GraphSAGE Tutorial

Feb 18, 2021

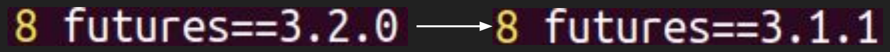
Table of Contents

1. Installation
2. Guides
3. Data pre-processing
4. About the code
    a. Train/ val/ test nodes and edges
    b. Adjacency list
    c. Identity features
    d. Node sampling
    e. Neighbor sampling
    f. Test Stats
5. Execution script

Installation

Steps

1. git clone https://github.com/williamleif/GraphSAGE
2. (optional) create a virtual environment
   a. pip install --user virtualenv
   b. python -m venv [env_name]
3. pip install -r requirements.txt
   a. If using python3:
   b. Open requirements.txt
   c. Replace futures==3.2.0 with futures==3.1.1



`futures==3.2.0` → `futures==3.1.1`

Guides

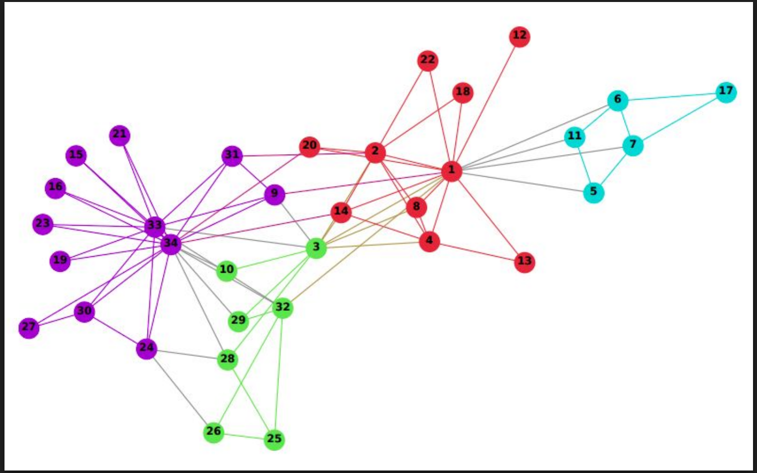
Guides

1. https://github.com/williamleif/GraphSAGE (README)
    a. How to install
    b. How to use
    c. Models description
2. Graphsage/example_data
    a. Example of the required data format
3. Graphsage/example_supervised.sh & Graphsage/example_unsupervised.sh
    a. Example command to run the training and evaluation code in grarphsage/supervised_train.py

Data

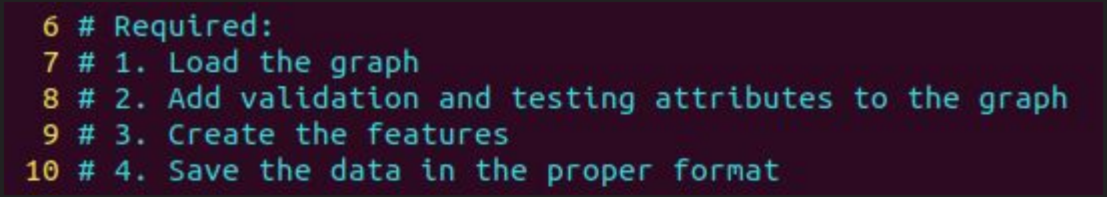
Karate Toy Data

- 34 nodes
- Node classification (4 classes)
- Node attributes (toy-data only):
    - Last name
    - Age
- Neighboring nodes:
    - High chance of same last name (e.g. siblings)
    - High change of similar age (±5)

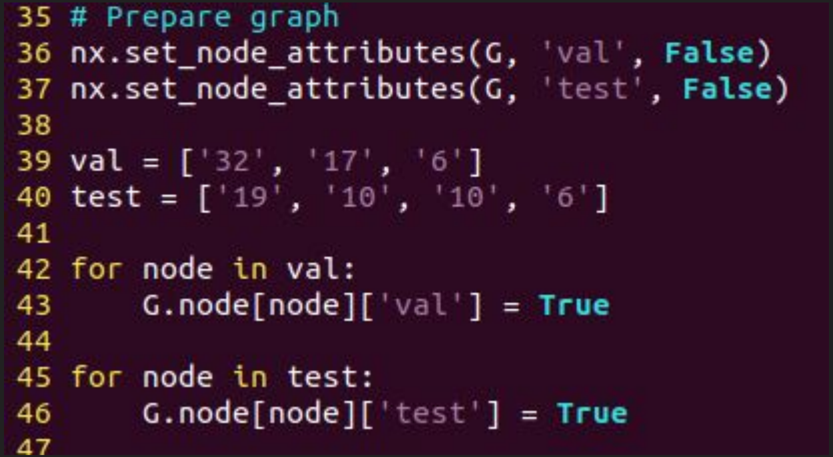


Preparing Data

Files: (data/prepare_data.py)



```
# Required:
# 1. Load the graph
# 2. Add validation and testing attributes to the graph
# 3. Create the features
# 4. Save the data in the proper format
```

