

2023/08/01

DSAIL summer-internship

STUDENT

유혜원

Hyewon Ryu

# Inductive Representation Learning on Large Graphs

Table of  
Contents

Page

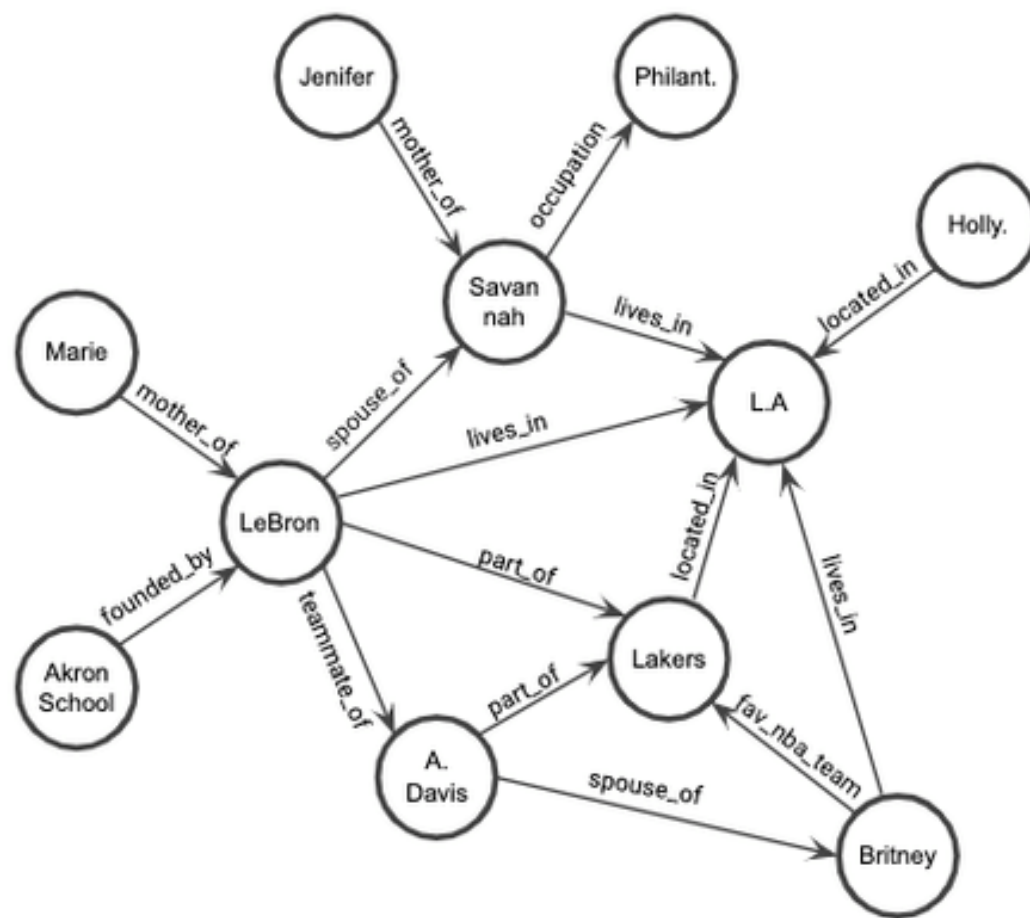
I	Research Background & Motivation	3
II	Related Works	5
III	Model Description	6
IV	Experiment Results	11
V	Implementation	13

# I Research Background & Motivation

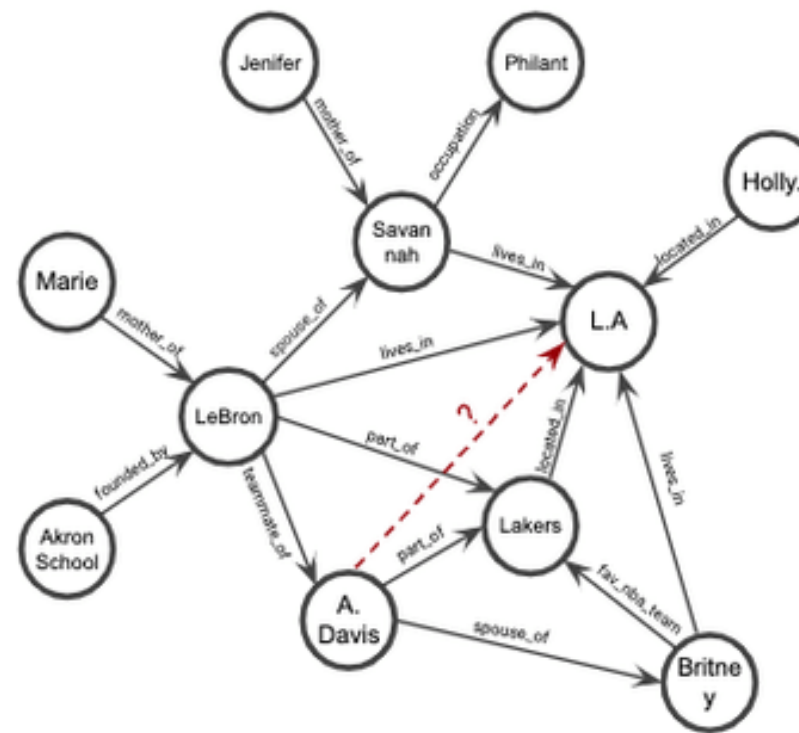
- Low-dimensional vector embedding of nodes in large graphs have proved extremely useful as feature inputs for a wide variety of prediction and graph analysis tasks
  - Node Classification, Clustering, Link Prediction
- Previous works have focused on embedding nodes from a single fixed graph
  - However, many real-world applications require quick embedding for unseen nodes, or entirely new (sub)graphs
  - We need a system which can be used on evolving graphs
  - ex) posts on Reddit, users and videos on Youtube

