5.3 链接预测

(English Version)

在某些场景中，用户可能希望预测给定节点之间是否存在边，这样的任务称作 **链接预测** 任务。

概述

基于GNN的链接预测模型的基本思想是通过使用所需预测的节点对 u, v 的节点表示 $\boldsymbol{h}_u^{(L)}$ 和 $\boldsymbol{h}_v^{(L)}$，计算它们之间存在链接可能性的得分 $y_{u,v}$。其中 $\boldsymbol{h}_u^{(L)}$ 和 $\boldsymbol{h}_v^{(L)}$ 由多层GNN计算得出。

$$y_{u,v} = \phi(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)})$$

本节把节点 u 和 v 之间存在连接可能性的 *得分* 记作 $y_{u,v}$。

训练一个链接预测模型涉及到比对两个相连接节点之间的得分与任意一对节点之间的得分的差异。 例如，给定一条连接 u 和 v 的边，一个好的模型希望 u 和 v 之间的得分要高于 u 和从一个任意的噪声分布 $v' \sim Pn(v)$ 中所采样的节点 v' 之间的得分。 这样的方法称作 *负采样*。

许多损失函数都可以实现上述目标，包括但不限于。

- 交叉熵损失: $\mathcal{L} = -\log \sigma(y_{u,v}) - \sum_{v_i \sim P_n(v), i=1,\ldots,k} \log[1 - \sigma(y_{u,v_i})]$
- 贝叶斯个性化排序损失: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} -\log \sigma(y_{u,v} - y_{u,v_i})$
- 间隔损失: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} \max(0, M - y_{u,v} + y_{u,v_i})$, 其中 M 是常数项超参数。

如果用户熟悉 implicit feedback 和 noise-contrastive estimation， 可能会发现这些工作的想法都很类似。

计算 u 和 v 之间分数的神经网络模型与 5.2 边分类/回归 中所述的边回归模型相同。

下面是使用点积计算边得分的例子。

```
class DotProductPredictor(nn.Module):
    def forward(self, graph, h):
        # h是从5.1节的GNN模型中计算出的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return graph.edata['score']
```

模型的训练

因为上述的得分预测模型在图上进行计算，用户需要将负采样的样本表示为另外一个图，其中包含所有负采样的节点对作为边。

下面的例子展示了将负采样的样本表示为一个图。每一条边 (u, v) 都有 k 个对应的负采样样本 (u, v_i)，其中 v_i 是从均匀分布中采样的。

```
def construct_negative_graph(graph, k):
    src, dst = graph.edges()

    neg_src = src.repeat_interleave(k)
    neg_dst = torch.randint(0, graph.num_nodes(), (len(src) * k,))
    return dgl.graph((neg_src, neg_dst), num_nodes=graph.num_nodes())
```

预测边得分的模型和边分类/回归模型中的预测边得分模型相同。

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.sage = SAGE(in_features, hidden_features, out_features)
        self.pred = DotProductPredictor()
    def forward(self, g, neg_g, x):
        h = self.sage(g, x)
        return self.pred(g, h), self.pred(neg_g, h)
```

训练的循环部分里会重复构建负采样图并计算损失函数值。

```
def compute_loss(pos_score, neg_score):
    # 间隔损失
    n_edges = pos_score.shape[0]
    return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges, -1)).clamp(min=0).mean()

node_features = graph.ndata['feat']
n_features = node_features.shape[1]
k = 5
model = Model(n_features, 100, 100)
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    negative_graph = construct_negative_graph(graph, k)
    pos_score, neg_score = model(graph, negative_graph, node_features)
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

训练后，节点表示可以通过以下代码获取。

```
node_embeddings = model.sage(graph, node_features)
```

(实际应用中)，有着许多使用节点嵌入的方法，例如，训练下游任务的分类器，或为相关实体推荐进行最近邻搜索或最大内积搜索。

异构图上的链接预测模型的训练

异构图上的链接预测和同构图上的链接预测没有太大区别。下文是在一种边类型上进行预测， 用户可以很容易地将其拓展为对多种边类型上进行预测。

例如，为某一种边类型，用户可以重复使用 异构图上的边预测模型的训练 里的 `HeteroDotProductPredictor` 来计算节点间存在连接可能性的得分。

```
class HeteroDotProductPredictor(nn.Module):
    def forward(self, graph, h, etype):
        # h是从5.1节中对异构图的每种类型的边所计算的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'), etype=etype)
            return graph.edges[etype].data['score']
```

要执行负采样，用户可以对要进行链接预测的边类型构造一个负采样图。

```
def construct_negative_graph(graph, k, etype):
    utype, _, vtype = etype
    src, dst = graph.edges(etype=etype)
    neg_src = src.repeat_interleave(k)
    neg_dst = torch.randint(0, graph.num_nodes(vtype), (len(src) * k,))
    return dgl.heterograph(
        {etype: (neg_src, neg_dst)},
        num_nodes_dict={ntype: graph.num_nodes(ntype) for ntype in graph.ntypes})
```

该模型与异构图上边分类的模型有些不同，因为用户需要指定在哪种边类型上进行链接预测。

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, rel_names):
        super().__init__()
        self.sage = RGCN(in_features, hidden_features, out_features, rel_names)
        self.pred = HeteroDotProductPredictor()
    def forward(self, g, neg_g, x, etype):
        h = self.sage(g, x)
        return self.pred(g, h, etype), self.pred(neg_g, h, etype)
```

训练的循环部分和同构图时一致。

```
def compute_loss(pos_score, neg_score):
    # 间隔损失
    n_edges = pos_score.shape[0]
    return (1 - pos_score.unsqueeze(1) + neg_score.view(n_edges, -1)).clamp(min=0).mean()

k = 5
model = Model(10, 20, 5, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
node_features = {'user': user_feats, 'item': item_feats}
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    negative_graph = construct_negative_graph(hetero_graph, k, ('user', 'click', 'item'))
    pos_score, neg_score = model(hetero_graph, negative_graph, node_features, ('user', 'click',
'item'))
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```