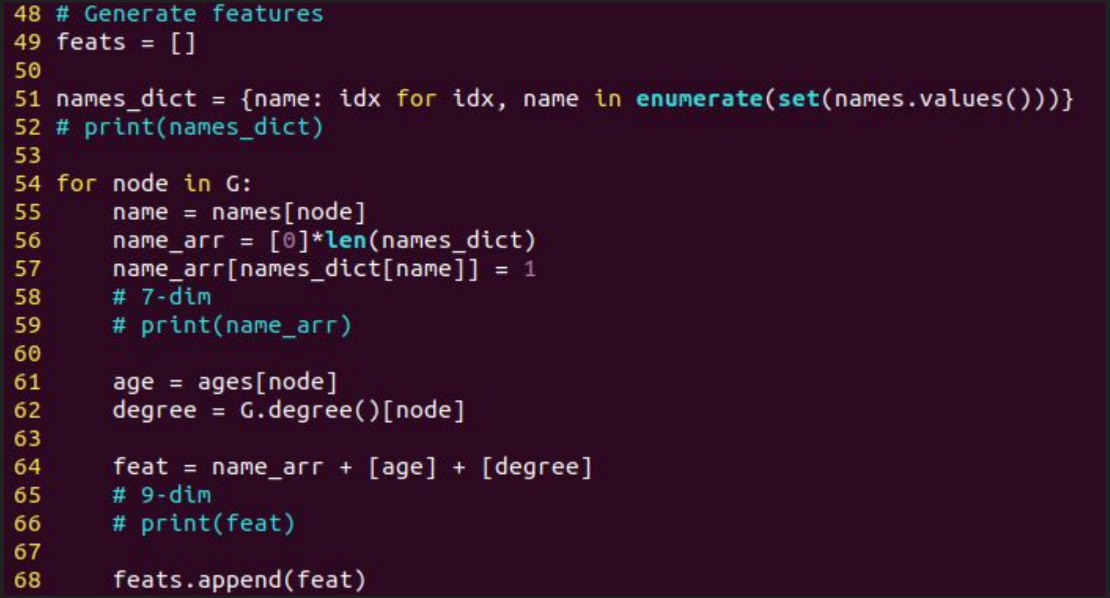


```
# Prepare graph
nx.set_node_attributes(G, 'val', False)
nx.set_node_attributes(G, 'test', False)

val = ['32', '17', '6']
test = ['19', '10', '10', '6']

for node in val:
    G.node[node]['val'] = True

for node in test:
    G.node[node]['test'] = True
```

Generating Features

In this case, using:

1. Name
   a. One-hot vector
2. Age
   a. Integer
3. Node degree
   a. Integer

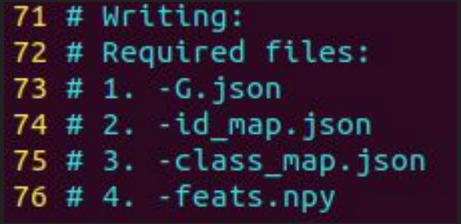


```python
# Generate features
feats = []

names_dict = {name: idx for idx, name in enumerate(set(names.values()))}
# print(names_dict)

for node in G:
    name = names[node]
    name_arr = [0]*len(names_dict)
    name_arr[names_dict[name]] = 1
    # 7-dim
    # print(name_arr)

    age = ages[node]
    degree = G.degree()[node]

    feat = name_arr + [age] + [degree]
    # 9-dim
    # print(feat)

    feats.append(feat)
```

Writing Data

Write the processed data in the required format (json or npy).

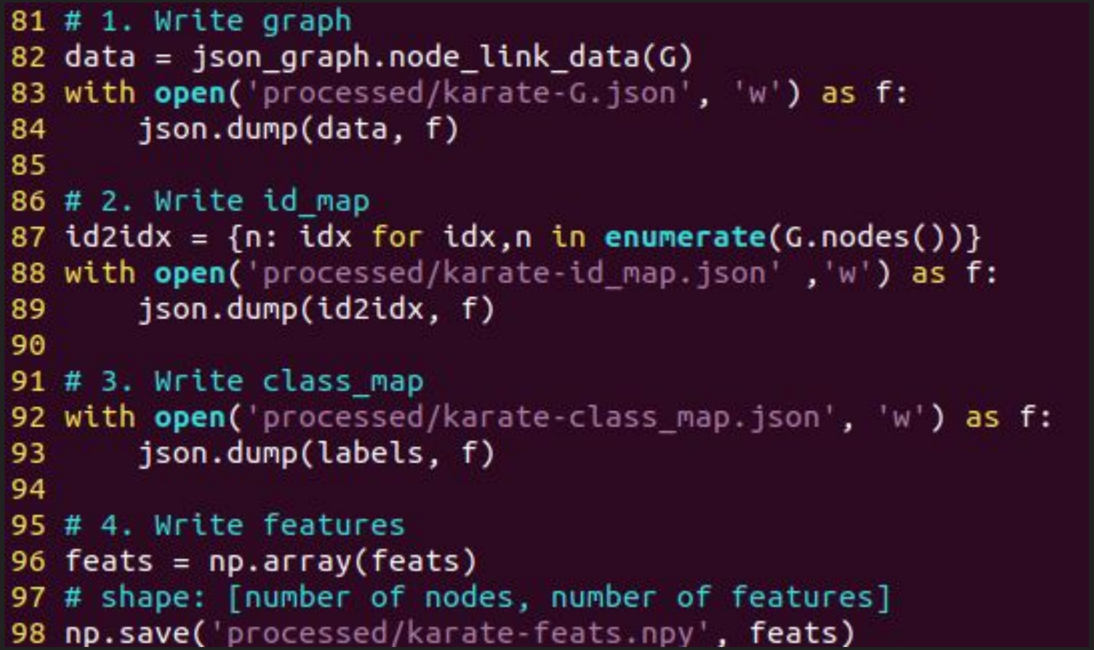
All files should include the same prefix.



```
# Writing:
# Required files:
# 1. -G.json
# 2. -id_map.json
# 3. -class_map.json
# 4. -feats.npy
```

