Link Prediction Implementation

Link Prediction as binary classification

positive labels = edges

negative labels = sample of node pairs w/o edges

```
import dgl
import torch
import torch.nn as nn
import torch.nn.functional as F
import itertools
import numpy as np
import scipy.sparse as sp
```

```
dataset = dgl.data.CoraGraphDataset()
g = dataset[0]
```

load dataset

```
u, v = g.edges()
eids = np.random.permutation(np.arange(g.num_edges()))
```

```
test_size = int(len(eids) * 0.1)
train_size = g.num_edges() - test_size
```

```
test_pos_u, test_pos_v = u[eids[:test_size]], v[eids[:test_size]]
train_pos_u, train_pos_v = u[eids[test_size:]], v[eids[test_size:]]
```

sparse matrix in coordinate format

find negative edges

```
adj = sp.coo_matrix((np.ones(len(u)),
                     (u.numpy(), v.numpy())))
adj_neg = 1 - adj.todense() - np.eye(g.num_nodes())
neg_u, neg_v = np.where(adj_neg != 0)
```

```
neg_eids = np.random.choice(len(neg_u), g.num_edges())
test_neg_u, test_neg_v = neg_u[neg_eids[:test_size]],
                         neg_v[neg_eids[:test_size]]
train_neg_u, train_neg_v = neg_u[neg_eids[test_size:]],
                           neg_v[neg_eids[test_size:]]
```

We should remove "test edges" from the graph.

```
train_g = dgl.remove_edges(g, eids[:test_size])
```

Build a graph neural network: GraphSage

```
from dgl.nn import SAGEConv
```

```
class GraphSAGE (nn.Module):
    def __init__(self, in_feats, h_feats):
        super(GraphSAGE, self).__init__()
        self.Conv1 = SAGEConv(in_feats, h_feats, 'mean')
        self.Conv2 = SAGEConv(h_feats, h_feats, 'mean')

    def forward(self, g, in_feats):
        h = self.Conv1(g, in_feats)
        h = F.relu(h)
        h = self.Conv2(g, h)
        return h
```

* for link prediction, we need to compute representations for pairs of nodes.

* in link prediction : positive graph
  negative graph

```python
train_pos_g = dgl.graph((train_pos_u, train_pos_v),
                        num_nodes = g.num_nodes())
train_neg_g = dgl.graph((train_neg_u, train_neg_v),
                        num_nodes = g.num_nodes())
test_pos_g = dgl.graph((test_pos_u, test_pos_v),
                       num_nodes = g.num_nodes())
test_neg_g = dgl.graph((test_neg_u, test_neg_v),
                       num_nodes = g.num_nodes())
```

How to compute the node-pair repr?

1.

```
import dgl.function as fn
class DotPredictor(nn.Module):
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.apply_edges(fn.u_dot_v('h', 'h', 'score'))
```

Computing a new node feature → g.apply_edges

$\underbrace{\text{'h', 'h'}}_{\text{dot product of 'h' of src and dst}}$, $\underbrace{\text{'score'}}_{\text{new edge feature}}$

② More complex than dot product:

Compute a scalar score for each edge

```python
class MLPPredictor(nn.Module):
    def __init__(self, h_feats):
        super().__init__()
        self.W1 = nn.Linear(h_feats * 2, h_feats)
        self.W2 = nn.Linear(h_feats, 1)

    def apply_edges(self, edges):
        h = torch.cat([edges.src['h'], edges.dst['h']]
                      , 1)
        return {'score': self.W2(F.relu(self.W1(h)))
                         .squeeze(1)}

    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.apply_edges(self.apply_edges)
            return g.edata['score']
```

Training Loop

```
model = GraphSAGE(train_g.ndata['feat'].shape[1], 16)

pred = DotPredictor()

def get_loss(pos, neg):
    scores = torch.cat([pos, neg])
    labels = torch.cat([torch.ones(pos.shape[0]),
                        torch.zeros(neg.shape[0])])
    return F.binary_cross_entropy_with_logits
                        (scores, labels)

def get_auc(pos, neg):
    scores = torch.cat([pos, neg]).numpy()
    labels = torch.cat([torch.ones(pos.shape[0]),
                  torch.zeros(neg.shape[0])]).numpy()
    return roc_auc_score(labels, scores)

opt = torch.optim.Adam(itertools.chain(model.parameters(),
                                       pred.parameters()),
                  lr=0.01)
```

```
all_logits = []
for e in epochs:
```

forward step

```
h = model(train_g, train_g.ndata['feat'])
pos_score = pred(train_pos_g, h)
neg_score = pred(train_neg_g, h)
loss = get_loss(pos_score, neg_score)
```

backward step

```
opt.zero_grad()
loss.backward()
opt.step()
```

TEST

```
from sklearn.metrics import roc_auc_score
with torch.no_grad():
    pos = pred(test_pos_g, h)
    neg = pred(test_neg_g, h)
    auc = get_auc(pos, neg)
```