PyWalk - semi-supervised label prediction for graph nodes

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Satti Sai Chandan Reddy , Srikar Paruchuru
(150101059 , 150101044)

under the guidance of

Amit Awekar

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Pywalk - semi-supervised label prediction for graph nodes**" is a bonafide work of **Satti Sai Chandan Reddy (Roll No. 150101059)**, **Srikar Paruchuru (Roll No. 150101044)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Prof. Amit Awekar**

Assistant Professor,

Department of Computer Science & Engineering,

Indian Institute of Technology Guwahati, Assam.

Contents

List of Figures

Chapter 1

Problem Statement

1.1 Overview:

Consider a social network of users, in which the political orientation of some users has to be determined (for example Democrats and Republican).

The above can be modelled as a graph wherein each user is a node and their connections are represented by the edges and their political inclination as their label. Note that the labels of a node depend on the neighbourhood in which it lies . Generally, the labels of some nodes are known and we want to predict the labels for the rest.

1.2 Concrete problem definition :

Given a graph $G = (V,E)$ with a label map $L : V' \rightarrow$L(V') , $V' \subseteq V$ (i.e. only a few of the labels are known with L as function on vertices mapping to their labels).

Assumption: L is assumed to just depend on the structure of the graph only.

The objective is to learn the labels of the remaining vertices (V-V').

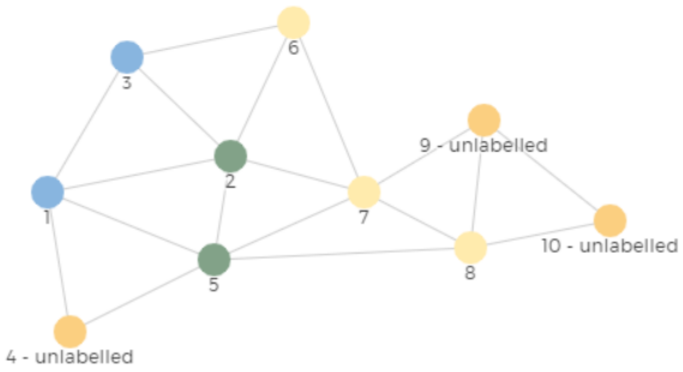


Fig. 1.1 (e) example of a graph with labelled nodes and a few unlabelled nodes

Chapter 2

Review of Prior Works

Almost all approaches outlined below obey the following template: They generate a representation for each node in the graph, called the node embeddings. These are generated in order to ensure that like nodes have similar embeddings. Thus we have the < embedding , label > tuples which can be used to train a learning model (like regression) to predict the labels for unlabelled nodes.

However, all the algorithms below outline the approaches to generate just the embeddings . Node embedding research in the recent years is mostly from 2 paradigms , random-walk based and neighbourhood based convolution approaches.

2.1 Convolution based approaches

The intuition behind these approaches is that they generate embeddings for a node by aggregating information from its local neighborhood

2.1.1 Graph Convolution Network(GCN) [KW16]

With the heavy success of CNN to the image domain,questions of whether these could be applied on other generalized domains like graphs is a very important consideration. Though, generalizing them to work on graphs whose structure was unknown is a challenging problem.

Why a Graph convolution?

Since, it is not direct, how to apply convolutions on random structured graphs, we use a view in the spectral domain.In Fourier domain, convolutions are point-wise multiplication of the Fourier transform. Also, Fourier transformation can be taken forward to graphs and this point-wise multiplication is the general graph convolution.

Hence, as defined consider the features $x \in R^n$ with a parameterized filter g_θ as:

$$g_\theta * x = U g_\theta U^T x \tag{2.1}$$

Here, U is the eigenvector of the normalized graph laplacian $L = D^{-1/2} A D^{-1/2}$. Of Course it could be normalized a little differently as well, but for a symmetric matrix we have used the above.

Note, that in (3.1) the multiplication take $O(N^2)$ and calculating the eigenvectors of a large graph also can be expensive. Hence, we use approximate filters for reducing complexity.

Approach

First, we discuss the mathematical approach and then move to an intuitive explanation. We can limit the conv kernel by representing it as a Chebyshev polynomial in kth order:

$$g_{\theta'} \star x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x \,,$$

Fig. 2.1 (a) Approximation with T_k being the chebyshev polynomial

The above is a Kth order polynomial in the laplacian which means it takes note of nodes which are only K distance away from the current node. However, in this paper they further take the approximation as K = 1 which is linear.

This linear layer wise model that is now obtained by stacking the conv filter of fig 3.1 can still recover a lot of data while not being constrained by the K parameter in the previous filter. They also perform parameter reduction by making 2 parameter $\theta = \theta_0 = -\theta_1$. Finally:

$$g_\theta \star x \approx \theta \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) x \,,$$

Fig. 2.2 (a) after constraining K and the parameters

We now move onto an intuitive approach which derives the same idea while relating to the classic CNN.

First, we redefine GCN as a neural network now. Given a graph $G = (E, V)$. the model take in:

- an input feature matrix NXF^0 feature matrix, X, where N is the number of nodes and F^0 is the input features.
- Also, A which is the adjacency matrix representation of G with size NXN.

Every hidden layer can be written as a non linear function: $H^i = f(H^{i-1}, A)$. The first layer is X while the last layer is Z with size NXF where F is the number of features on the output layer, it may be z for a single label.

Consider the propagation "f" as :

$$f(H^i, A) = \sigma(AH^iW^i)$$

Fig. 2.3 (a) a simple propagation rule

W^i is the weight matrix of dimensions F^iXF^{i+1} and σ is the activation function. However, there are a few problems with the above propagation rule. If you taken $W = Iand\sigma asidentity$ we can easily observe that the:
value of f comes to be the average of the neighbourhood nodes.

- However, not the node itself, hence we add a connection to itself by adding $A = A+I$.
- Also, that different nodes have different degrees and they must be scaled hence, we multiply with $D^{-}1$ which is the inverse of the degree matrix. For a symmetric matrix we normalize with D on either sides and finally obtain.