Writing Data

1. Graph written as json
2. ID map as json
   a. It is required because nodes can be indexed by strings instead of integers.
3. Class map as json
4. Features as numpy array



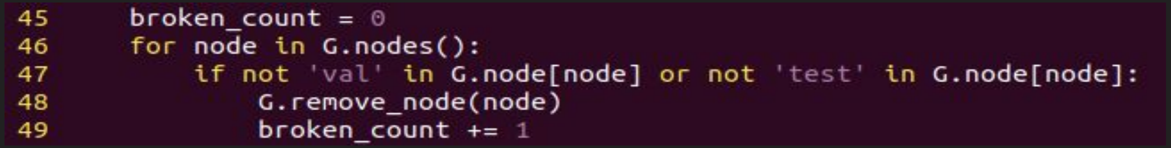
```
# 1. Write graph
data = json_graph.node_link_data(G)
with open('processed/karate-G.json', 'w') as f:
    json.dump(data, f)

# 2. Write id_map
id2idx = {n: idx for idx,n in enumerate(G.nodes())}
with open('processed/karate-id_map.json' ,'w') as f:
    json.dump(id2idx, f)

# 3. Write class_map
with open('processed/karate-class_map.json', 'w') as f:
    json.dump(labels, f)

# 4. Write features
feats = np.array(feats)
# shape: [number of nodes, number of features]
np.save('processed/karate-feats.npy', feats)
```

GraphSAGE Code

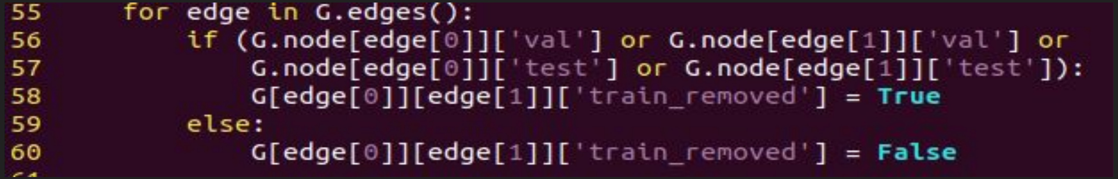
Files: utils.py

Train/ Val/ Test Nodes and Edges

- Graph should have ‘**val’** and ‘**test**’ attribute on every node.
- Edges connected to ‘val’ or ‘test’ nodes are **removed.**



```
    broken_count = 0
    for node in G.nodes():
        if not 'val' in G.node[node] or not 'test' in G.node[node]:
            G.remove_node(node)
            broken_count += 1
```

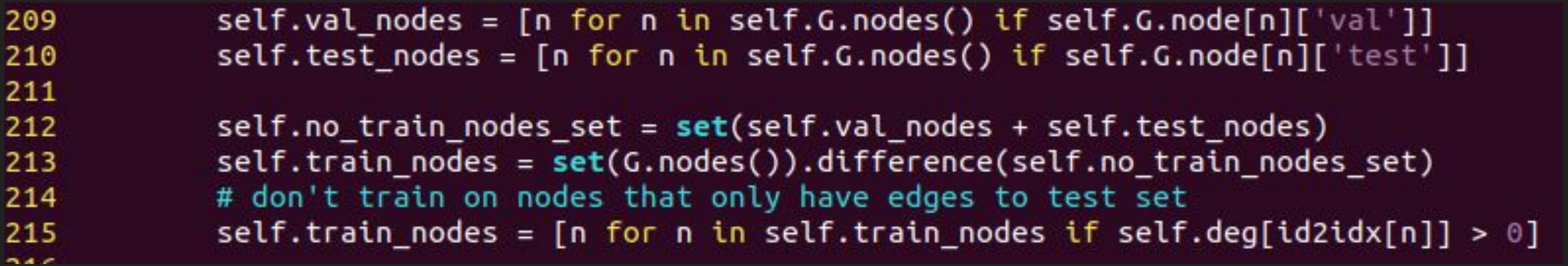


```
    for edge in G.edges():
        if (G.node[edge[0]]['val'] or G.node[edge[1]]['val'] or
            G.node[edge[0]]['test'] or G.node[edge[1]]['test']):
            G[edge[0]][edge[1]]['train_removed'] = True
        else:
            G[edge[0]][edge[1]]['train_removed'] = False
```

