2023/08/01

DSAIL summer-internship

STUDENT

유혜원
Hyewon Ryu

Inductive Representation Learning on Large Graphs

Table of Contents

Page

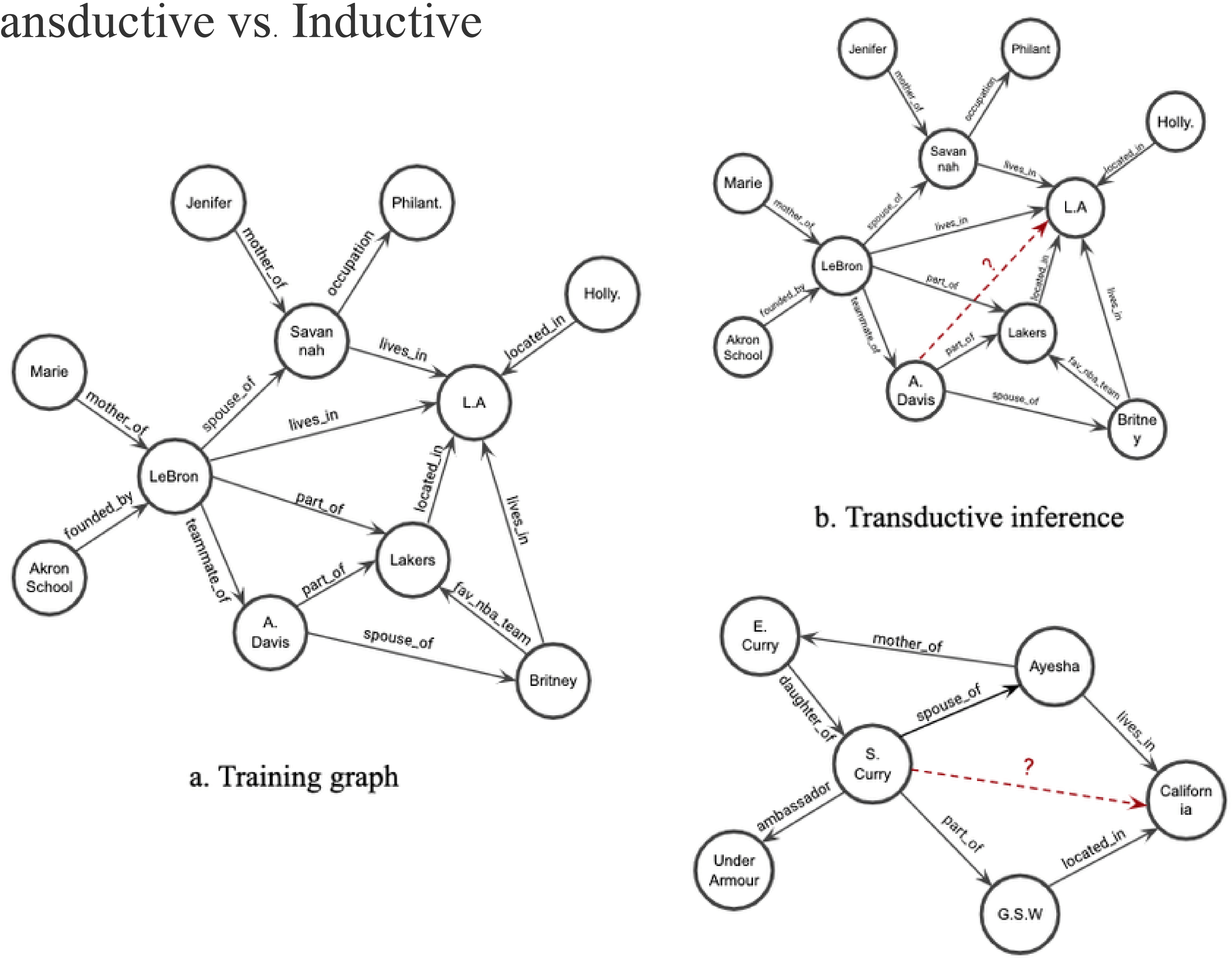
I Research Background & Motivation

- 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 grobal 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 transtuctive
- Planetoid-I
  - inductive
  - do not use any graph structural information & feature information

2) Supervised learning over graphs

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

3) Graph convolution networks

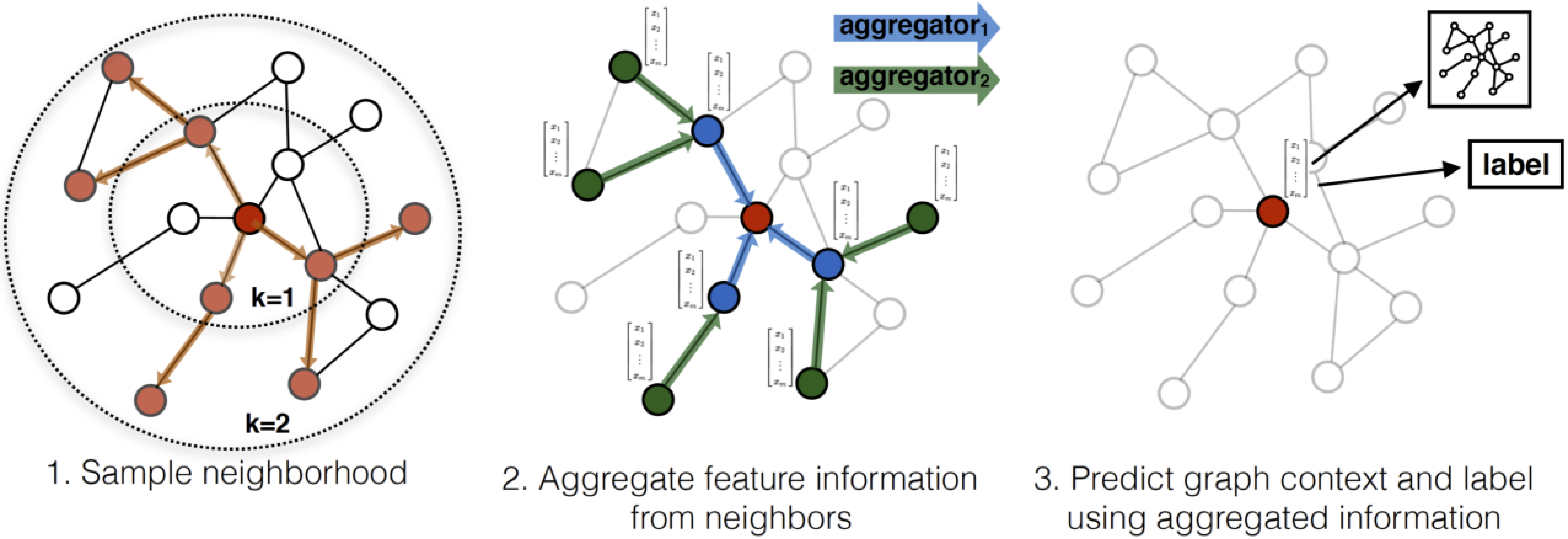
- do not scale to large graphs or are designed for whole-graph classification
- only applied to transductive setting

III Model Description

Process of Embedding Generation

Aggregate function parameters & weight matrix W

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



1. Sample neighborhood

2. Aggregate feature information from neighbors

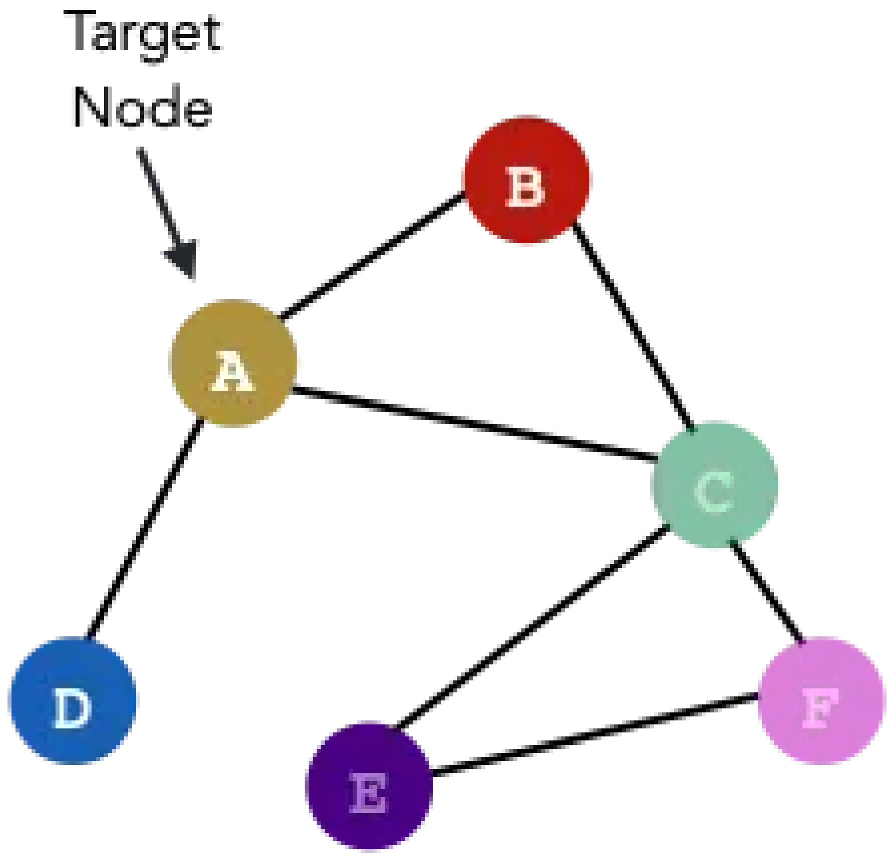
3. Predict graph context and label using aggregated information