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

Fig. 2.4 (a) Final propagation rule which is exactly as fig 3.2

Upon using the above propagation rule, even with initializing random weights and taking identity as the feature matrix X we obtained embedding which closely represent the actual structure. In the paper they have explained the reason for this is that the GCN is a generalized version of the Weisfeiler-Lehman algorithm on graphs.

Conclusion

For semi-supervised classification, we apply cross-entropy loss over the labelled examples, with the weights trained by gradient descent.

Finally, we state the complexity of the filter that we have used as : $O(|\epsilon|FC)$ where F is the input feature dimension, C is the ouput feature dimension and ϵ is the edges. Also, note that the product of $A * X$ is done efficiently and note taken as a comple N X N by N X F multiplication.Here, A has a sparse matrix representation and hence memory will be in $O(\epsilon)$

2.1.2 GraphSAGE [HYL17]

Compared to the GCN paper which was a transductive model which means that all possible nodes will be used in training given only a few labels it makes prediction on the unlabelled nodes. This is especially useful in node embeddings, since the entire graph may contribute something to the structure, however it does not generalize to unseen nodes or to new graphs.

Hence, this paper focuses on generating low dimension features of output which generalize to new nodes or completely new graphs will same input feature dimension.

Objective

The aim is to learn a set of aggregator functions which use the node input features and also learn the structure of the graph to predict new nodes or sub graphs and also for new graphs.

How inductive?

Each aggregator function collects information of node features from the k-th neighbourhood of the node as in the fig 3.5. We mentioned that is was an inductive model which can generalize to unseen nodes and graphs, this happens because of the learned aggregator functions.

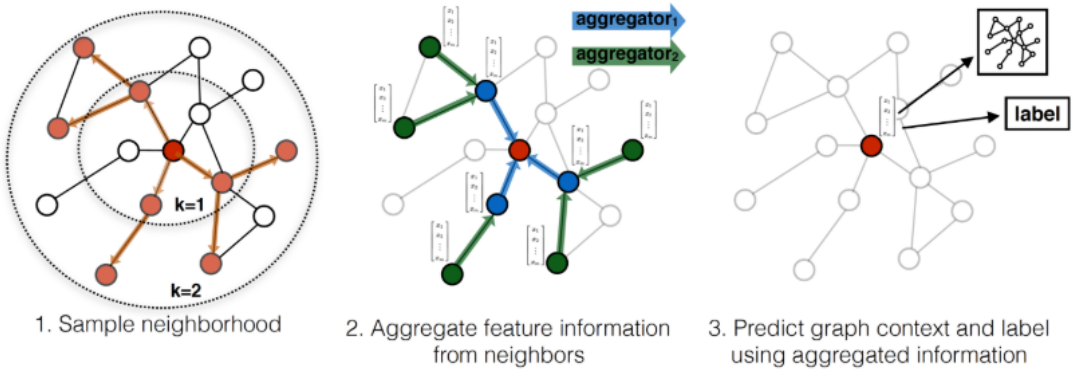


Fig. 2.5 (a) Aggregation approach in GraphSAGE

GraphSAGE can be trained in both a completely supervised approach as well as an unsupervised loss function.

Approach

Firstly, assuming we have the aggreagator function we show the algorithm for the forward pass of the model, similar to that in CNN with K layers. 2.6 shows that at each k nodes

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Fig. 2.6 (a) Generating embeddings for the nodes

start pooling the node feature information from the different depths of the graph at each level.

What the algorithm does is considering the graph $G = (E, V)$ and the input features as x_i as input. At each step i.e. at each k we have for all vertices, aggregation of the the neighbourhood node features into the vector $h_{N_v}^k$ and concatenating the current vector representation h_v^{k-1} and the previous vector into the new representation h_v^k through a neural network layer thus multiplying with weights and a non linear activation.

obtaining the parameters

Since, the unsupervised domain cannot use label information the paper uses a loss function to model the output feature representation z_u. Also, the aggregation functions and weight matrices are trained by gradient descent.

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right),$$

Fig. 2.7 (a) The loss function

Here, v is a node that occurs at a constant random walk distance from u. v_n are samples from a negative sampling distribution P_n.

aggeregator

Since, the aggregator is supposed to aggregate the feature representations of nearby nodes which have no specific ordering. The aggregator is supposed to be invariant under changes or ordering. The paper identifies 3 aggregator functions:

1. MEAN aggregator: It takes the vectorwise mean in h_v^k and combining the lines 4 and 5 in algorithm 1 with the MEAN aggregator becomes the propagation rule that was discussed in the GCN approach.

2. LSTM aggregator: Although, not symmetric we take a random ordering of the nodes of the nodes neighbours. It expresses more information than the MEAN aggregator.

3. POOLing aggregator: Here, each neighbours vector is sent through a fully connected layer; which is later sent through a MAX-POOL layer. It is symmetric and easily trainable and is hence used by the paper in their final results. 2.8

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\}),$$

Fig. 2.8 (a) Pooling aggregator

2.2 Random-walk based approaches

Other methods, focused on the fact that nodes have similar embedding if the occur together in the same random walk.

2.2.1 Deepwalk [PAS14]

Overview

Deepwalk generates distributed embeddings of each node in a graph by generating local truncated random walks which are fed into a Skipgram model . The walks are treated as

sentences by skipgram and the embedding generated is based on the context of the word in the sentence i.e the role of a node in the path. Let V be the set of vertices. Let Φ be the embedding function.