Files: minibatch.py

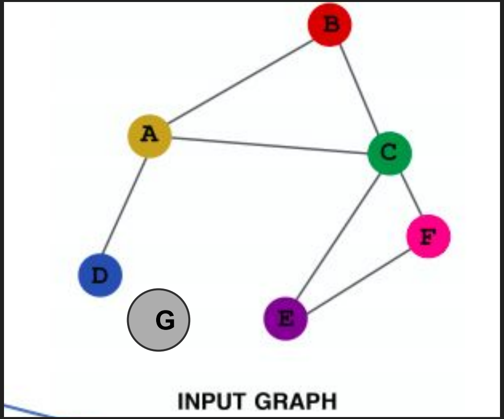
Train/ Val/ Test Nodes and Edges

- Val and test nodes determined by the node attribute.
- The same node can be used for both val and test.
- Train nodes are all nodes except val and test.



```
        self.val_nodes = [n for n in self.G.nodes() if self.G.node[n]['val']]
        self.test_nodes = [n for n in self.G.nodes() if self.G.node[n]['test']]

        self.no_train_nodes_set = set(self.val_nodes + self.test_nodes)
        self.train_nodes = set(G.nodes()).difference(self.no_train_nodes_set)
        # don't train on nodes that only have edges to test set
        self.train_nodes = [n for n in self.train_nodes if self.deg[id2idx[n]] > 0]
```

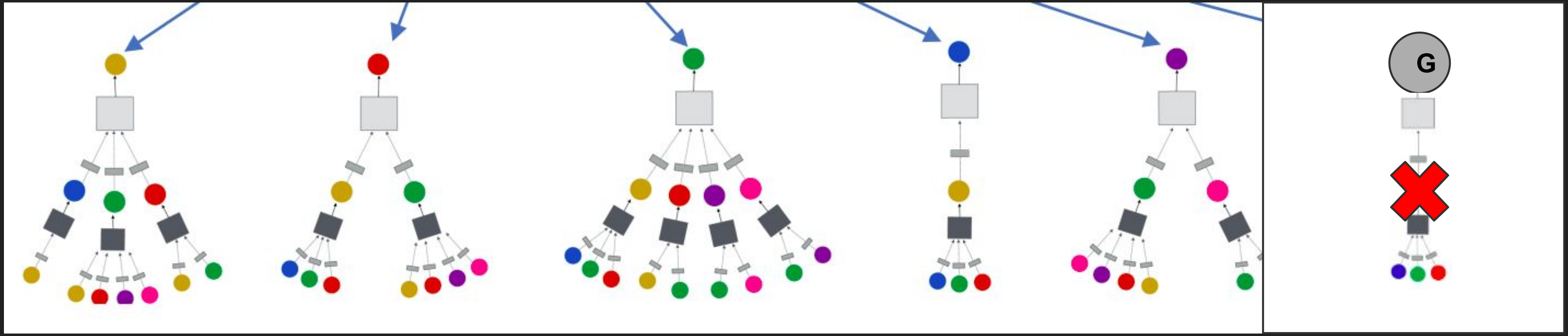
Implementation Challenge



INPUT GRAPH

Problem:
Isolated nodes have no neighbors for aggregation

Solution:
Create a dummy vector to be the neighbor for isolated nodes.



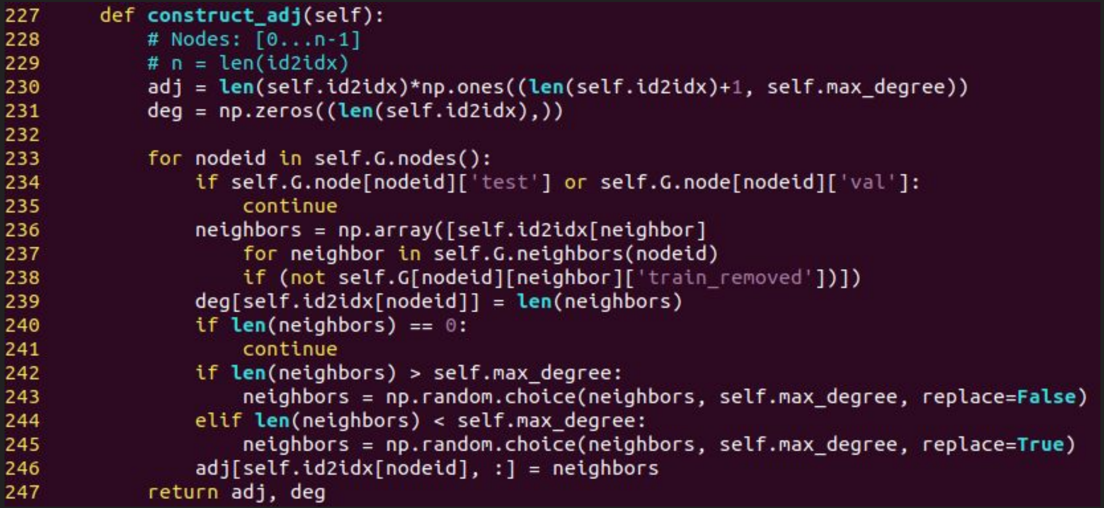
Adjacency List

Let max_degree = 128

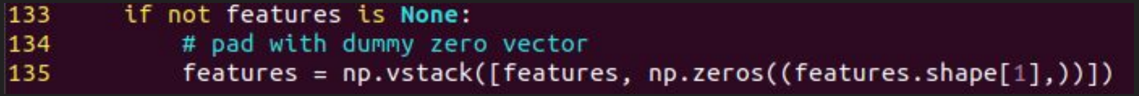
Number of neighbors:

- If (>128):
  - Randomly choose 128 neighbors
- If (<128):
  - Randomly repeat some neighbors
- If (0):
  - Only points to dummy node

Files: minibatch.py, supervised_train.py



```
    def construct_adj(self):
        # Nodes: [0...n-1]
        # n = len(id2idx)
        adj = len(self.id2idx)*np.ones((len(self.id2idx)+1, self.max_degree))
        deg = np.zeros((len(self.id2idx),))

        for nodeid in self.G.nodes():
            if self.G.node[nodeid]['test'] or self.G.node[nodeid]['val']:
                continue
            neighbors = np.array([self.id2idx[neighbor]
                for neighbor in self.G.neighbors(nodeid)
                if (not self.G[nodeid][neighbor]['train_removed'])])
            deg[self.id2idx[nodeid]] = len(neighbors)
            if len(neighbors) == 0:
                continue
            if len(neighbors) > self.max_degree:
                neighbors = np.random.choice(neighbors, self.max_degree, replace=False)
            elif len(neighbors) < self.max_degree:
                neighbors = np.random.choice(neighbors, self.max_degree, replace=True)
            adj[self.id2idx[nodeid], :] = neighbors
        return adj, deg
```



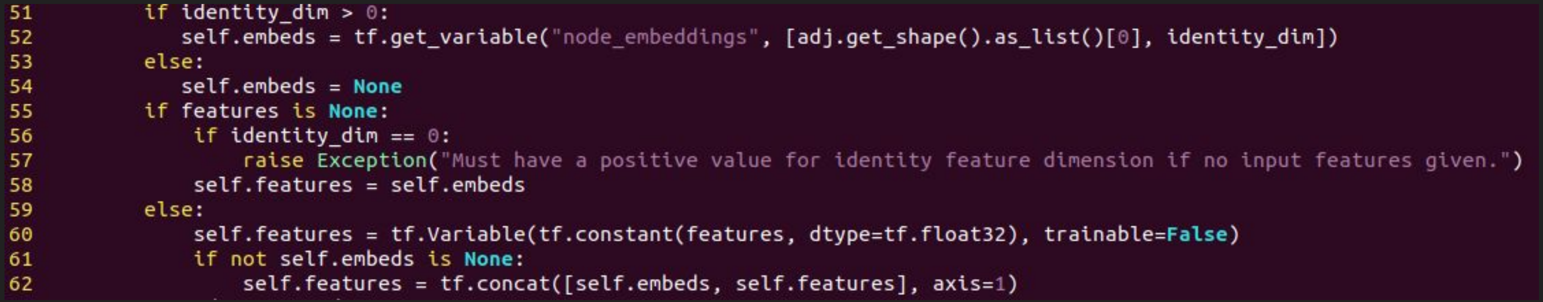
```
    if not features is None:
        # pad with dummy zero vector
        features = np.vstack([features, np.zeros((features.shape[1],))])
```