# I Research Background & Motivation

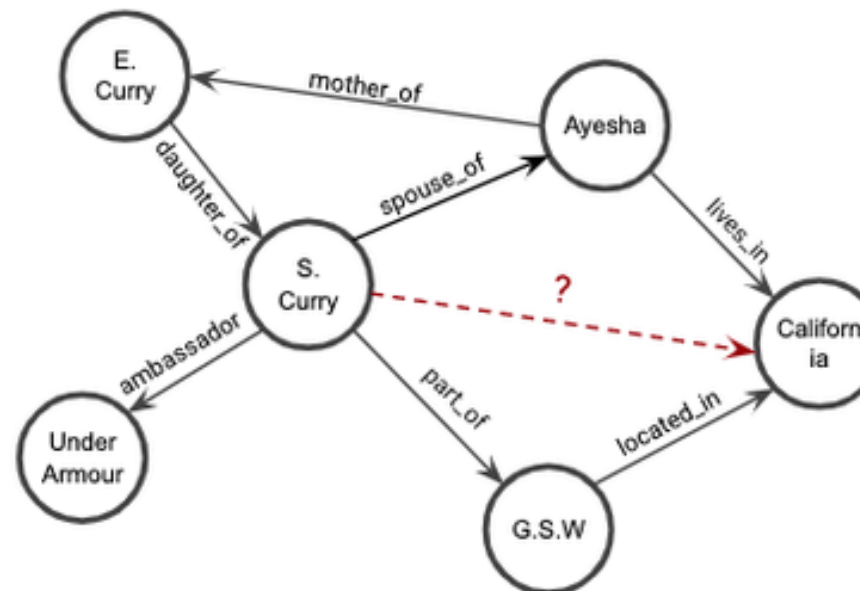
## Transductive vs. Inductive



a. Training graph



b. Transductive inference



c. Inductive inference

Most existing approaches are inherently transductive.

- Matrix-factorization-based
  - do not generalize to unseen data
- Graph Convolution Networks (GCNs) have only been applied in the transductive setting

Inductive node embedding is difficult

- Unseen nodes requires “aligning” newly observed subgraphs
- Need to learn graph structure both in local and global view

In this work, extend GCNs to the task of inductive unsupervised learning

## II Related Works

### 1) Factorization-based embedding approaches

- Embedding using random walk statistics and matrix-factorization-based learning
  - inherently transductive
- Planetoid-I
  - inductive
  - do not use any graph structural information & feature information

### 2) Supervised learning over graphs

- Neural Network approaches to supervised learning over graph structures
  - main focus is to classify entire graphs, not to generate representations for individual nodes

### 3) Graph convolution networks

- do not scale to large graphs or are designed for whole-graph classification
- only applied to transductive setting