The Goal is to maximize the likelihood of observing vertex v_i given all previous vertex embeddings in the random walk.

$$\Pr\Big(v_i \mid \big(\Phi(v_1), \Phi(v_2), \cdots, \Phi(v_{i-1})\big)\Big)$$

However , the above metric is clearly computationally expensive since it is proportional to walk length. A recent relaxation in language modelling removes the ordering constraint which yields the new optimization goal.

$$\underset{\Phi}{\text{minimize}} \quad -\log \Pr\big(\{v_{i-w}, \cdots, v_{i+w}\} \setminus v_i \mid \Phi(v_i)\big)$$

w is the window size

Skipgram

Skipgram is a language model that maximizes the co-occurrence probability among the words that appear within a window, w, in a sentence i.e Given a representation of a node, maximize probability of neighbours in the walk. Assuming independence the above equation is simplified as follows.Hierarchical Softmax is used as the posterior distribution.

$$\underset{\Phi}{\text{minimize}} \quad -\log \Pr\big(\{v_{i-w}, \cdots, v_{i+w}\} \setminus v_i \mid \Phi(v_i)\big)$$

Algorithm

The pseudo code for deepwalk and skipgram :

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
    SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
  end for
end for

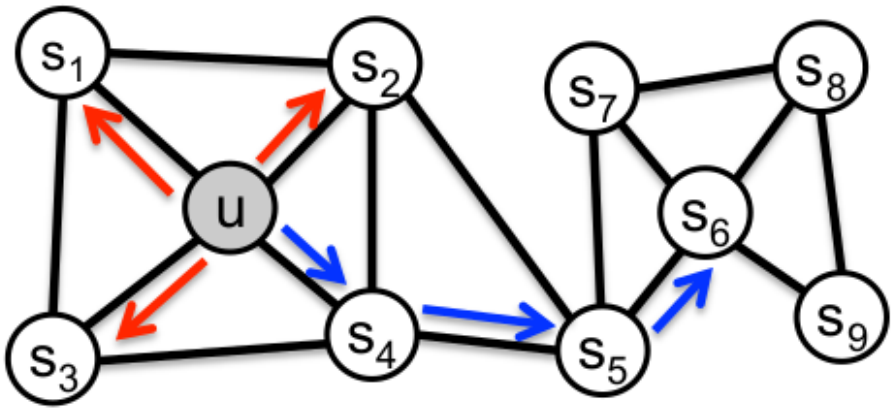
Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

for each $v_j \in \mathcal{W}_{v_i}$ **do**
  for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
    $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
    $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
  end for
end for

2.2.2 Node2vec [GL16]

Overview

The definition of equivalence in graph could be made flexible i.e consider the graph below The pair (u,s1) lies in the same strongly connected community (homophily) while (u,s6) are structurally similar(they both act as junctions in their communities) despite being far away from each other.The node2vec algorithm modifies deepwalk algorithm in order to



accommodate for the new definitions. The algorithm defines two parameters which defines similarity.

- **p** : controls likelihood of a random walk re-visiting a node . If p<1 , it means high chances of revisit , hence the generated random walk would be local.
- **q** : helps distinguish between inward and outward nodes . If q<1 , it means outward exploration is encouraged ensuring global random walks.

The transition probability matrix π is determined using p and q metrics. Hence deepwalk is a special case of Node2vec where p=1 and q=1.

Algorithm

The pseudocode for Node2vec :

Note : While deepwalk used skipgram model in word2vec , node2vec employs a more recent implementation of word2vec that uses negative sampling to arrive at the embeddings .The function AliasSample refers to that.

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
  π = PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  for $iter = 1$ **to** r **do**
    for all nodes $u \in V$ **do**
      $walk$ = node2vecWalk(G', u, l)
      Append $walk$ to $walks$
  f = StochasticGradientDescent(k, d, $walks$)
  return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
  Inititalize $walk$ to $[u]$
  for $walk\_iter = 1$ **to** l **do**
    $curr = walk[-1]$
    V_{curr} = GetNeighbors($curr$, G')
    s = AliasSample(V_{curr}, π)
    Append s to $walk$
  return $walk$

2.2.3 HARP - Hierarchical Representation Learning for networks [CPHS18]

Overview:

HARP is a hierarchical model which utilizes any of the previous explained random-walk approaches and aims to improve them by enforcing global structural information through graph coarsening.

Algorithm

HARP consists of three parts 1. graph representation, 2. graph embedding and 3. representation refinement. The pseduocode is as follows:

Algorithm 1 HARP($G, Embed()$)

Input:
graph $G(V, E)$
arbitrary graph embedding algorithm EMBED()
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

$G_0, G_1, \cdots, G_L \leftarrow$ GRAPHCOARSENING(G)
Initialize Φ'_{G_L} by assigning zeros
$\Phi_{G_L} \leftarrow$ EMBED(G_L, Φ'_{G_L})
for $i = L - 1$ to 0 **do**
    $\Phi'_{G_i} \leftarrow$ PROLONGATE$(\Phi_{G_{i+1}}, G_{i+1}, G_i)$
    $\Phi_{G_i} \leftarrow$ EMBED(G_i, Φ'_{G_i})
end for
return Φ_{G_0}

Graph Coarsening:

The first aim is to coarsen the graph by decreasing the number of vertices in such a way that the global information is still preserved, this is done by a star collapse followed by and edge collapse.

Graph Embedding:

The embeddings are then generated for the top layer of the graph with random initialization using any of the approaches like Deepwalk and node2vec.

Graph Representation and Prolongation:

This graph embedding of the layer above say $G_(i+1)$ is then initialized onto the graph G_i since graph in the upper layers are the children or the same nodes.

Graph Embedding on Original graph:

This is continued till a refined embedding is obtained on the base graph which is then fed to regression algorithm to evaluate similar to other methods.

Algorithm 2 GraphCoarsening(G)

Input: graph $G(V, E)$
Output: Series of Coarsened Graphs $G_0, G_1, \cdots, G_L$

1: $L \leftarrow 0$
2: $G_0 \leftarrow G$
3: **while** $|V_L| \geq threshold$ **do**
4: $\quad L \leftarrow L + 1$
5: $\quad G_L \leftarrow$ EDGECOLLAPSE(STARCOLLAPSE(G))
6: **end while**
7: **return** $G_0, G_1, \cdots, G_L$

Fig. 2.9 (d) HARP pseudocode

2.3 Paradigm Comparison and Insights

Random walk approaches with the exception of HARP are parallelizable , scalable and online while convolution based approaches are sequential and more accurate by virtue of their neighbourhood aggregation property . HARP is a recent improvement on general random walk based approaches (like deepwalk and node2vec) employing hierarchy as means to access global structural information.