- Nodes #0…#(n-1)
  - Actual nodes
- Node #n
  - Dummy node

Identity Features

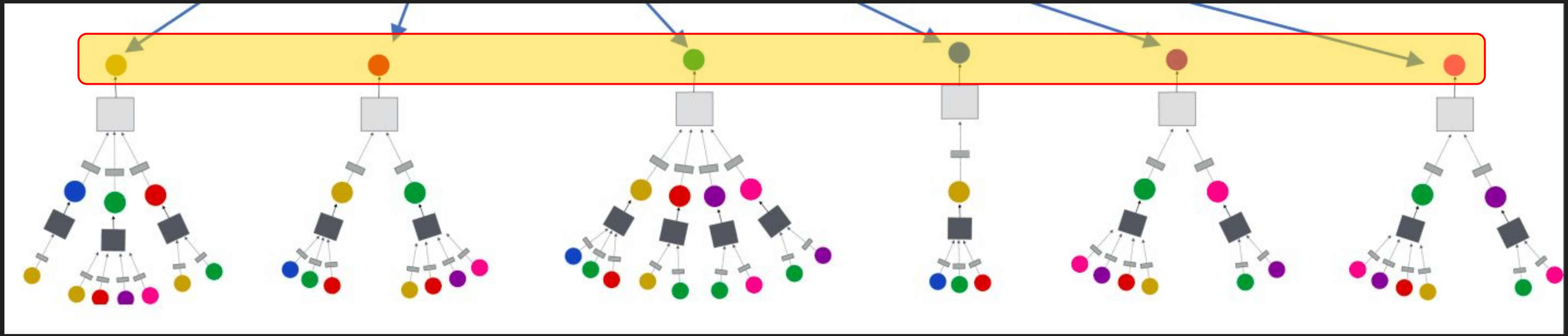
Files: supervised_models.py

- Identity features: randomly initialized features for each node.
- If “features” and “identity features” are both provided, they will be concatenated together.
    - E.g. 9-dimensional features + 50-dimensional identity features → 59-dimensional total features.
    - E.g. shape of “features” is (501, 9), shape of “embeds” is (501, 50) → shape of total features is (501, 59).



```
        if identity_dim > 0:
           self.embeds = tf.get_variable("node_embeddings", [adj.get_shape().as_list()[0], identity_dim])
        else:
           self.embeds = None
        if features is None:
            if identity_dim == 0:
                raise Exception("Must have a positive value for identity feature dimension if no input features given.")
            self.features = self.embeds
        else:
            self.features = tf.Variable(tf.constant(features, dtype=tf.float32), trainable=False)
            if not self.embeds is None:
                self.features = tf.concat([self.embeds, self.features], axis=1)
```

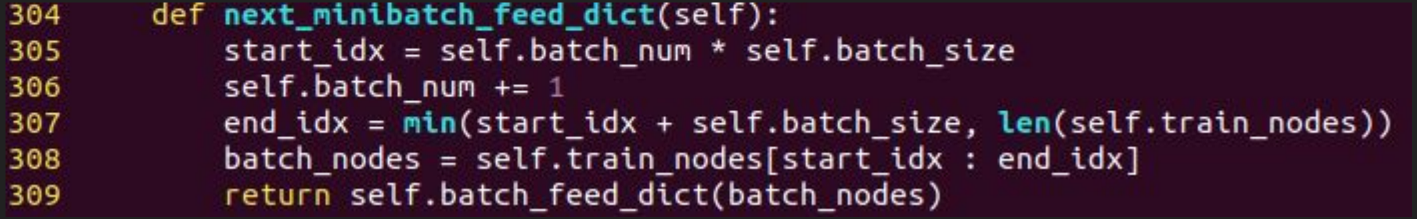
Node Sampling



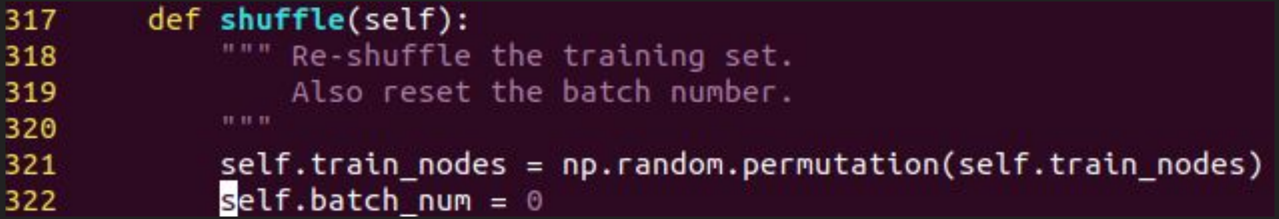
Node Sampling

Files: minibatch.py

- (line 308) nodes are sampled from train_nodes in order.
- Every epoch, train_nodes is shuffled.

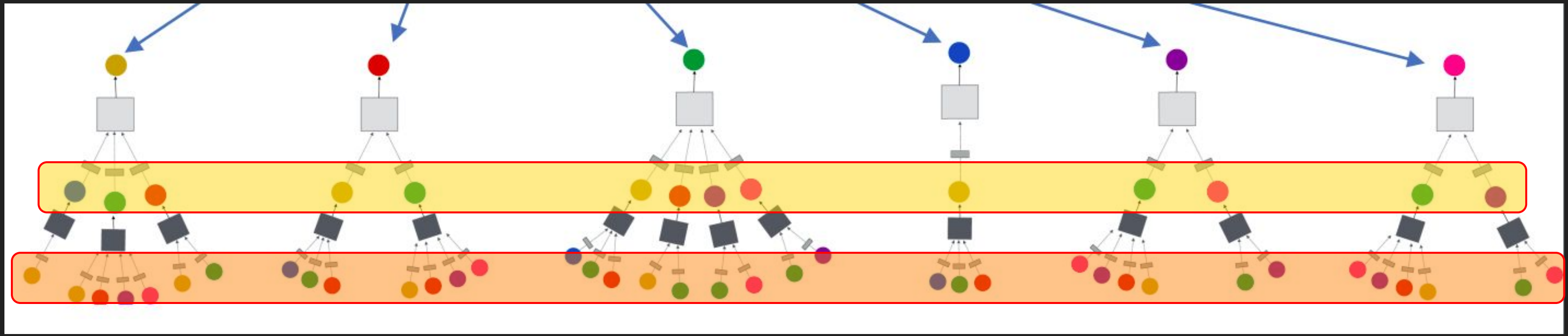


```
304     def next_minibatch_feed_dict(self):
305         start_idx = self.batch_num * self.batch_size
306         self.batch_num += 1
307         end_idx = min(start_idx + self.batch_size, len(self.train_nodes))
308         batch_nodes = self.train_nodes[start_idx : end_idx]
309         return self.batch_feed_dict(batch_nodes)
```



