the hypergraph structure. For the former, we leverage the adjacency matrix associated with a hypergraph [Zhou et al. [2006]], which is defined as:

$$A_{\text{hyp}} = H^T W_e H - D_v$$

where $W_e$ is a diagonal matrix containing the weights of each hyperedge and $D_v$ is a diagonal matrix containing the degree of each vertex. We take the weight of a hyperedge as its occurrence number, i.e. $W_e(i, i) = R(g_i), \forall i \in \{1, \ldots, n\}$. The adjacency matrix ($A_{\text{hyp}} \in \mathbb{R}^{V \times V}$) associates a weight between a pair of nodes while taking into account the weights of all the hyperedges that encompass a pair of nodes. The weighted graph associated with $A_{\text{hyp}}$ in some sense serves as a proxy to the actual hypergraph structure. We can provide $A_{\text{hyp}}$ as input to Node2Vec and
hypothesize that the random walk over this proxy hypergraph should allow the skip-gram model to learn more meaningful node level embeddings which can be combined to construct hyperedge level embeddings. We refer to these node level embeddings as N2V-hyp.

We still have not met our second objective to learn hyperedge embeddings directly. We again wish to leverage Node2Vec for our purpose. However, Node2Vec only works for graphs by performing random walk over nodes. Therefore, we ask ourselves the question, that can we treat hyperedges as nodes? There can be other ways of doing so, but here we suggest two techniques. In the first technique, we simply consider the hypergraph dual \cite{Berge1984}, whose incidence matrix is \( H_{\text{dual}} = H^T \). The adjacency matrix associated with the hypergraph dual is:

\[
A_{\text{dual}} = H_{\text{dual}}^T W_v H_{\text{dual}} - D_e = HW_v H^T - D_e
\]

where \( W_v \) is a diagonal matrix containing the weights of each node and \( D_e \) is a diagonal matrix containing the degree of each hyperedge. We assume no weights on the nodes and take \( W_v = I \). The matrix \( A_{\text{dual}} \in \mathbb{R}^{G \times G} \) represents another hypergraph, but the roles of nodes and hyperedges have now switched. For example, in case of words and sentences, the hyperedges were sentences, but in the hypergraph dual the words become hyperedges and the nodes within a word’s hyperedge represent all the sentences in which the word has appeared. We again give \( A_{\text{dual}} \) (i.e., a graph proxy for the dual hypergraph) as input to Node2Vec, but this time we get output as the embeddings for the hyperedges associated with the nodes in dual. We refer to these hyperedge embeddings as h2v-dual.

In the second technique, we consider the following adjacency matrix \( A_{\text{inv}} \in \mathbb{R}^{G \times G} \):

\[
A_{\text{inv}} = HH^T
\]

associated with what we refer to as the inverted hypergraph. This inverted hypergraph is a graph (unlike the dual which is a hypergraph) and there is an edge between two nodes if the hyperedges corresponding to the nodes in the original hypergraph have nodes in common. Weight of this edge is the number of common nodes. We again give \( A_{\text{inv}} \) as input to Node2Vec to get embeddings for the hyperedges associated with the nodes in inverted hypergraph. We refer to these hyperedge embeddings as h2v-inv.

### A.3 Hyperedge2Vec Using Spectral Embeddings

These set of methods extract embeddings as the eigenvectors associated with Laplacian matrices corresponding to the various adjacency matrices discussed in the previous section. We consider the following graph Laplacians:

\[
L_{\text{graph}} = I - D_v^{-1/2} A_{\text{graph}} D_v^{-1/2}
\]

\[
L_{\text{hyp}} = I - D_v^{-1/2} A_{\text{hyp}} D_v^{-1/2}
\]

\[
L_{\text{inv}} = I - D_e^{-1/2} A_{\text{inv}} D_e^{-1/2}
\]

\[
L_{\text{dual}} = I - D_e^{-1/2} A_{\text{dual}} D_e^{-1/2}
\]

We get the \( d \) eigenvectors associated with the smallest \( d \) eigenvalues of the above graph Laplacians as the embeddings. We get vertex embeddings using Eq. \[19\] and hyperedge embedding using Eq. \[21\] and we refer to them together as e2v-hyp. Similarly, we get another pair of vertex embeddings (using Eq. \[18\]) and hyperedge embedding (using Eq. \[21\]). We refer to the later pair as e2v.