We have identified potential areas in the random walk approach paradigm that could be used to come up with interesting results . The key ideas that we identified are:

- Using random walks to record neighbourhood information
- Using hierarchy to understand global information while preserving local information.
- Understand and explore the possibility of using a hierarchical structure to effectively condense the current graph information

Chapter 3

Algorithm and Implementation

Here, we provide the algorithm by first explaining the data structure we use and how we construct it, motivation for it and how training occurs on this new structure.

3.1 Algorithm :

3.2 Overview :

Flowchart showing the steps of the algorithm, starting from pyramid construction to random walk to the downstream learning to obtain labels for unknown nodes.

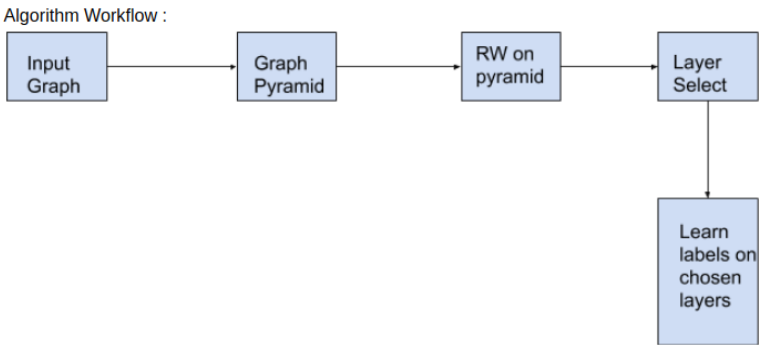


Fig. 3.1 (e) Pipeline representing the algorithm flow

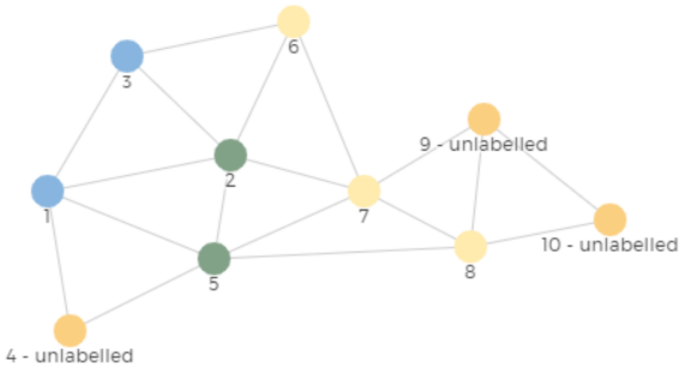


Fig. 3.2 (e) example of a graph with labelled nodes and a few unlabelled nodes

3.3 General Outline :

A hierarchical structure is constructed on top of the input graph G in such a way that a bunch of nearby nodes (all labelled with the same label say l) shall be coalesced into a single super node S . The idea is that the embedding for node S (say $\epsilon(S)$) , essentially represents the embedding of all the nodes it's connected to . Hence the tuple $< \epsilon(\mathrm{S})$, $l >$ can be fed into the embedding-label learning model instead of all the node embeddings in the sub graph. This hierarchical structure termed as the pyramid is more concretely defined in the following section.

3.4 The Pyramid :

An overview of the pyramid and it's key features.

- Base Layer/ Base graph : The input graph itself, on top which the pyramid is constructed . It is also called as the lowest layer / Layer 0 .
- Branching Factor b : The number of nodes that can coalesce into a super node.

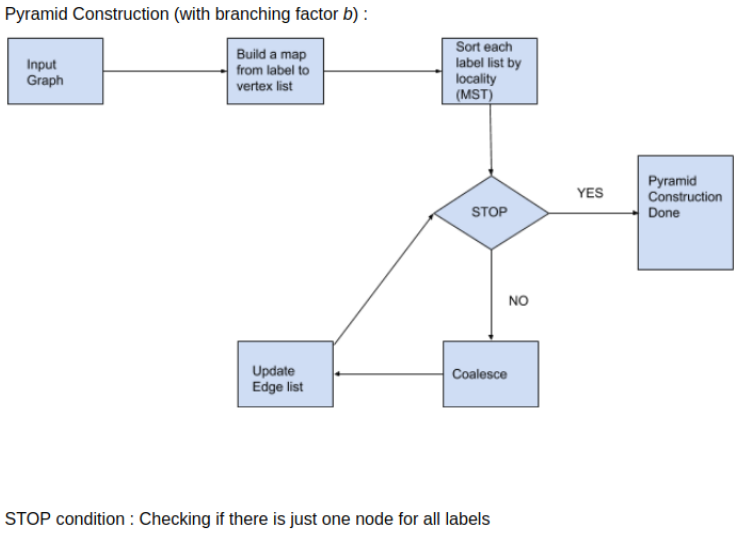


Fig. 3.3 (e) Workflow of the pyramid construction for the hierarchical approach

- Label Pooling : Only nodes that are of the same label l must be coalesced . The created super node S shall be assigned the label l. This is achieved by maintaining an inverse map of label to list of vertices LV: $l \rightarrow [V]$.

- Local Pooling : Like labelled nodes that are nearby must be coalesced into a super node . This is done in order to preserve the local patterns in the graph . Coalescing nodes that are very far way in the graph would render the created super node's embedding meaningless . Local Pooling is achieved by doing a MST walk on individual label graphs and then coalescing them into super nodes.

- Directed edges between Layers : The edge between a super node and the set of nodes it represents is directed from the super node towards each of it's nodes with a weight of 1.0 i.e The edges in between layers from top to bottom . This ensures that the random walks that start on on the base graph, stay on the base graph , hence preserving the integrity of the resulting embeddings while allowing the top layers get their own embeddings . If the edges were un directed , there could be random walks originating from a base graph node that end up traversing the top layers (that are

filled with dummy super nodes that we had created) . Then the resulting embeddings wouldn't be of much use.