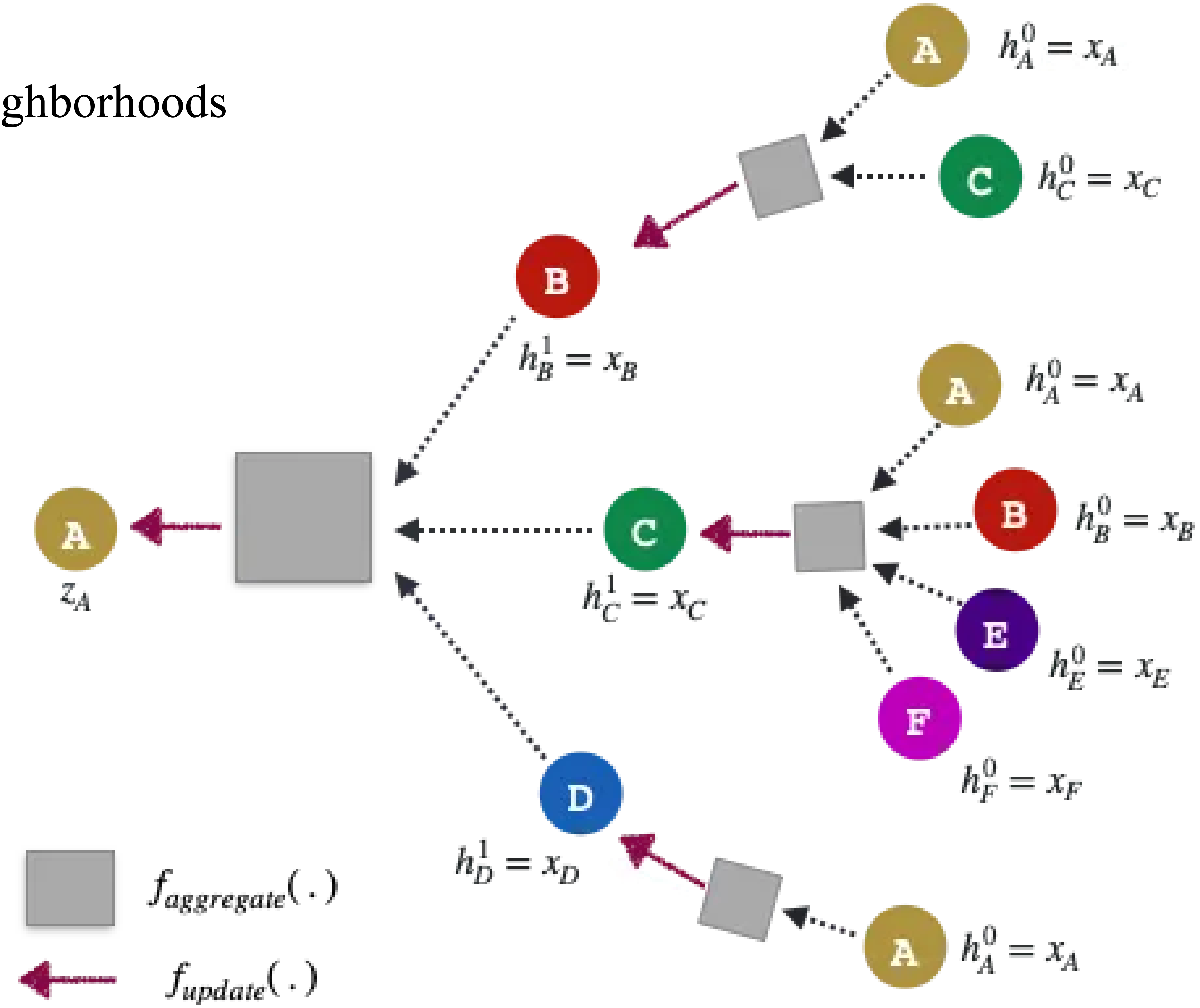
Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

III Model Description

Detailed Aggregating Process

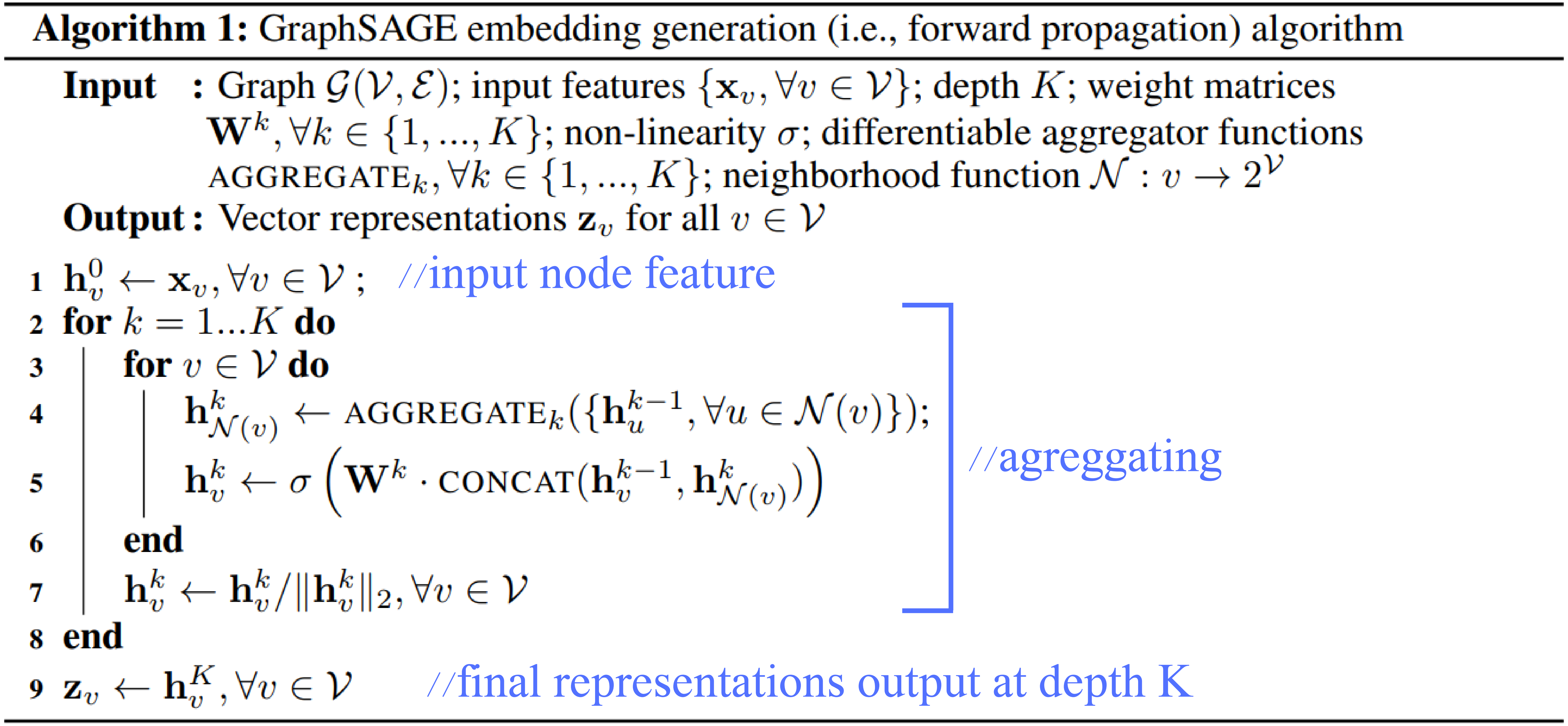
Learn the topological structure of node neighborhoods