### III Model Description

#### Process of Embedding Generation

Assume that the model has already been trained and parameters are fixed

→ Aggregate function parameters & weight matrix  $W$

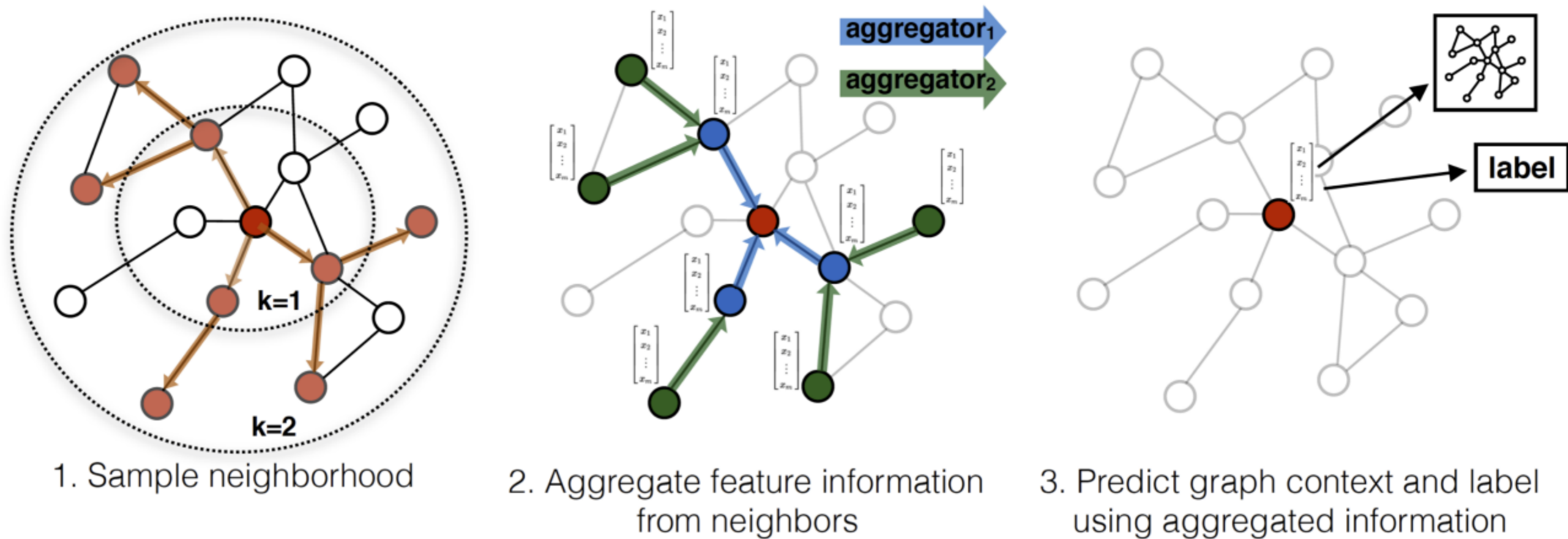
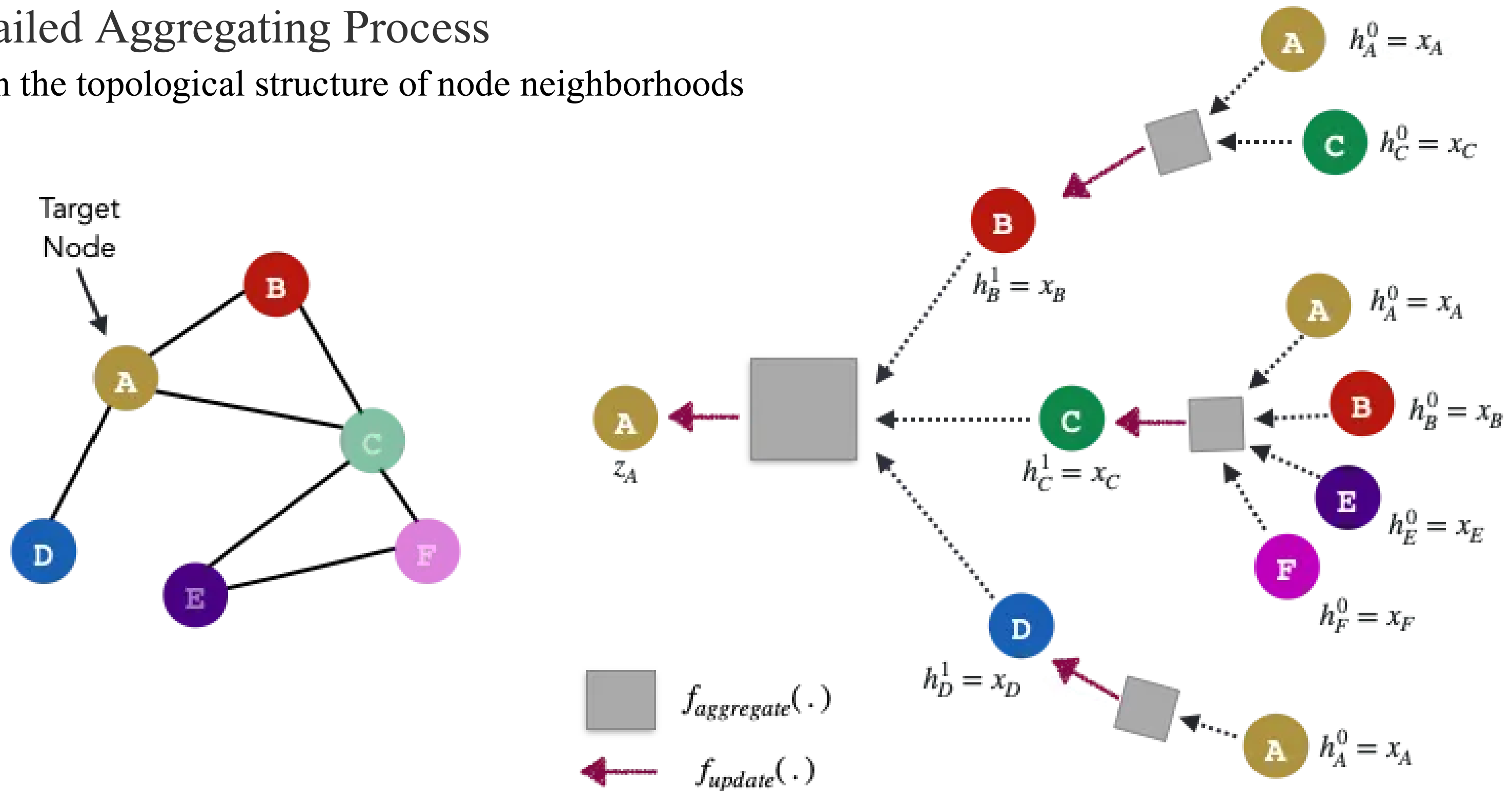


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

### III Model Description

#### Detailed Aggregating Process

Learn the topological structure of node neighborhoods





### III Model Description

#### Pseudo Code of Embedding Generation

---

**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  $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$ ; non-linearity  $\sigma$ ; differentiable aggregator functions  $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$ ; neighborhood function  $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

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

```
1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ; //input node feature
2 for  $k = 1 \dots K$  do
3   for  $v \in \mathcal{V}$  do
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ ;
5      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$  //agreggating
6   end
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
8 end
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$  //final representations output at depth K
```

---



### III Model Description

#### Learning the Parameters of GraphSAGE

Aggregate function parameters & weight matrix  $W$

Definition of neighborhood

$\mathcal{N}(v)$  : fixed-size, uniform draw from the set  $\{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$

Loss function

- Fully unsupervised setting

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

$v$  : node that co-occurs near  $u$  on fixed length random walk

$\sigma$  : sigmoid function

$P_n$  : negative sampling distribution

$Q$  : number of negative samples

- Task-specific objective
  - cross-entropy loss etc

### III Model Description

#### Aggregator Architectures

→ Aggregator functions must operate over an unordered set of vectors

Mean Aggregator: inductive variant of the GCN

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}))$$

→ concatenation : simple form of a “skip connection”  
between the different search depth

#### LSTM Aggregator

- have larger expressive capability
- not symmetric
  - simply applying the LSTMs to a random permutation of the node's neighbors

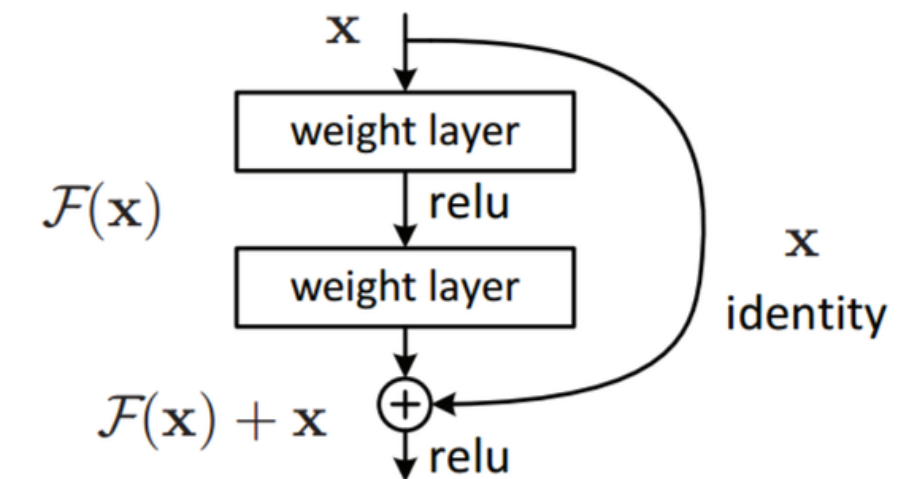
#### Pooling Aggregator

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

- element-wise max pooling

#### Skip Connection

: prevent gradient vanishing (exploding)