- Edge weights in the upper Layers : Consider the supernode pair s1 and s2 . The weight assigned to this pair shall be the sum of weight of all the edges between vertices in s1 and vertices in s2 (which are in the lower layer).

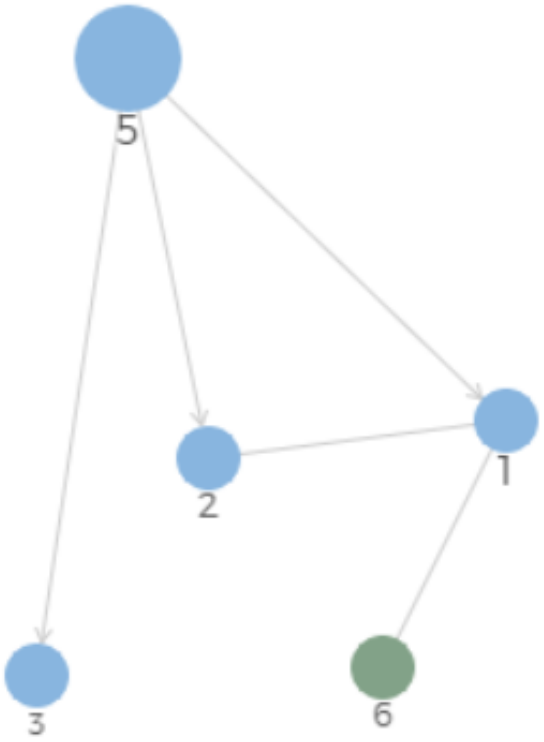


Fig. 3.4 (e) Supernode connection as directed edges

Hence , the Pyramid is essentially a Local-Label Pooled Weighted and Directed graph. The pseudo code for buildPyramid() is given below:

3.5 Pseudo Code :

Pseudo code of the entire Algorithm is shown below:

Notation

$G = (V, E, L)$, $G^p = (V^p, E^p, L^p)$ ▷ Graph as a tuple

$E : VxV \rightarrow R$ ▷ weighted edge list

$L : V^{'} \rightarrow L(V^{'})$ ▷ $V^{'} \subseteq V$, label map

$VS : V \rightarrow V$ ▷ vertex to supervertex map

$LV : l \rightarrow 2^V$ ▷ label to vertexlist map

$VE : V \rightarrow \epsilon(V)$ ▷ vertex to embedding map

$b : branchingfactor$

procedure ALGORITHM(V, E, L)

$LV = getInverse(L)$

$LV = LocalPool(LV)$

$G_p = pyramid(V, E, L, LV)$

$VE = WDnode2vec(G_p)$ ▷ Weighte Directed node2vec

$model = Train(VE)$

$accuracy = predict(model, V - V^{'})$ ▷ On vertices that aren't labelled

end procedure

Now, we will describe the functions used in the procedure:

First, the function for pooling of nodes in the lower layer to those in the upper layer

procedure LOCAL POOL(LV)

for label in LV **do**

$vertex.list = LV(label)$

$vertex.list = MST(vertexlist)$

end for

end procedure

Main function for constructing the pyramid data structure which is essentially a graph so adding new nodes and edges to the graph.

procedure PYRAMID(V, E, L, LV)

Require: $(V^p, E^p, L^p) = \{\}$ ▷ Start with pyramid as empty tuple

while True **do**

$(V^p, E^p, L^p).add(V, E, L)$

if stop(LV) **then** ▷ If single node in top layer stop

$return(V^p, E^p, L^p)$

end if

$V, L, LV, VS = Coalesce(V, L, LV, b)$

$E = next.Edge.list(E, VS)$

end while

end procedure

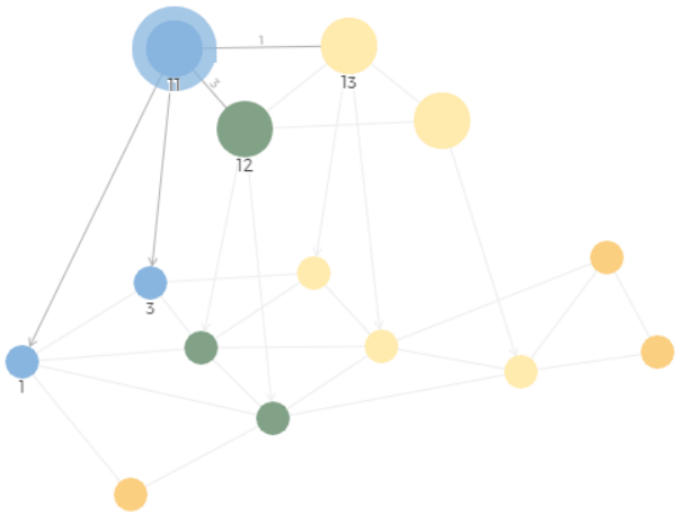


Fig. 3.5 (e) Connection between supernodes have weights as sum of weights in the lower layer

The coalesce function involves making the nodes in the upper layers and setting the label of the nodes from which it is obtained.

procedure COALESCE(V, L, LV, b)