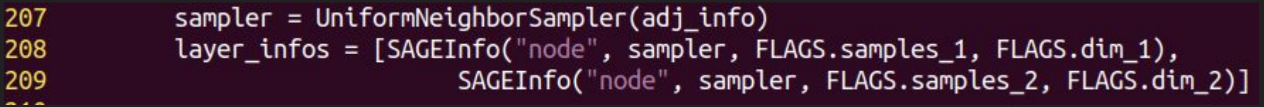
```
317     def shuffle(self):
318         """ Re-shuffle the training set.
319             Also reset the batch number.
320         """
321         self.train_nodes = np.random.permutation(self.train_nodes)
322         self.batch_num = 0
```

Neighbor Sampling

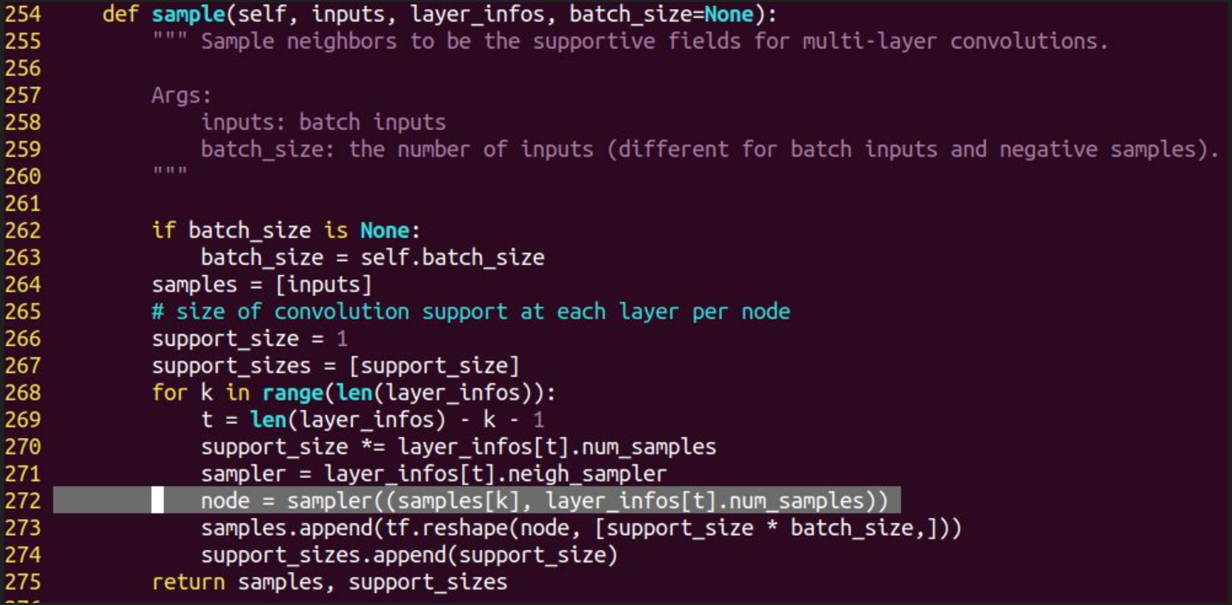


Neighbor Sampling

Files:
supervised_train.py
models.py



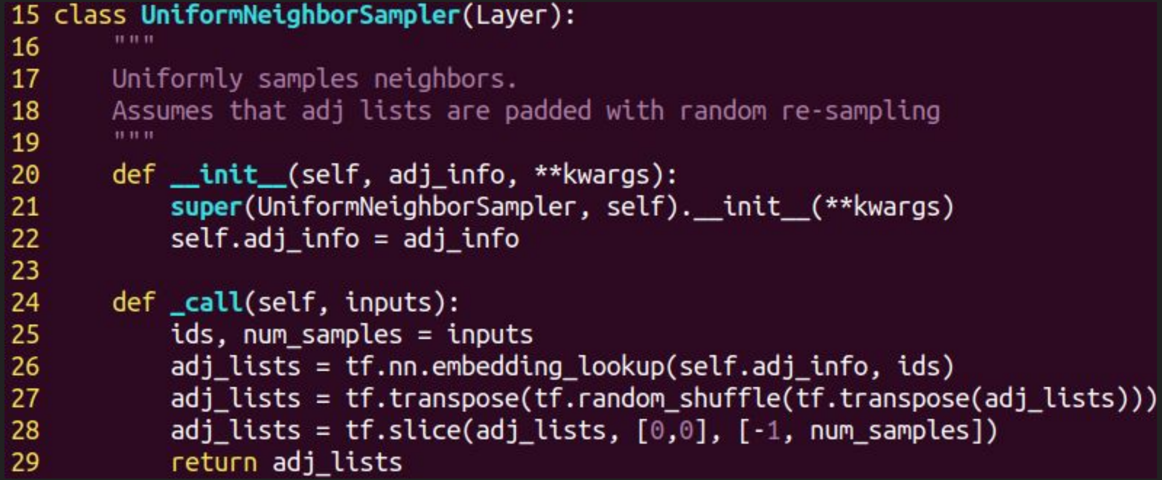
```
        sampler = UniformNeighborSampler(adj_info)
        layer_infos = [SAGEInfo("node", sampler, FLAGS.samples_1, FLAGS.dim_1),
                       SAGEInfo("node", sampler, FLAGS.samples_2, FLAGS.dim_2)]
```



```
    def sample(self, inputs, layer_infos, batch_size=None):
        """ Sample neighbors to be the supportive fields for multi-layer convolutions.

        Args:
            inputs: batch inputs
            batch_size: the number of inputs (different for batch inputs and negative samples).
        """

        if batch_size is None:
            batch_size = self.batch_size
        samples = [inputs]
        # size of convolution support at each layer per node
        support_size = 1
        support_sizes = [support_size]
        for k in range(len(layer_infos)):
            t = len(layer_infos) - k - 1
            support_size *= layer_infos[t].num_samples
            sampler = layer_infos[t].neigh_sampler
            node = sampler((samples[k], layer_infos[t].num_samples))
            samples.append(tf.reshape(node, [support_size * batch_size,]))
            support_sizes.append(support_size)
        return samples, support_sizes
```