## IV Experiment Results

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

Name	Citation		Reddit		PPI	
	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	<b>0.908</b>	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	<b>0.907</b>	<b>0.954</b>	0.482	<b>0.612</b>
GraphSAGE-pool	<b>0.798</b>	<b>0.839</b>	0.892	0.948	<b>0.502</b>	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

## IV Experiment Results

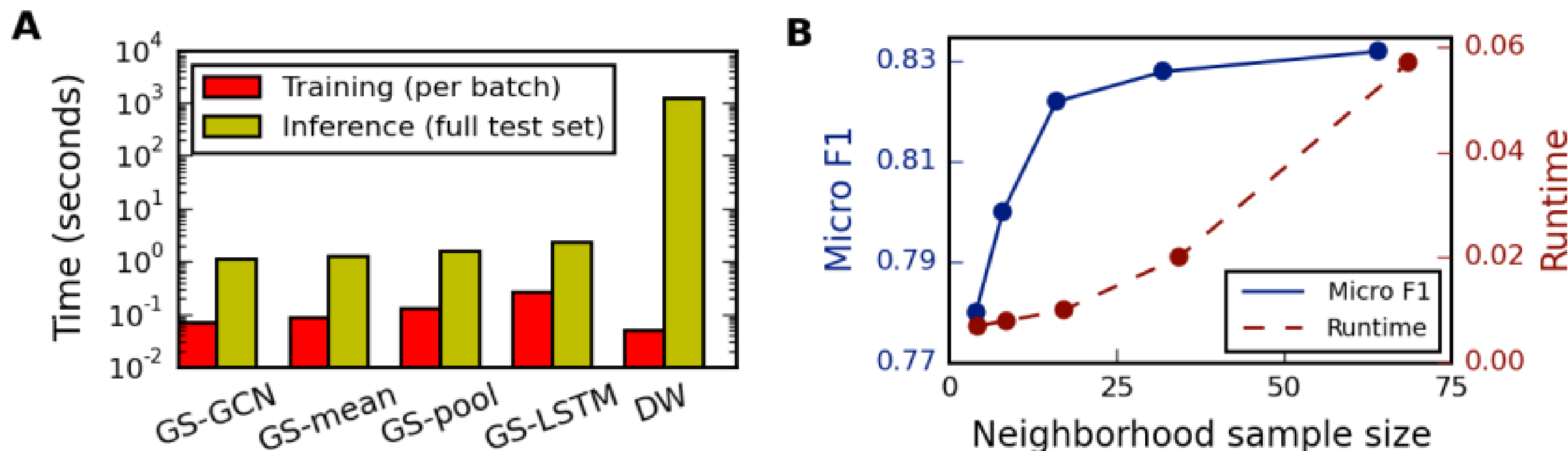


Figure 2: **A:** Timing experiments on Reddit data, with training batches of size 512 and inference on the full test set (79,534 nodes). **B:** Model performance with respect to the size of the sampled neighborhood, where the “neighborhood sample size” refers to the number of neighbors sampled at each depth for  $K = 2$  with  $S_1 = S_2$  (on the citation data using GraphSAGE-mean).

## V Implementation

Dataset: Cora [\[not in paper\]](#)

Node: Papers in machine learning domain

Edge: Citation between papers

→ [node classification into specific domain \(7 classes\)](#)

number of nodes: 2708

number of edges: 10556

train set: 140 nodes

validation set: 500 nodes

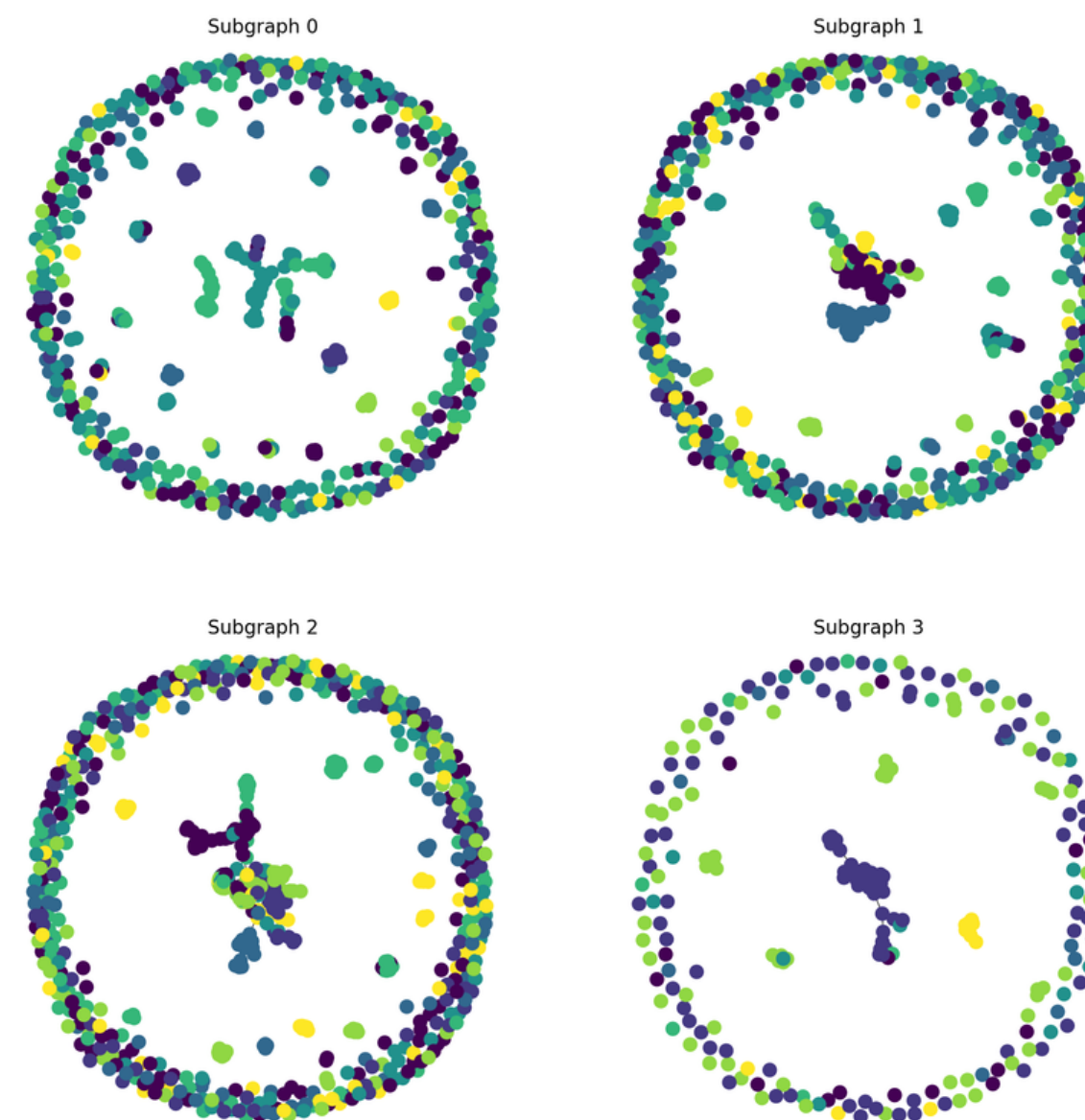
test set: 1000 nodes

```
dataset = Planetoid(root='tmp/Cora', name='Cora')
```