Require: $(new.V) = \{\}$

Require: $(new.L) = \{\}$

Require: $(new.LV) = \{\}$

Require: $VS = \{\}$

for label in LV **do**

$vertex.list = LV[label]$

for every b nodes in vertex.list **do**

$vertex = new.node()$

$new.L[vertex] = label$

$new.LV[label].append[vertex]$

$VS[node] = vertex$

$new.V.append(vertex)$

end for

end for

return $new.V, new.L, new.LV, VS$

end procedure

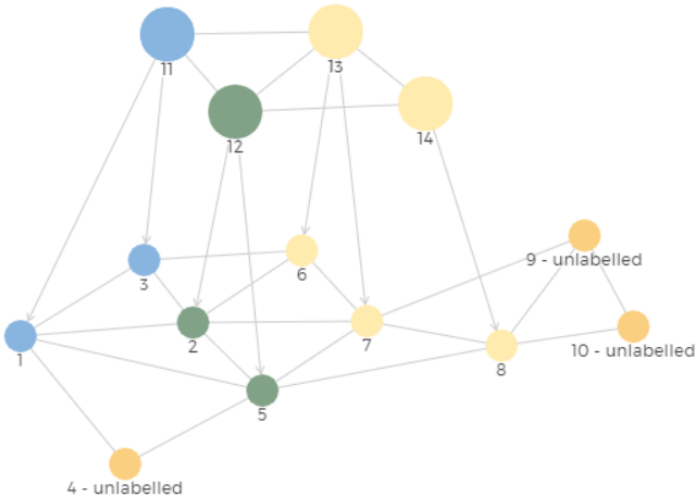


Fig. 3.6 (e) Pyramid data structure with layers 1 and 2

3.6 Points to note: in reference to HARP and our algorithm

- HARP aims to give a multi-level graph representation to obtain a better representation for the base graph, while our method focuses on creating multiple levels for the purpose of using lesser labels for training and also an absence of labels in the base layer would not affect our method as much.

- Also, HARP is built as an extension over the random walk approaches like Deepwalk and node2vec, while our algorithm aims to be a standalone improvement of general random walk appraoches using node2vec.

- Mainly, HARP builds levels on the entire graph since it does not leverage label information, while our algorithm leverages the label information of the graph ahead of training.

- Also, training is done on the entire pyramid structure as a whole while HARP evaluates each level in order to predict better representation for the base layer.

We also note from the image below that HARP does not have substantial increase on node2vec which is the best random-walk approach.

Also, note from the figure below, statistics of HARP from the datasets of DBLP, BlogCatalog, Citeseer respectively.

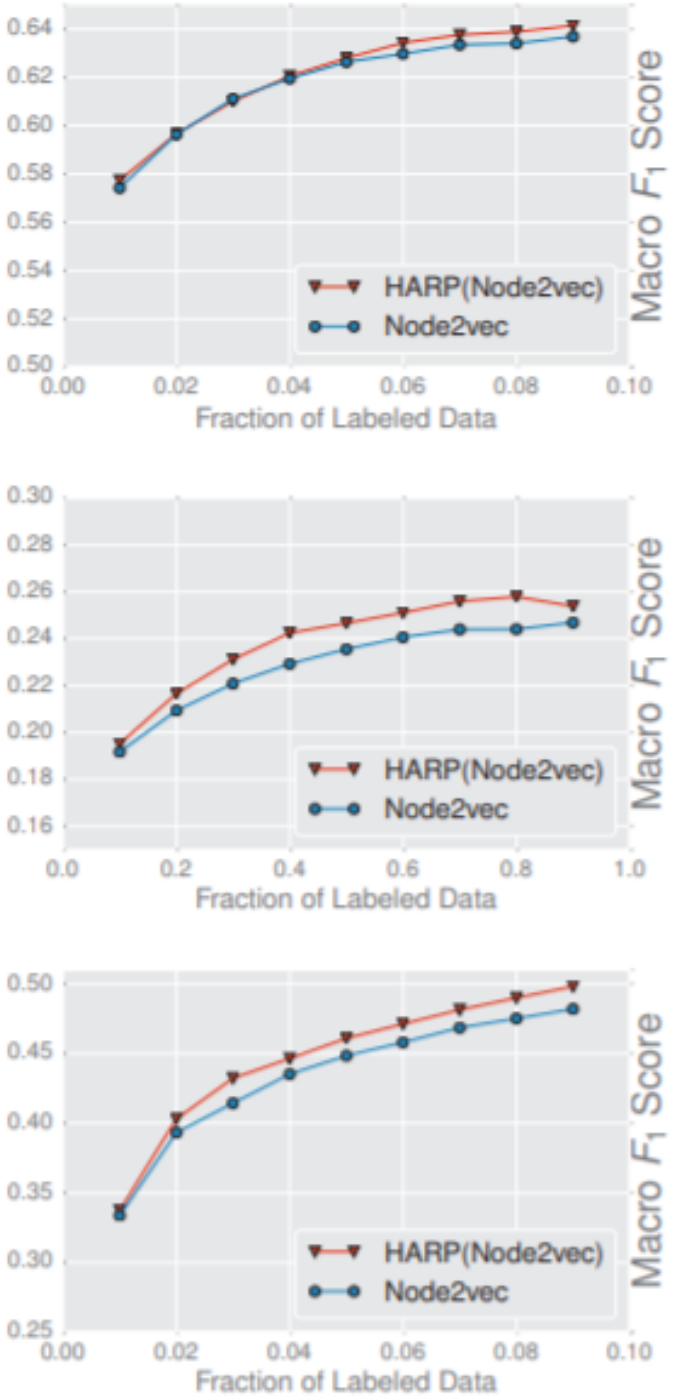


Fig. 3.7 (e) HARP statistics

Chapter 4

Results and Datasets

4.1 Implementation Details

- Random-walk algorithm used was node2vec with parameter values p = 1.0 , q = 1.0 and set flags for directed and weighted graph implementation.
- Learning model at the end was a "onevsrest(ovr)" Logistic Regression classifier.
- The window of the layers considered (i.e. without base layer) in pywalk algorithm implementation depends on the size of the pyramid, for CiteSeer: branching factor = 2, pyramid.size (layers) = 10, window = 2-5.
- For BlogCatalog: branching factor = 500, pyramid.size = 3, window = 0-3(entire pyramid).

Code repository : `www.github.com/srikarparuchuru/GraphClassify`

We tabulate the results here along with the datasets used:

4.2 Citeseer (Sen et al)[SNB+08]

CiteSeer is a citation network between publications in Computer Science. The paper has 6 labels showing who the paper belongs to as : Agents, AI, DB, IR, ML and HCI.