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

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

```
$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;    //input node feature
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
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$    //final representations output at depth K
```

//agreggating

III Model Description

Aggregate function parameters & weight matrix W

Learning the Parameters of GraphSAGE

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

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

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

σ : sigmoid function

P_n : negative sampling distribution

Q : number of negative samples

- Task-specific objective
    - cross-entropy loss etc

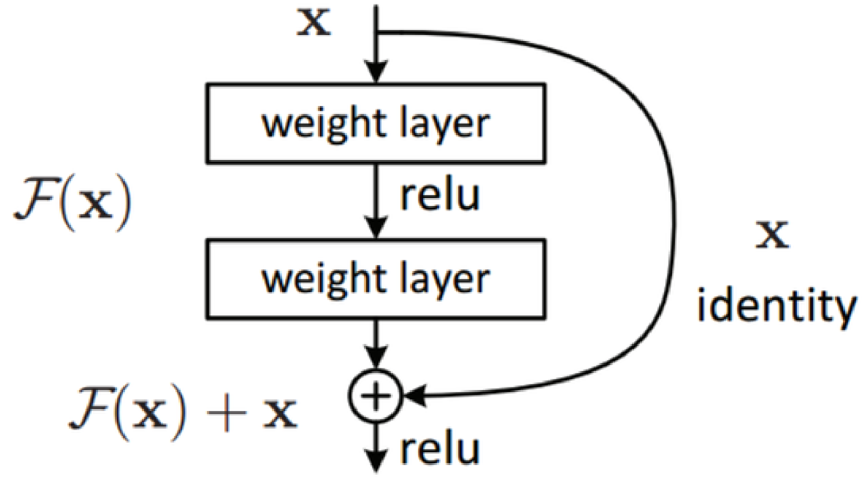
III Model Description

Aggregator Architectures

→ Aggregator functions must operate over an unordered set of vectors

Skip Connection

: prevent gradient vanishing (exploding)


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Mean Aggregator: inductive variant of the GCN

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

→ 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 neighbers

Pooling Aggregator

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

IV Experiment Results

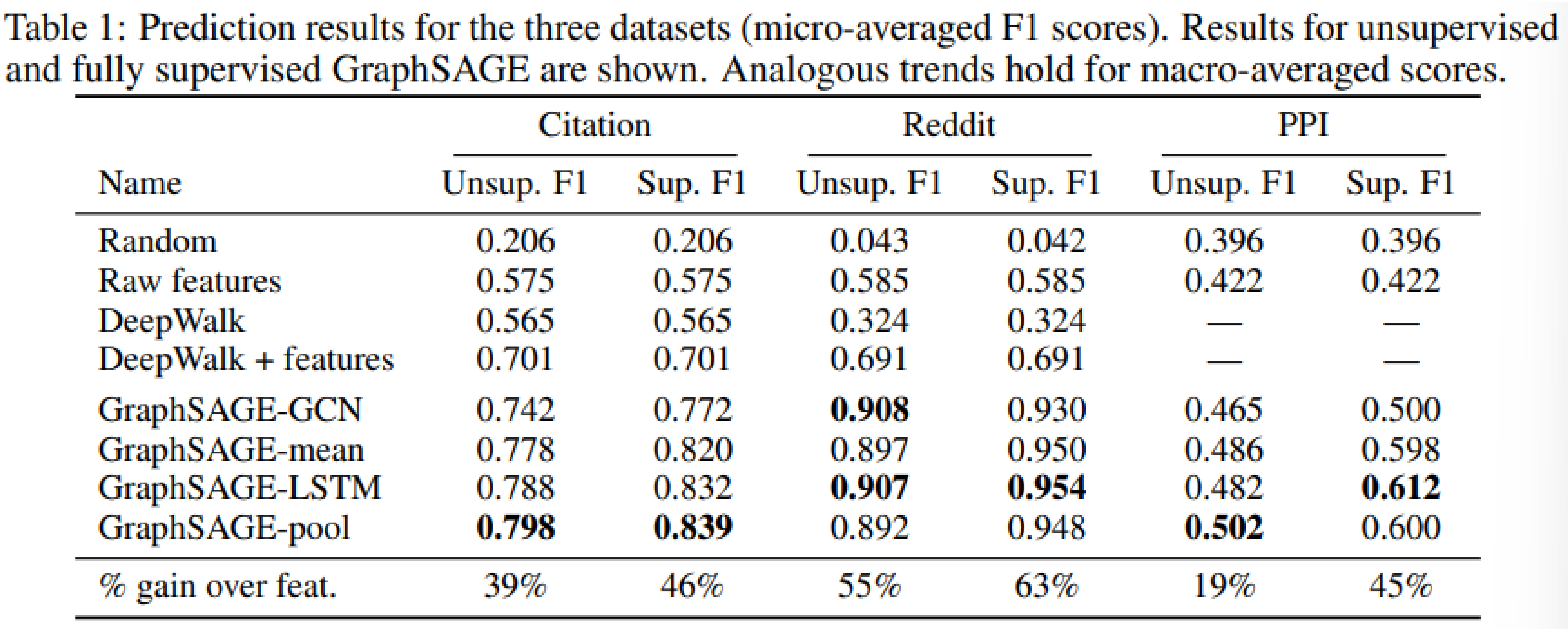


Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	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	**0.908**	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	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

IV Experiment Results

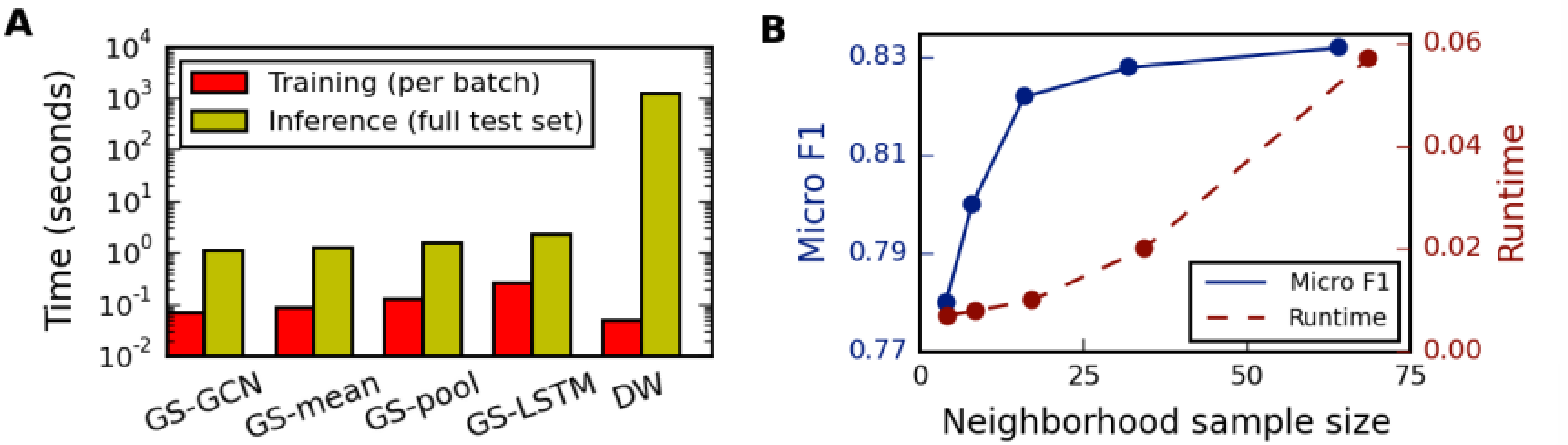


Figure 2: **A**: Timing experiments on Reddit data, with training batches of size 512 and inference on the full test set (79,534 nodes). **B**: Model performance with respect to the size of the sampled neighborhood, where the "neighborhood sample size" refers to the number of neighbors sampled at each 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)