# V Implementation

## Batch creation & Neighborhood sampling

```
1 # Create batches with neighbor sampling
2 train_loader = NeighborLoader(
3     data,
4     num_neighbors = [25, 10], # same as paper
5     batch_size = 40,
6     input_nodes = data.train_mask
7 )
```

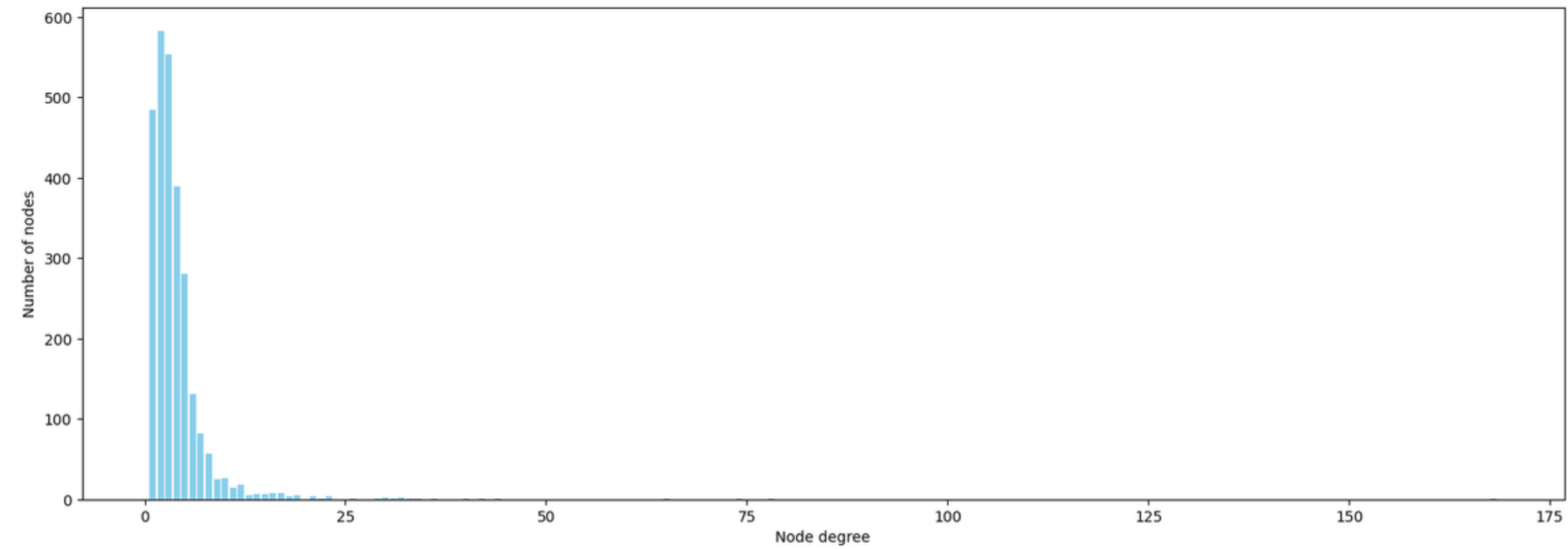


subgraph 0: Data(x=[500, 1433], edge\_index=[2, 817], y=[500], train\_mask=[500], val\_mask=[500], test\_mask=[500], n\_id=[500], e\_id=[817], input\_id=[40], batch\_size=40)  
subgraph 1: Data(x=[563, 1433], edge\_index=[2, 918], y=[563], train\_mask=[563], val\_mask=[563], test\_mask=[563], n\_id=[563], e\_id=[918], input\_id=[40], batch\_size=40)  
subgraph 2: Data(x=[688, 1433], edge\_index=[2, 1324], y=[688], train\_mask=[688], val\_mask=[688], test\_mask=[688], n\_id=[688], e\_id=[1324], input\_id=[40], batch\_size=40)  
subgraph 3: Data(x=[230, 1433], edge\_index=[2, 414], y=[230], train\_mask=[230], val\_mask=[230], test\_mask=[230], n\_id=[230], e\_id=[414], input\_id=[20], batch\_size=20)