- Vertices: 3312
- Edges : 4732
- Classes : 6

First we will show the number of data points in training that we will use for the learning algorithm (regression model). We do not use the base layer, but we use layers 2-5 which are the coalesced forms. Also, we show the graph for label rate of 1-10% since the label rate for real world graphs do not exceed this limit.

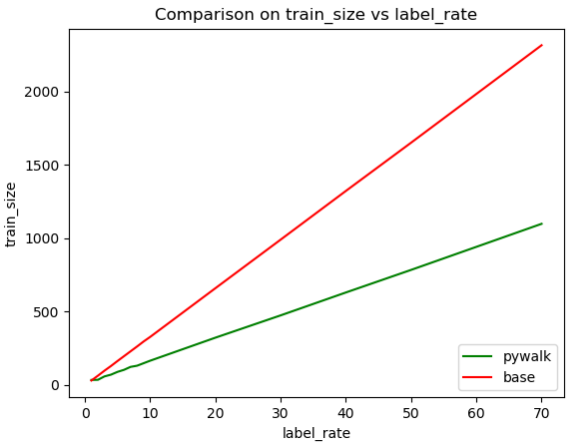


Fig. 4.1 (a) Training data used for label rate from 1-70

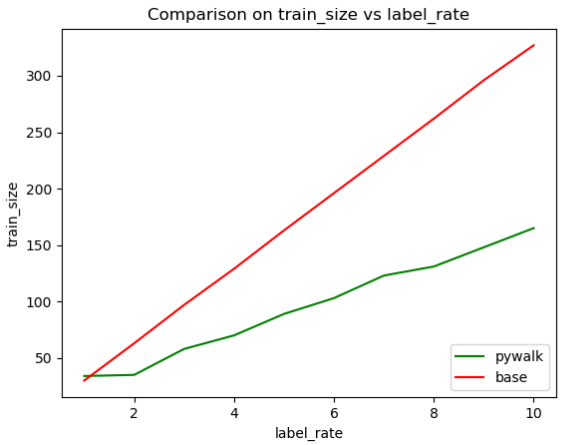


Fig. 4.2 (b) Training data for label rate from 1-10

Now, we show the comparison of macro-F1 score and micro-F1 score. Here, we compare

- 1. Our algorithm pywalk on layers 2-5 (other than base layer).
- 2. Node2vec which runs only on the base graph.
- 3. Our algorithm utlizing the entire pyramid strcucture.

4.2.1 Observation:

As we note from the random walk approach on HARP, which performs labelling rate from 1-15%, for the reason that most real-world graph would have only that amount of information (labels in our case) available to us.

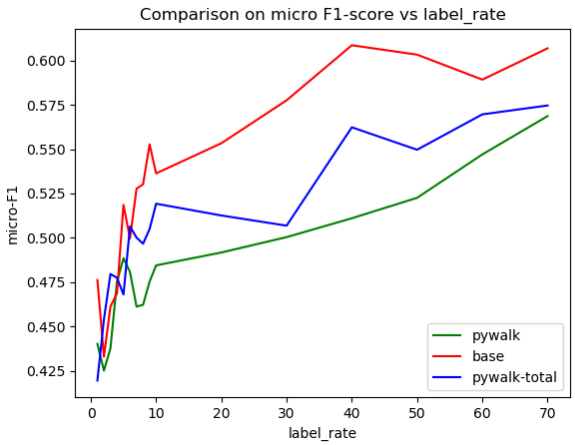


Fig. 4.3 (a) Macro F1 scores with label rate from 1-70

Fig. 4.4 (b) Macro F1 scores label rate from 1-10

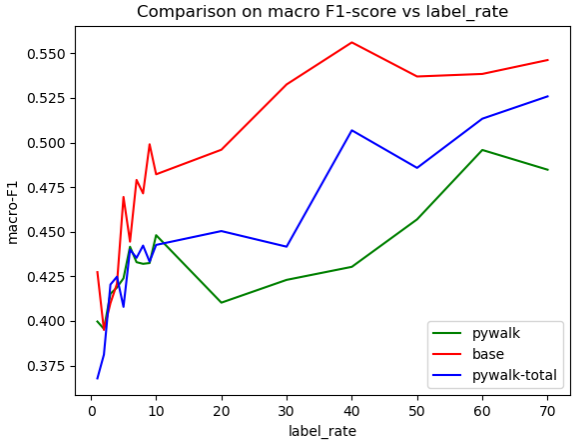


Fig. 4.5 (a) Macro F1 scores with label rate from 1-70

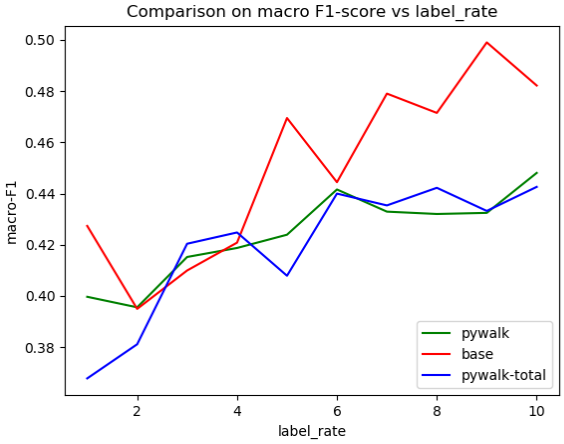


Fig. 4.6 (b) Macro F1 scores label rate from 1-10

Hence, seeing the right image in the graph plotted shows the classification rates which are relevant for us. Some notable observations:

- The Micro-F1 score at very low labelling rate is approx 0.08 (at 1 its is 0.48 and 0.44) at below the base node2vec at worst, however the training size as we can see from the first figure has halved, which shows that even for lower training sizes our algorithm performs comparably well to the baseline of node2vec.
- Also, the classification rate overtakes the baseline at various labelling rates which shows a comparable performance at a lower training size.
- Hence, at a loss of of 0.08 micro-F1 score we have a tradeoff with using a far lesser training size.