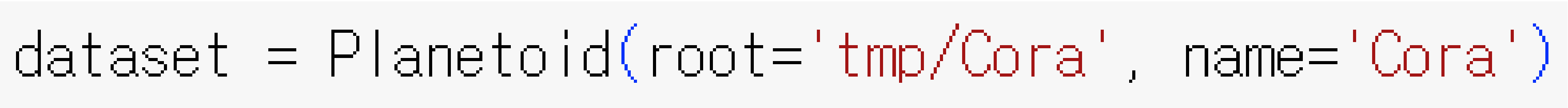
number of nodes: 2708

number of edges: 10556

train set: 140 nodes

validation set: 500 nodes

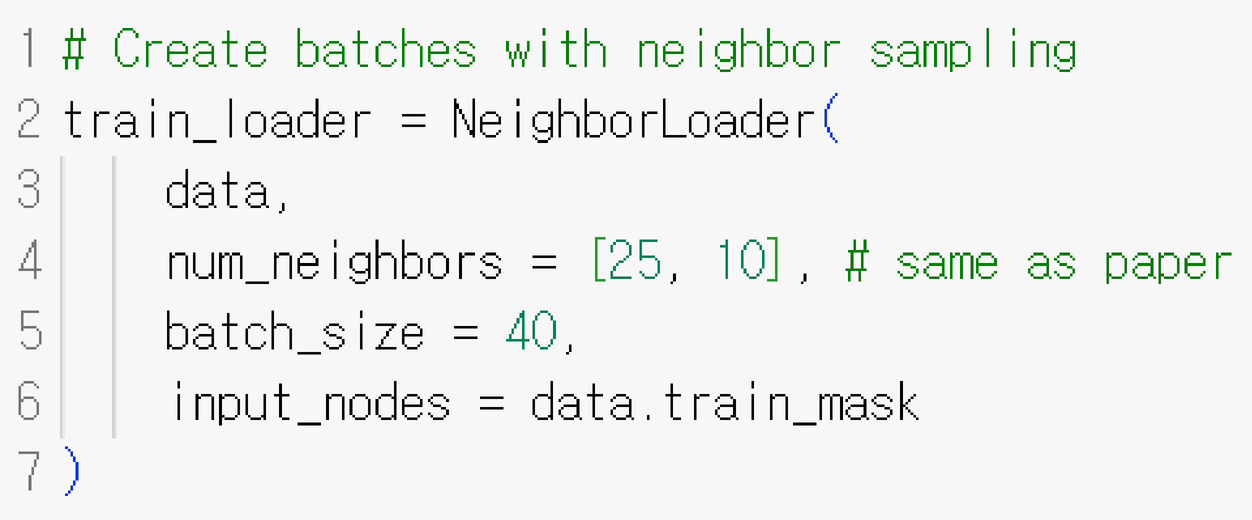
test set: 1000 nodes



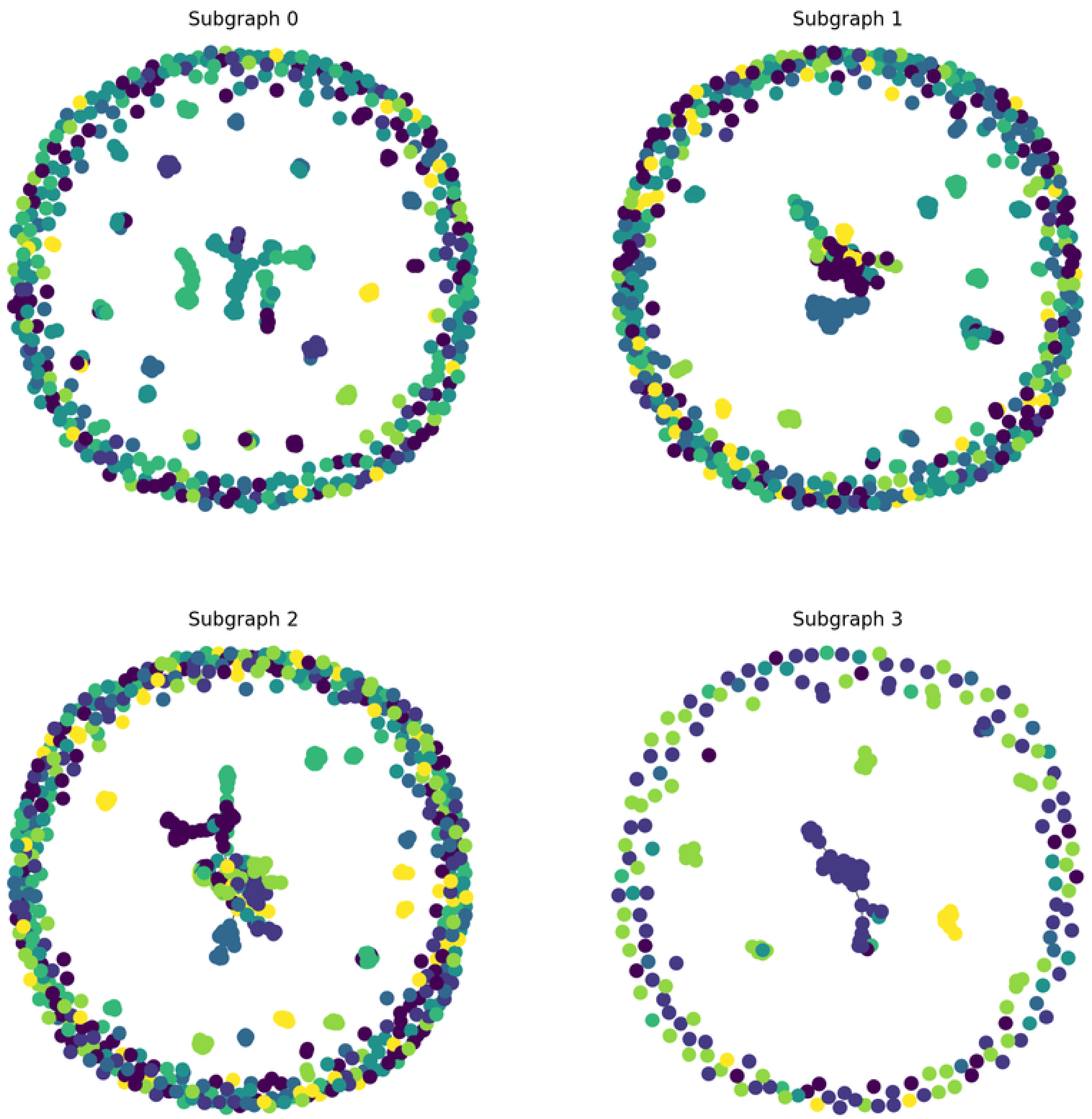
```
dataset = Planetoid(root='tmp/Cora', name='Cora')
```

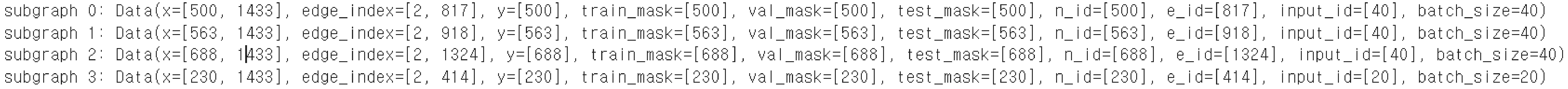
V Implementation

Batch creation & Neighborhood sampling



```
# Create batches with neighbor sampling
train_loader = NeighborLoader(
        data,
        num_neighbors = [25, 10], # same as paper
        batch_size = 40,
        input_nodes = data.train_mask
)
```