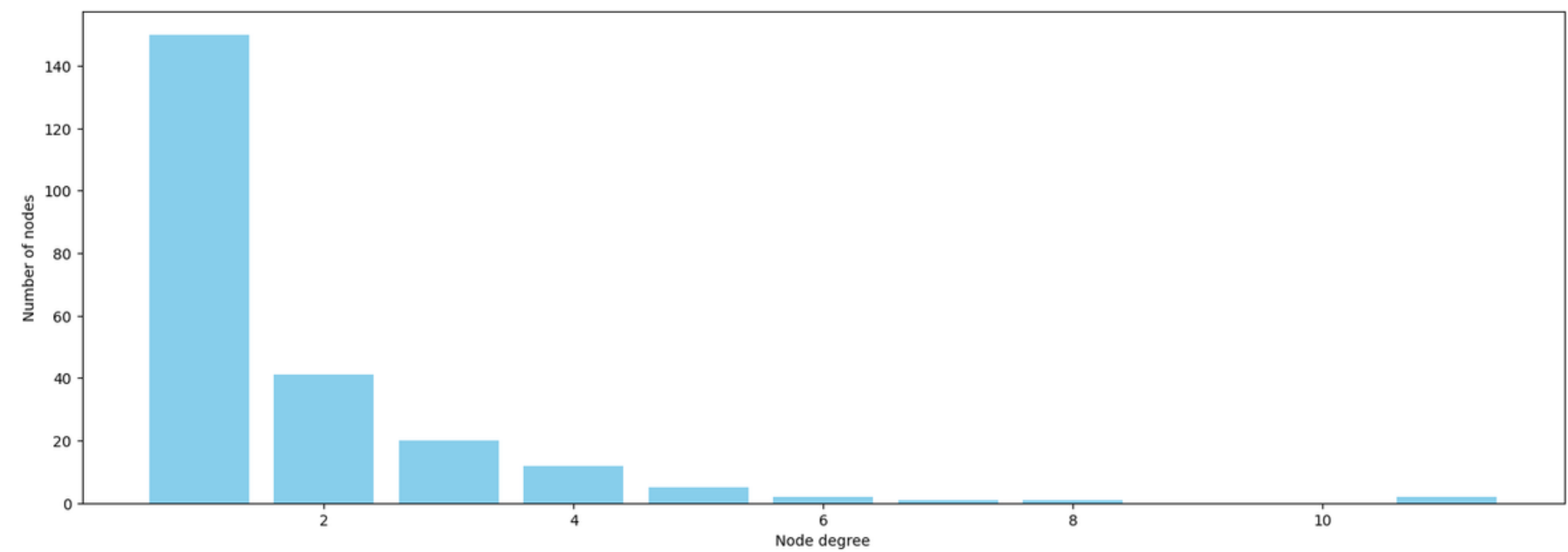


# V Implementation

## Degree



original graph degree



last subgraph degree

## V Implementation

### Accuracy calculation

```
1 def accuracy(pred_y, y):  
2     return((pred_y == y).sum()/len(y)).item()
```

### GraphSAGE

```
1 class GraphSAGE(torch.nn.Module):  
2     def __init__(self, dim_in, dim_h, dim_out, agg = 'mean'):  
3         super().__init__()  
4         self.sage1 = SAGEConv(dim_in, dim_h, aggr=agg)  
5         self.sage2 = SAGEConv(dim_h, dim_out, aggr=agg)  
6         self.optimizer = torch.optim.Adam(self.parameters(),  
7                                           lr=0.01,  
8                                           weight_decay=5e-4)  
9  
10    def forward(self, x, edge_index):  
11        h = self.sage1(x, edge_index)  
12        h = torch.relu(h)  
13        h = F.dropout(h, p=0.5, training=self.training)  
14        h = self.sage2(h, edge_index)  
15        return F.log_softmax(h, dim=1)  
16
```

$$\mathbf{h}'_i = \mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}_i}(\mathbf{h}_j)$$

# V Implementation

## GraphSAGE

### training

```
17 def fit(self, data, epochs):
18     criterion = torch.nn.CrossEntropyLoss()
19     optimizer = self.optimizer
20
21     self.train()
22     for epoch in range(epochs+1):
23         total_loss = 0
24         acc = 0
25         val_loss = 0
26         val_acc = 0
27
28         # train on batches
29         for batch in train_loader:
30             optimizer.zero_grad()
31             out = self(batch.x, batch.edge_index)
32             loss = criterion(out[batch.train_mask], batch.y[batch.train_mask])
33             total_loss += loss.item()
34             acc += accuracy(out[batch.train_mask].argmax(dim=1), batch.y[batch.train_mask])
35             loss.backward()
36             optimizer.step()
37
38         # validation
39         val_loss += criterion(out[batch.val_mask], batch.y[batch.val_mask])
40         val_acc += accuracy(out[batch.val_mask].argmax(dim=1), batch.y[batch.val_mask])
41
42         # Print metrics every 10 epochs
43         if epoch % 5 == 0:
44             print(f'Epoch {epoch:>3} | Train Loss: {loss/len(train_loader):.3f} '
45                   f'| Train Acc: {acc/len(train_loader)*100:>6.2f}%'
46                   f'| Val Loss: {val_loss/len(train_loader):.2f} '
47                   f'| Val Acc: {val_acc/len(train_loader)*100:.2f}%')
48
```

### test

```
49 @torch.no_grad()
50 def test(self, data):
51     test_acc = 0
52     self.eval()
53
54     for batch in train_loader:
55         out = self(batch.x, batch.edge_index)
56         test_acc += accuracy(out[batch.test_mask].argmax(dim=1), batch.y[batch.test_mask])
57
58     print(f'GraphSAGE test accuracy: {test_acc/len(train_loader)*100:.2f}%')
```

