Implement GNN to build a session-based recommender system

The dataset comes from RecSys Challenge 2015, which is aiming to build a session-based recommender system.

There are two main missions:

1. to predict the purchase depends on the click action
2. to predict the goods purchased

Here is the link to Dataset: [https://www.kaggle.com/chadgostopp/recsys-challenge-2015 (https://www.kaggle.com/chadgostopp/recsys-challenge-2015)](https://www.kaggle.com/chadgostopp/recsys-challenge-2015)

The dataset comprises two part:

1. yoochoose-clicks.dat the click action
2. yoochoose-buys.dat the purchase action

In [1]:

```
# firstly, do some processing on the input data

from sklearn.preprocessing import LabelEncoder
import pandas as pd
import numpy as np
```

In [2]:

```
df = pd.read_csv('C:\Project\DL\yoochoose-data\yoochoose-clicks.dat', header=None,low_memory=False)
# since the original data is not labeled by columns, rename the columns
df.columns = ['session_id', 'timestamp', 'item_id', 'category']

buy_df = pd.read_csv('C:\Project\DL\yoochoose-data\yoochoose-buys.dat', header=None,low_memory=False
buy_df.columns = ['session_id', 'timestamp', 'item_id', 'price', 'quantity']

df.head()
```

Out[2]:

	session_id	timestamp	item_id	category
0	1	2014-04-07T10:51:09.277Z	214536502	0
1	1	2014-04-07T10:54:09.868Z	214536500	0
2	1	2014-04-07T10:54:46.998Z	214536506	0
3	1	2014-04-07T10:57:00.306Z	214577561	0
4	2	2014-04-07T13:56:37.614Z	214662742	0

In [3]:

```
# using sci-kit to encode the label for better feature recognition
item_encoder = LabelEncoder()
df['item_id'] = item_encoder.fit_transform(df.item_id)
```

In [4]:

```
# sampling choose 100000 data, since my computer don't have GPU for cuda
# I tried to run more data, but it not worked without GPU
sampled_session_id = np.random.choice(df.session_id.unique(), 100000, replace=False)

# look up the choosed data feature
df = df.loc[df.session_id.isin(sampled_session_id)]
df.nunique()
```

Out[4]:

```
session_id    100000
timestamp     358461
item_id        20526
category         118
dtype: int64
```

In [5]:

```
# Add purchase information to the data set
df['label'] = df.session_id.isin(buy_df.session_id)
df.head()
```

Out[5]:

	session_id	timestamp	item_id	category	label
772	278	2014-04-01T20:28:59.485Z	39688	0	False
773	278	2014-04-01T20:29:00.694Z	39688	0	False
863	269	2014-04-01T08:35:45.863Z	39602	0	False
864	269	2014-04-01T08:36:06.076Z	39607	0	False
865	269	2014-04-01T08:36:34.683Z	39602	0	False

To make GNN, we need to first imagine a model. In this case, every session is a big graph and inside the node comprises two parts infomation:

1. whether it is purchased
2. the item id of the good

And every node in the graph is conencting together like linkedlist

In [6]:

```
import torch
from torch_geometric.data import InMemoryDataset
from torch_geometric.data import Data
from tqdm import tqdm

class YooChooseBinaryDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_transform=None):
        super(YooChooseBinaryDataset, self).__init__(root, transform, pre_transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_file_names(self):
        return []
    @property
    def processed_file_names(self):
        return ['./yoochoose_processed.dataset']

    def download(self):
        pass

    def process(self):

        data_list = []

        # process by session_id
        grouped = df.groupby('session_id')
        for session_id, group in tqdm(grouped):
            sess_item_id = LabelEncoder().fit_transform(group.item_id)
            group = group.reset_index(drop=True)
            group['sess_item_id'] = sess_item_id
            node_features = group.loc[group.session_id==session_id,['sess_item_id','item_id']].sort_

            node_features = torch.LongTensor(node_features).unsqueeze(1)
            target_nodes = group.sess_item_id.values[1:]
            source_nodes = group.sess_item_id.values[:-1]

            edge_index = torch.tensor([source_nodes, target_nodes], dtype=torch.long)
            x = node_features

            y = torch.FloatTensor([group.label.values[0]])

            data = Data(x=x, edge_index=edge_index, y=y)
            data_list.append(data)

        data, slices = self.collate(data_list)
        torch.save((data, slices), self.processed_paths[0])
```