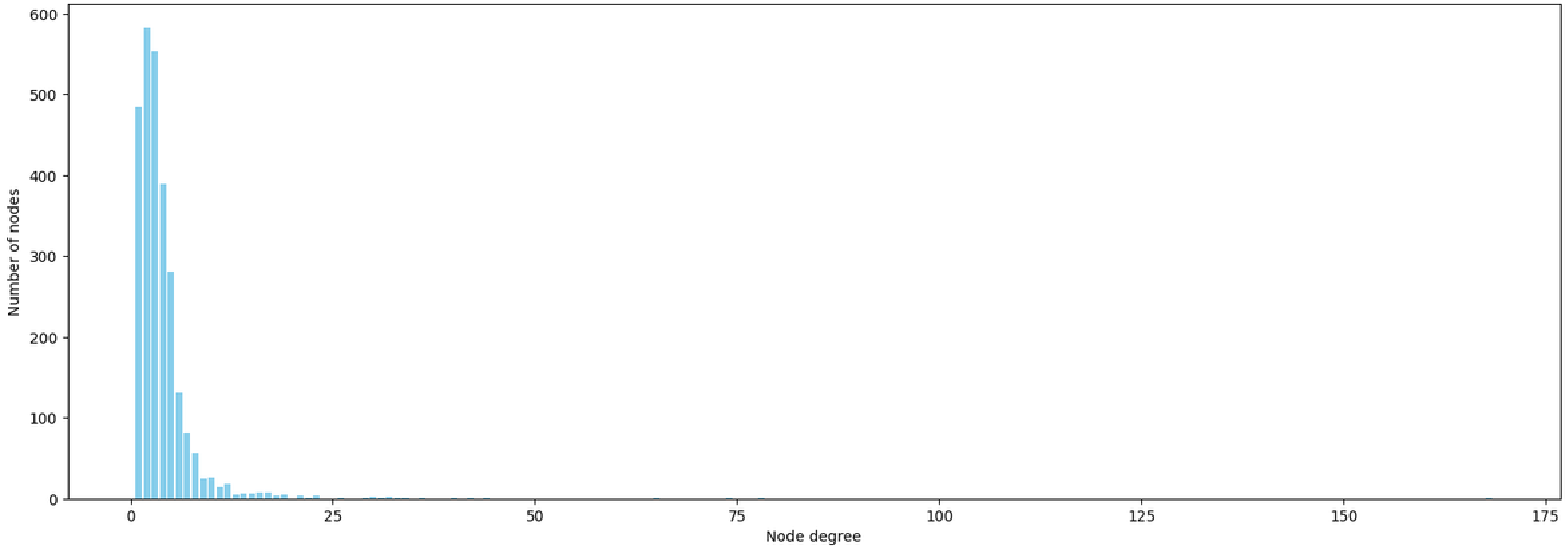




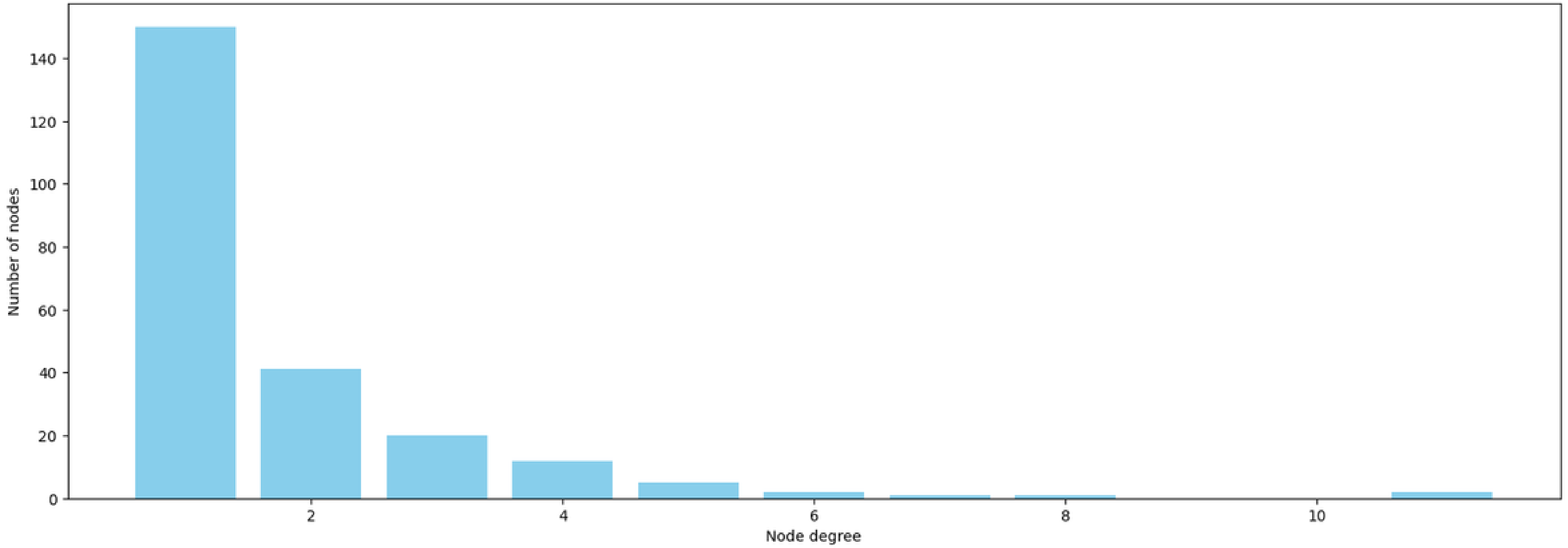
```
subgraph 0: Data(x=[500, 1433], edge_index=[2, 817], y=[500], train_mask=[500], val_mask=[500], test_mask=[500], n_id=[500], e_id=[817], input_id=[40], batch_size=40)
subgraph 1: Data(x=[563, 1433], edge_index=[2, 918], y=[563], train_mask=[563], val_mask=[563], test_mask=[563], n_id=[563], e_id=[918], 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



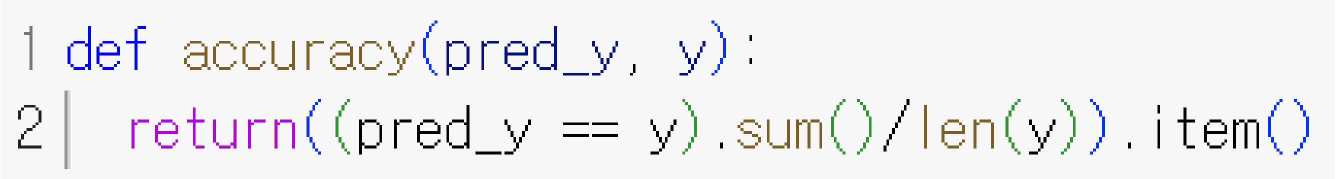
original graph degree



last subgraph degree

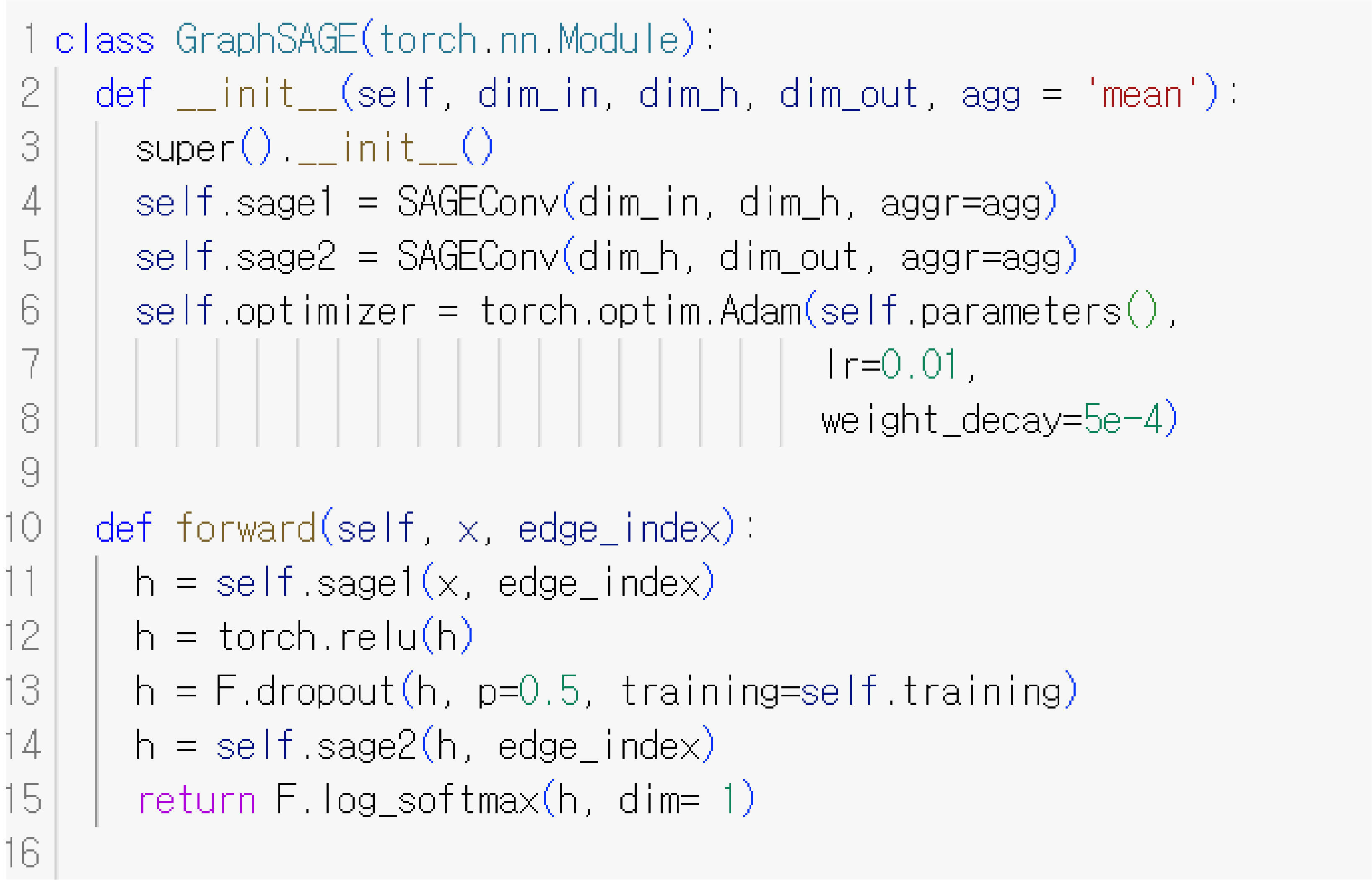
V Implementation

Accuracy calculation



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

GraphSAGE



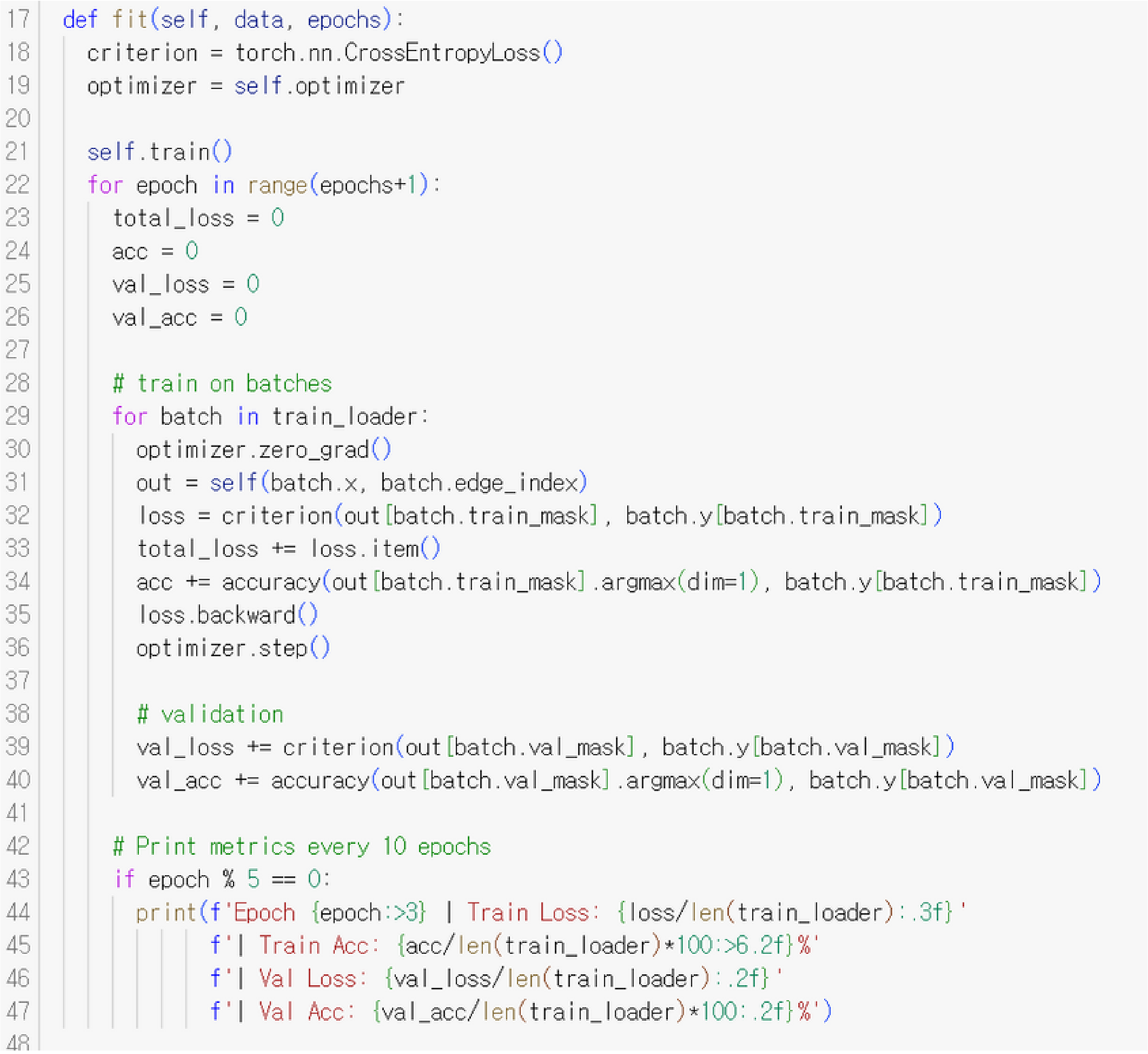
```python
class GraphSAGE(torch.nn.Module):
  def __init__(self, dim_in, dim_h, dim_out, agg = 'mean'):
    super().__init__()
    self.sage1 = SAGEConv(dim_in, dim_h, aggr=agg)
    self.sage2 = SAGEConv(dim_h, dim_out, aggr=agg)
    self.optimizer = torch.optim.Adam(self.parameters(),
                                      lr=0.01,
                                      weight_decay=5e-4)