### run code

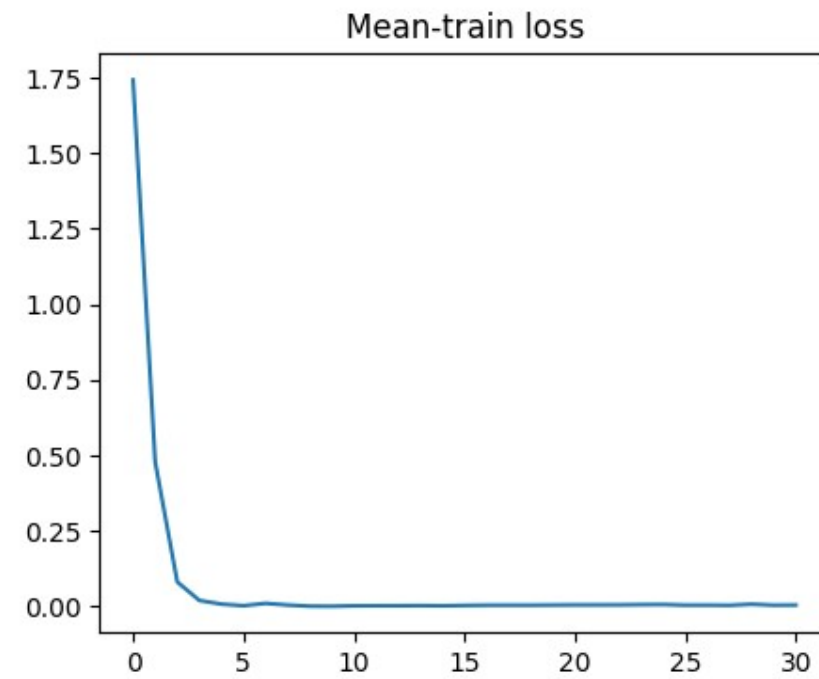
```
1 exp1_max_small = GraphSAGE(dataset.num_features, 64, dataset.num_classes, agg='max')
2 print(exp1_max_small)
3
4 # Train
5 exp1_max_small.fit(data, 30)
6
7 # Test
8 exp1_max_small.test(data)
```