Neighbor Sampling

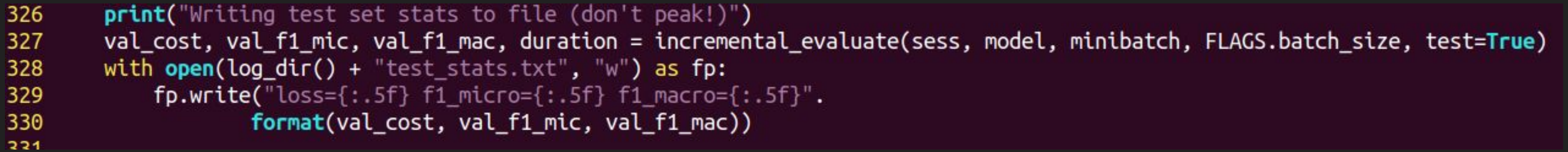
Files: neigh_samplers.py



```
class UniformNeighborSampler(Layer):
    """
    Uniformly samples neighbors.
    Assumes that adj lists are padded with random re-sampling
    """
    def __init__(self, adj_info, **kwargs):
        super(UniformNeighborSampler, self).__init__(**kwargs)
        self.adj_info = adj_info

    def _call(self, inputs):
        ids, num_samples = inputs
        adj_lists = tf.nn.embedding_lookup(self.adj_info, ids)
        adj_lists = tf.transpose(tf.random_shuffle(tf.transpose(adj_lists)))
        adj_lists = tf.slice(adj_lists, [0,0], [-1, num_samples])
        return adj_lists
```

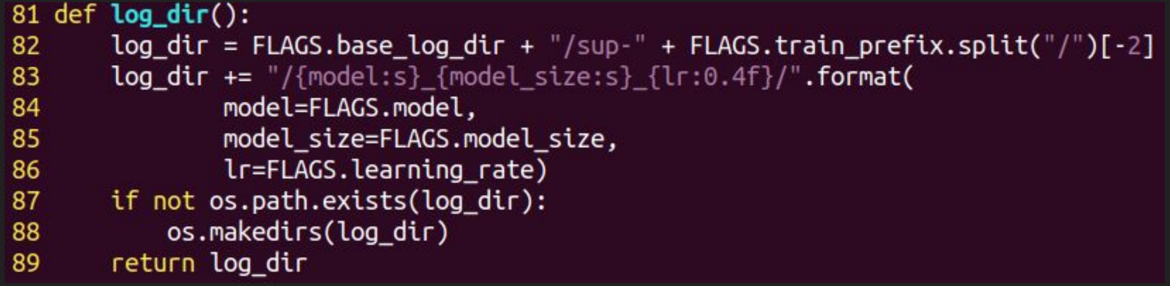
Test Stats

File: supervised_train.py



```
    print("Writing test set stats to file (don't peak!)")
    val_cost, val_f1_mic, val_f1_mac, duration = incremental_evaluate(sess, model, minibatch, FLAGS.batch_size, test=True)
    with open(log_dir() + "test_stats.txt", "w") as fp:
        fp.write("loss={:.5f} f1_micro={:.5f} f1_macro={:.5f}".
                format(val_cost, val_f1_mic, val_f1_mac))
```

- Log dir is:
    - ./sup-[data folder name]
- Test results are saved there.



```
def log_dir():
    log_dir = FLAGS.base_log_dir + "/sup-" + FLAGS.train_prefix.split("/")[-2]
    log_dir += "/{model:s}_{model_size:s}_{lr:0.4f}/".format(
            model=FLAGS.model,
            model_size=FLAGS.model_size,
            lr=FLAGS.learning_rate)
    if not os.path.exists(log_dir):
        os.makedirs(log_dir)
    return log_dir
```

Execution Script

Execution Script

In “GraphSAGE” folder:

-
  ```
  python -m graphsage.supervised_train --train_prefix data/processed/karate
  ```

Because the training data is small, adjust batch_size, print_every, etc.

-
  ```
  python -m graphsage.supervised_train --train_prefix data/processed/karate --batch_size 4 --print_every 2 --validate_iter 5
  ```

Done