4.3 BlogCatalog (Tang and Liu) [TL09]

It is a network of of social relationships between users on the blog catalog website. The label represents the categories.

- Vertices: 10,312
- Edges : 333,983
- Classes : 39

Here, since we use an MST random walk, we use a high branching factor so that vertex maps do not overrun dictionary memory, hence using the algorithm variant of not using the base graph leads to very few training examples, e.g. using 80 nodes to predict 5000 samples even with a labelling rate of 70%. Hence, we use the entire pyramid structure as a graph for classification we observe that the loss of accuracy is more acute here in the order of 5%.

Given the running of baselines on BlogCatalog dataset.

And performance of our algorithm with respect to the micro F1 score.

(1).png

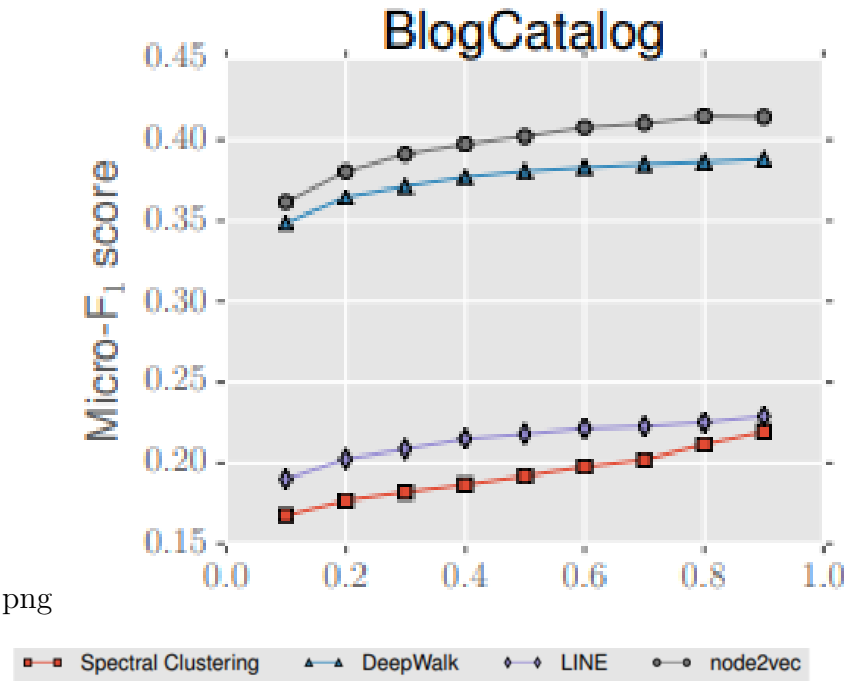


Fig. 4.7 (e) Classification rates on BlogCatalog

4.3.1 Observation:

The loss of accuracy can be attributed to the higher branching factor we've used which results in a smaller pyramid more specifically with the number of supernodes in the upper layer decreasing by a very large factor, e.g. the number of nodes in the upper layer amounts to around 80. Hence, the supernodes aren't as local as we'd like them to be. Therefore, local information is being lost heavily.

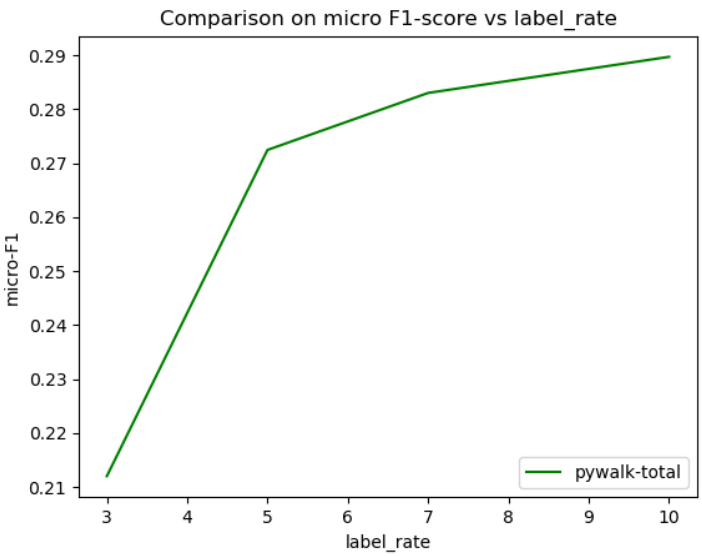


Fig. 4.8 (e) Classification rates on BlogCatalog

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Pywalk essentially condenses the input graph data from local neighbourhoods, thus trying to preserve local information while pooling global information. This was seen in the case of CiteSeer where a branching factor of 2 preserved locality for supernodes and thus lead to consistent results while reducing heavily on the learning model training data.

5.2 Future Work

- The coalesce algorithm uses MST-walk from which it take nodes of the same label, we could extend it to a problem setting scenario where learning occurs on how to coalesce the graph as well.

- We observe that coalescing of nodes into supernodes, collects global information which the base layer does not since random walks encode local information only effectively. Hence, generating an embedding for the entire graph could be an extension of our algorithm.

  We have worked on this, and if we solve the problem that random walk approaches have random initialization and hence the embedding as we coalesce might belong to

different vector spaces. A solution to this problem would be a good direction for the algorithm.

- Also the expressive power of these graphs algorithms can be improved as given by a theory construct presented by Jure Leskovec ([XHLJ18]), implementing the construct could be direction of working.

- **Temporal graphs** : Generating embeddings for graphs with time varying edges is a challenge because the meaning of similarity is now accompanied with the question of frame of reference i.e wrt space or time ?

References

[CPHS18] Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. Harp: Hierarchical representation learning for networks. 2018.

[GL16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[KW16] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[PAS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[SNB+08] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[TL09] Lei Tang and Huan Liu. Relational learning via latent social dimensions. pages 817–826, 2009.

[XHLJ18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.