# V Implementation

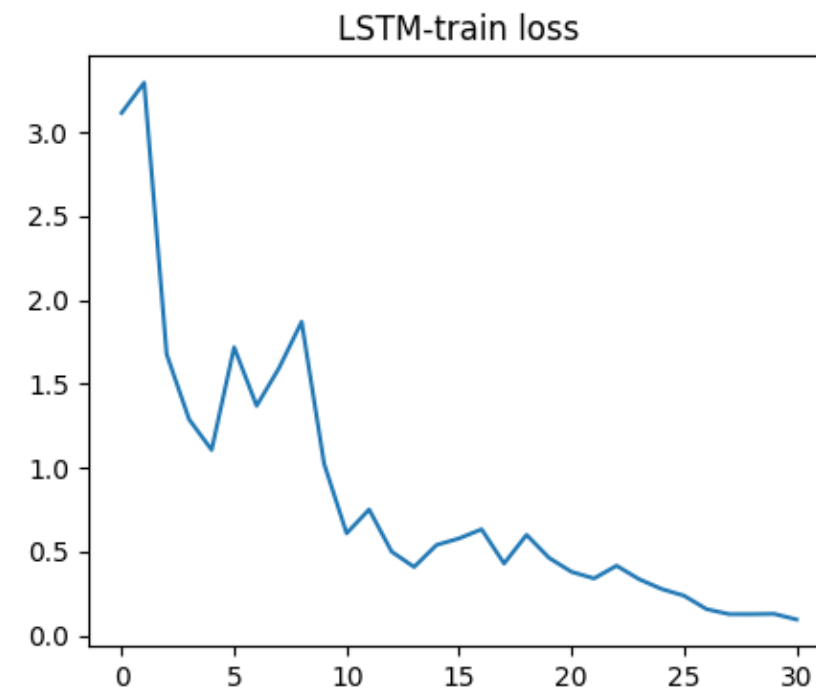
## Result

Train loss

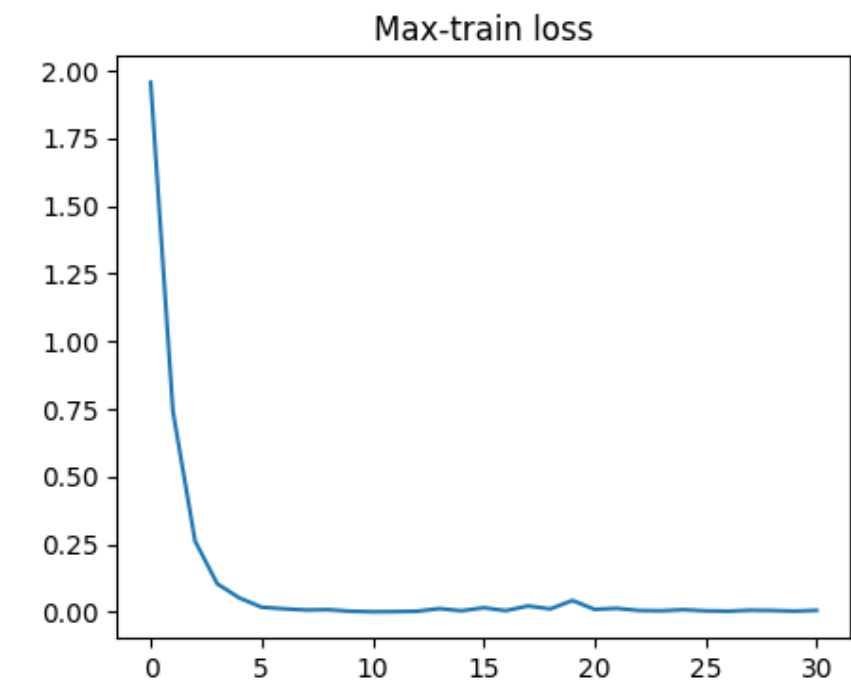
Mean



LSTM

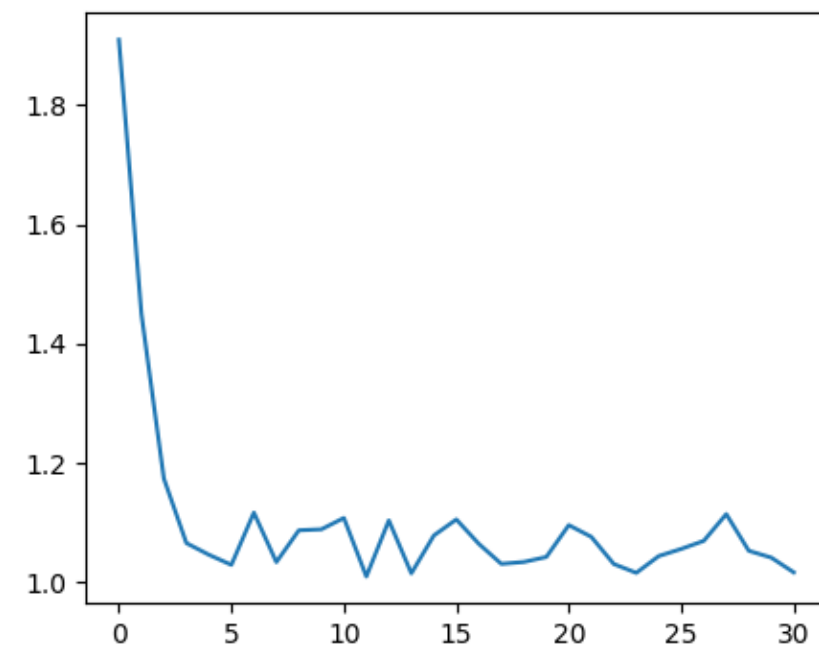


Max

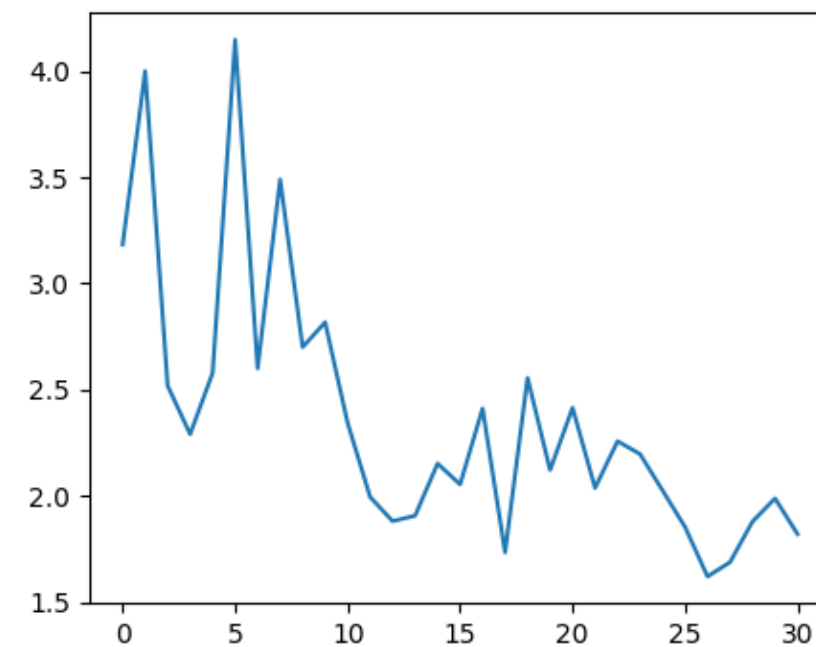


Validation loss

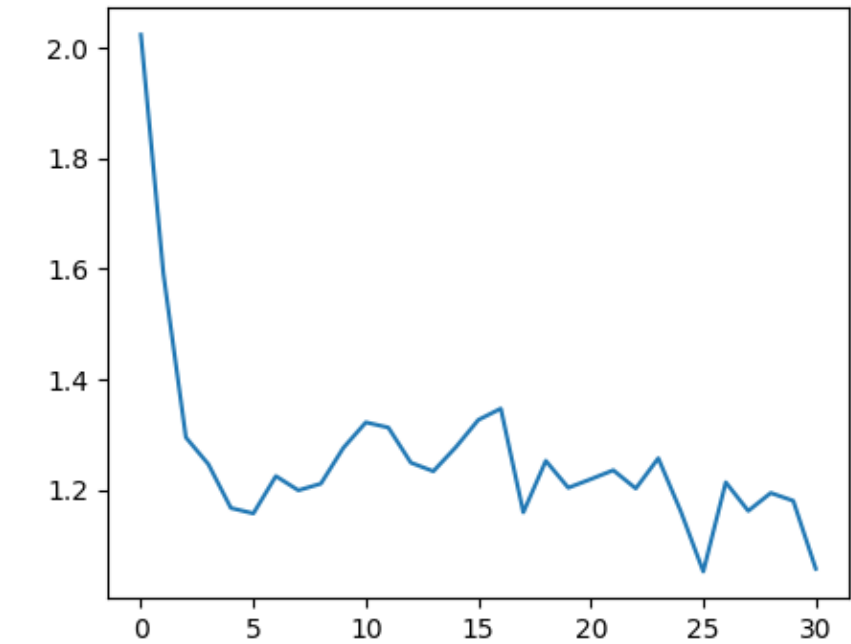
Mean



LSTM



Max



# V Implementation

## Paper Result

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

Name	Citation		Reddit		PPI	
	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	<b>0.908</b>	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	<b>0.907</b>	<b>0.954</b>	0.482	<b>0.612</b>
GraphSAGE-pool	<b>0.798</b>	<b>0.839</b>	0.892	0.948	<b>0.502</b>	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

## Implementaion Result

Mean	
small	big
65.85%	65.86%
LSTM	
small	big
59.27%	64.00%
Max	
small	big
63.44%	62.90%

Training time: Mean < Max <<<<<< LSTM

2023/08/01

DSAIL summer-internship

STUDENT

유혜원

Hyewon Ryu

Thank you  
for listening