  def forward(self, x, edge_index):
    h = self.sage1(x, edge_index)
    h = torch.relu(h)
    h = F.dropout(h, p=0.5, training=self.training)
    h = self.sage2(h, edge_index)
    return F.log_softmax(h, dim= 1)

```

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

V Implementation

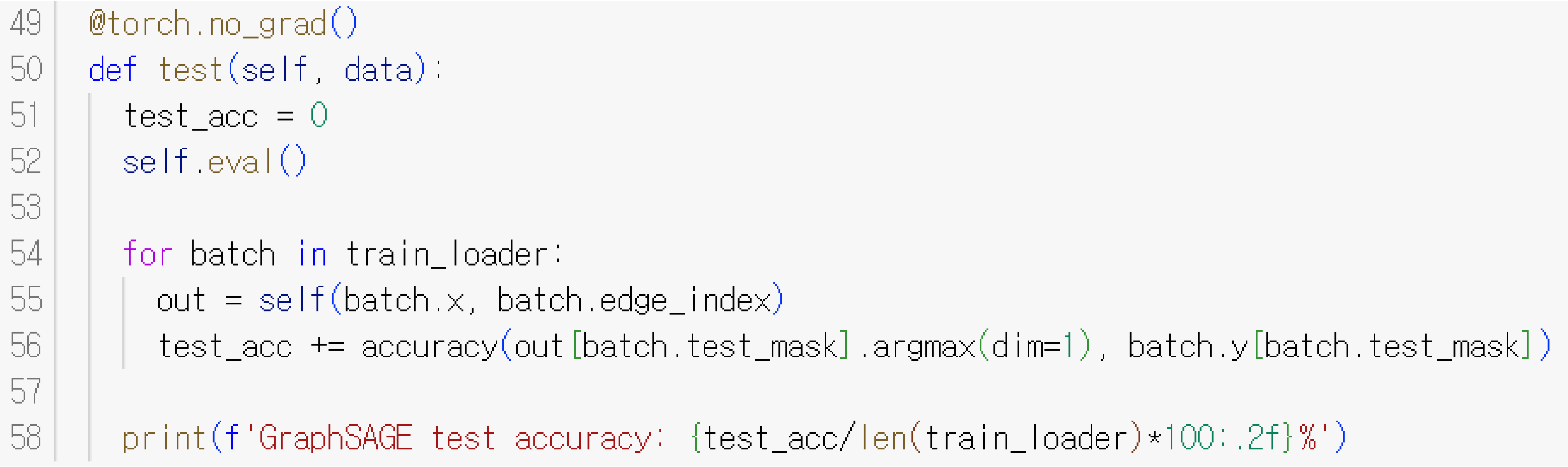
GraphSAGE

training



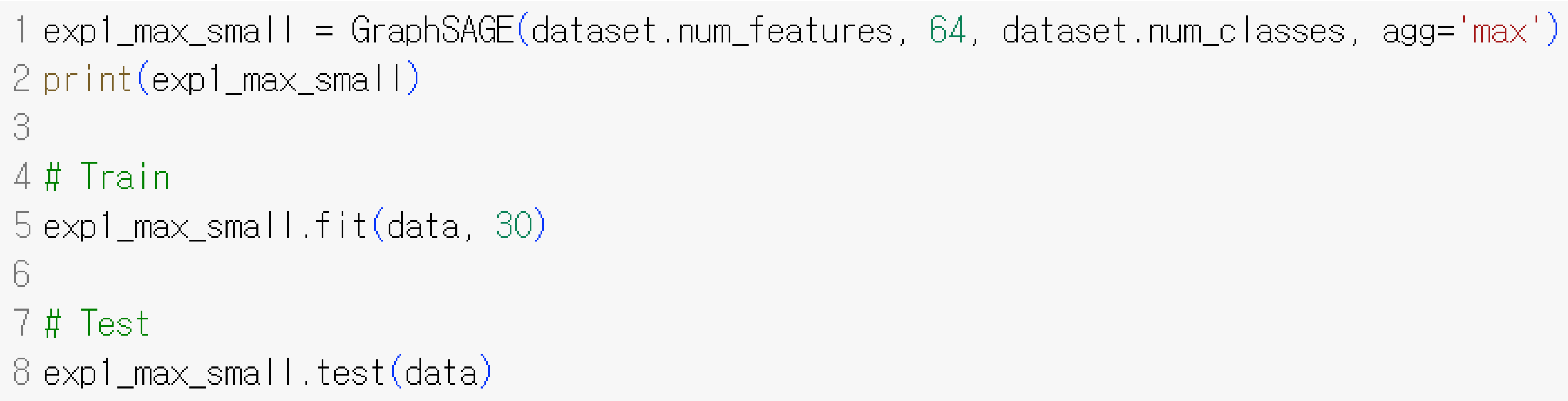
```
def fit(self, data, epochs):
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = self.optimizer

  self.train()
  for epoch in range(epochs+1):
    total_loss = 0
    acc = 0
    val_loss = 0
    val_acc = 0

    # train on batches
    for batch in train_loader:
      optimizer.zero_grad()
      out = self(batch.x, batch.edge_index)
      loss = criterion(out[batch.train_mask], batch.y[batch.train_mask])
      total_loss += loss.item()
      acc += accuracy(out[batch.train_mask].argmax(dim=1), batch.y[batch.train_mask])
      loss.backward()
      optimizer.step()

      # validation
      val_loss += criterion(out[batch.val_mask], batch.y[batch.val_mask])
      val_acc += accuracy(out[batch.val_mask].argmax(dim=1), batch.y[batch.val_mask])

    # Print metrics every 10 epochs
    if epoch % 5 == 0:
      print(f'Epoch {epoch:>3} | Train Loss: {loss/len(train_loader):.3f}'
            f'| Train Acc: {acc/len(train_loader)*100:>6.2f}%'
            f'| Val Loss: {val_loss/len(train_loader):.2f}'
            f'| Val Acc: {val_acc/len(train_loader)*100:.2f}%')

```

test



```
@torch.no_grad()
def test(self, data):
  test_acc = 0
  self.eval()

  for batch in train_loader:
    out = self(batch.x, batch.edge_index)
    test_acc += accuracy(out[batch.test_mask].argmax(dim=1), batch.y[batch.test_mask])

  print(f'GraphSAGE test accuracy: {test_acc/len(train_loader)*100:.2f}%')
```

run code



```
exp1_max_small = GraphSAGE(dataset.num_features, 64, dataset.num_classes, agg='max')
print(exp1_max_small)

# Train
exp1_max_small.fit(data, 30)

# Test
exp1_max_small.test(data)
```

V Implementation

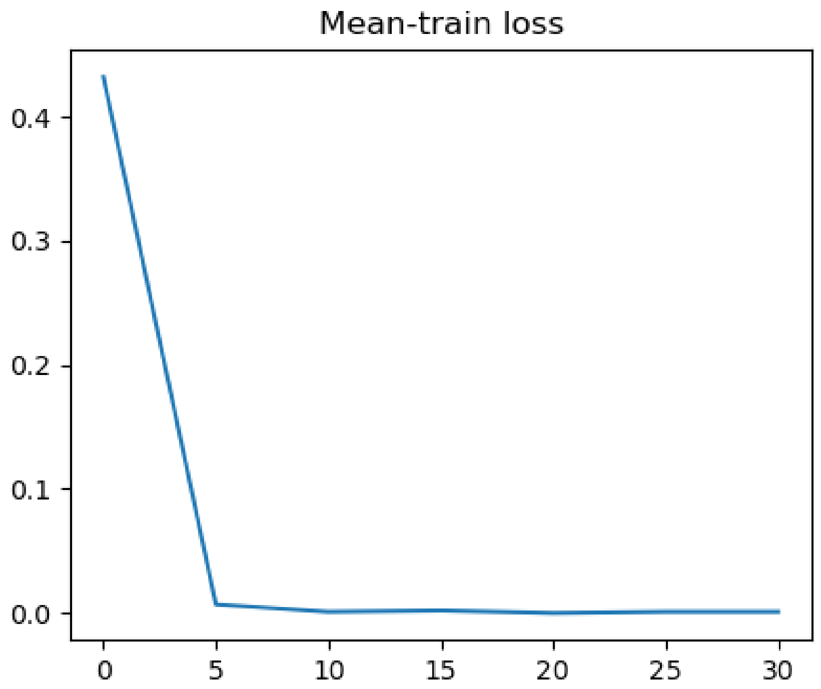
Result

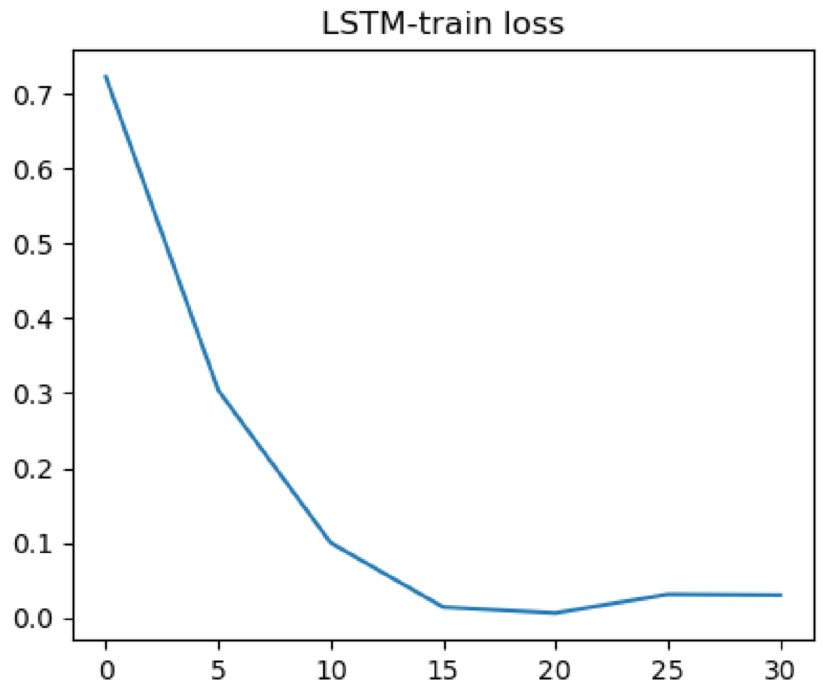
Mean

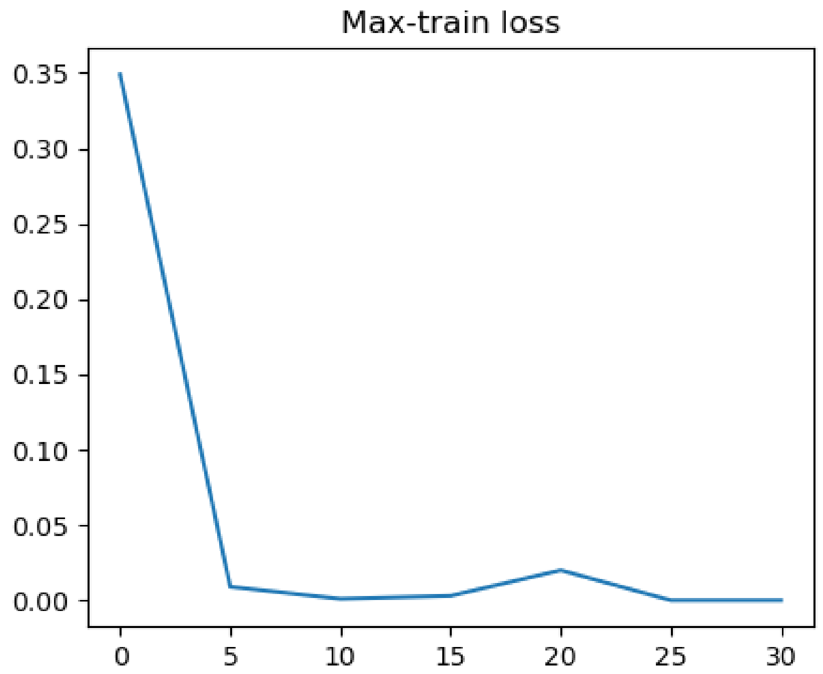
LSTM

Max

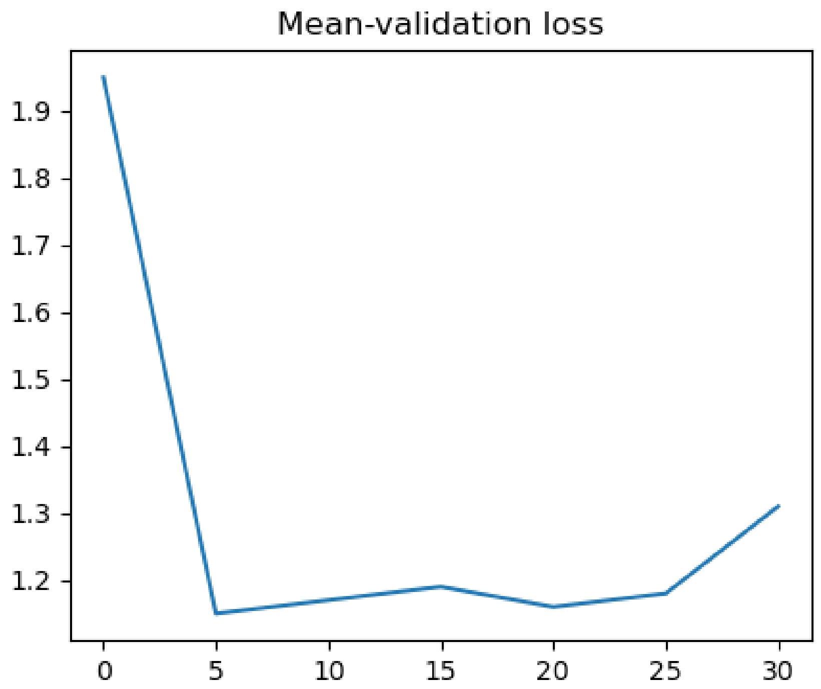
Train loss

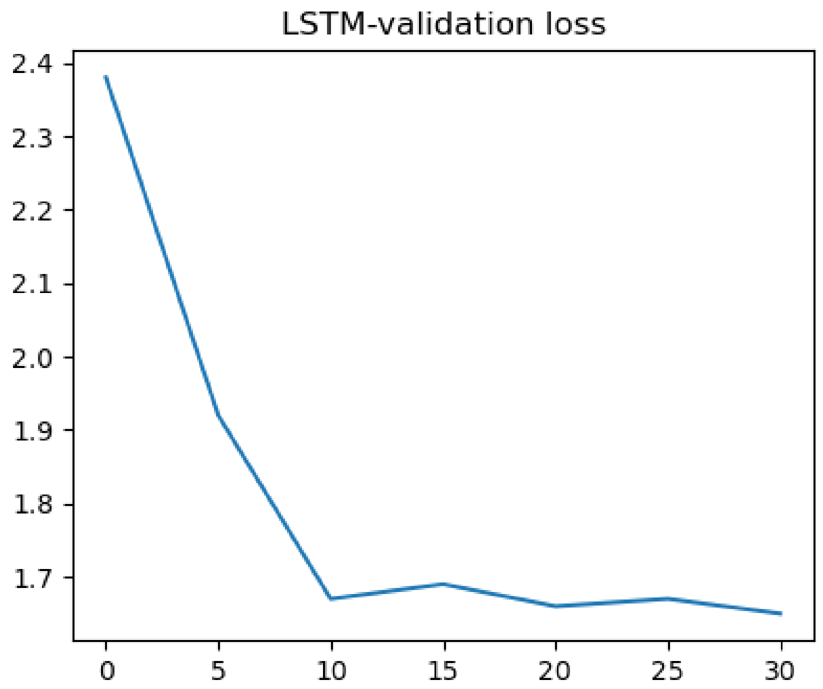


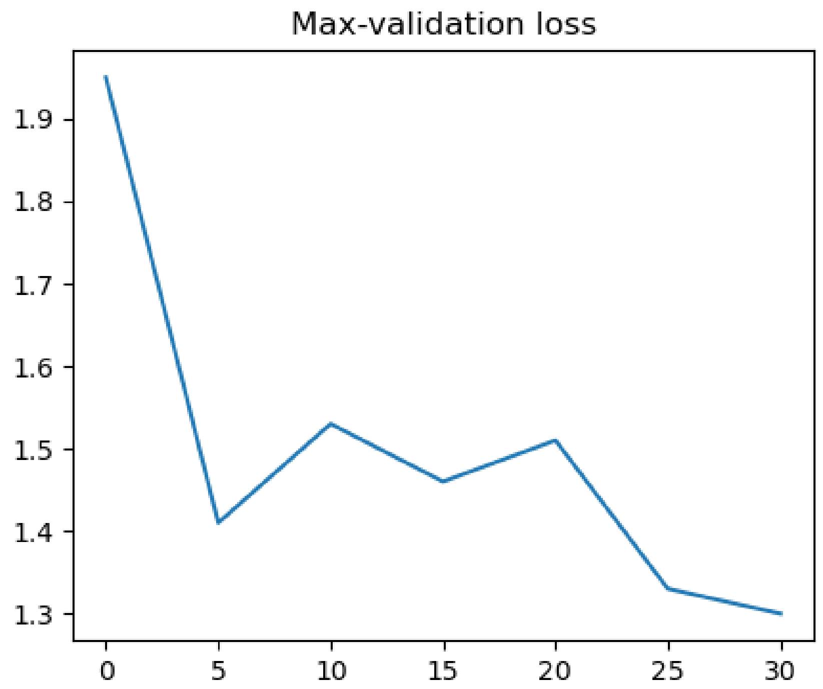




Validation loss







V Implementation

Paper Result

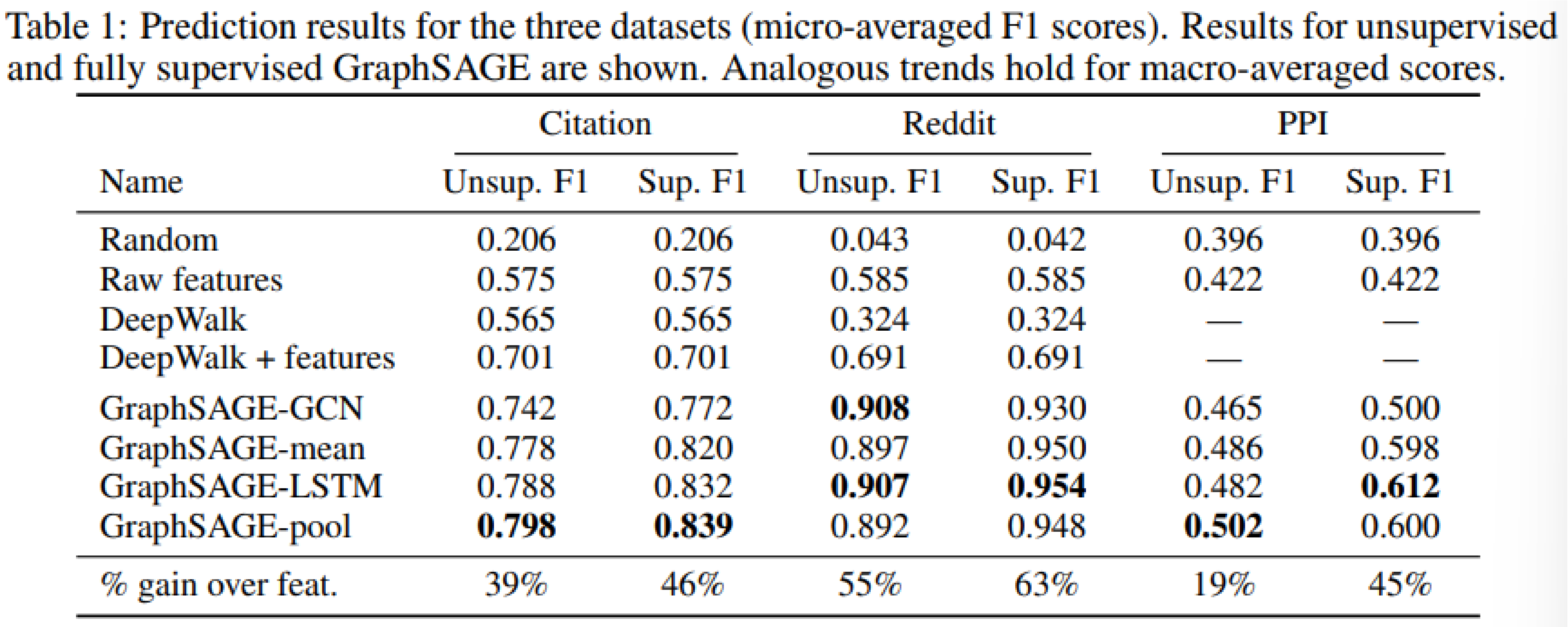


Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	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	**0.908**	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	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	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%

2023/08/01

DSAIL summer-internship

STUDENT

유혜원

Hyewon Ryu

Thank you for listening