In [7]:

```
# Separate the three data sets
dataset = YooChooseBinaryDataset("./")
dataset = dataset.shuffle()
train_dataset = dataset[:80000]
val_dataset = dataset[80000:90000]
test_dataset = dataset[90000:]
len(train_dataset), len(val_dataset), len(test_dataset)
```

```
Processing...
  0%|
| 0/100000 [00:00<?, ?it/s]C:\Users\Steven\anaconda3\envs\pytorch\lib\site-packages
\ipykernel_launcher.py:37: UserWarning: Creating a tensor from a list of numpy.ndarr
ays is extremely slow. Please consider converting the list to a single numpy.ndarray
with numpy.array() before converting to a tensor. (Triggered internally at  ..\torch
\csrc\utils\tensor_new.cpp:201.)
C:\Users\Steven\anaconda3\envs\pytorch\lib\site-packages\ipykernel_launcher.py:40: D
eprecationWarning: In future, it will be an error for 'np.bool_' scalars to be inter
preted as an index
100%|██████████████████████████████████████████████████████████████████████████████
██████████████████████████████████████████████████████████| 100000/100000
[05:50<00:00, 284.96it/s]
Done!
```

Out[7]:

```
(80000, 10000, 10000)
```

In this Neural Network, we need to choose a MessagePassing method. I use the SageConv layer coming from **Inductive Representation Learning on Large Graphs**. And here is the link to the paper.

http://papers.nips.cc/paper/6703-inductive-representation-learningon-large-graphs.pdf (http://papers.nips.cc/paper/6703-inductive-representation-learningon-large-graphs.pdf)

In [8]:

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import TopKPooling
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp

embed_dim = 128

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = SAGEConv(embed_dim, 128)
        self.pool1 = TopKPooling(128, ratio=0.8)
        self.conv2 = SAGEConv(128, 128)
        self.pool2 = TopKPooling(128, ratio=0.8)
        self.conv3 = SAGEConv(128, 128)
        self.pool3 = TopKPooling(128, ratio=0.8)
        self.item_embedding = torch.nn.Embedding(num_embeddings=df.item_id.max() +1, embedding_dim=e
        self.lin1 = torch.nn.Linear(256, 128)
        self.lin2 = torch.nn.Linear(128, 64)
        self.lin3 = torch.nn.Linear(64, 1)
        self.bn1 = torch.nn.BatchNorm1d(128)
        self.bn2 = torch.nn.BatchNorm1d(64)
        self.act1 = torch.nn.ReLU()
        self.act2 = torch.nn.ReLU()

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = self.item_embedding(x)
        x = x.squeeze(1)
        x = F.relu(self.conv1(x, edge_index))
        x, edge_index, _, batch, _, _ = self.pool1(x, edge_index, None, batch)
        x1 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
        x = F.relu(self.conv2(x, edge_index))
        x, edge_index, _, batch, _, _ = self.pool2(x, edge_index, None, batch)
        x2 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
        x = F.relu(self.conv3(x, edge_index))
        x, edge_index, _, batch, _, _ = self.pool3(x, edge_index, None, batch)
        x3 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
        x = x1 + x2 + x3
        x = self.lin1(x)
        x = self.act1(x)
        x = self.lin2(x)
        x = self.act2(x)
        x = F.dropout(x, p=0.5, training=self.training)
        x = torch.sigmoid(self.lin3(x)).squeeze(1)
        return x
```

In [9]:

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import remove_self_loops, add_self_loops


class SAGEConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(SAGEConv, self).__init__(aggr='max')  # "Max" aggregation.
        self.lin = torch.nn.Linear(in_channels, out_channels)
        self.act = torch.nn.ReLU()
        self.update_lin = torch.nn.Linear(in_channels + out_channels, in_channels, bias=False)
        self.update_act = torch.nn.ReLU()

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        edge_index, _ = remove_self_loops(edge_index)
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)

    def message(self, x_j):
        # x_j has shape [E, in_channels]

        x_j = self.lin(x_j)
        x_j = self.act(x_j)

        return x_j

    def update(self, aggr_out, x):
        # aggr_out has shape [N, out_channels]

        new_embedding = torch.cat([aggr_out, x], dim=1)

        new_embedding = self.update_lin(new_embedding)
        new_embedding = self.update_act(new_embedding)
        return new_embedding
```

In [10]:

```
from torch_geometric.data import DataLoader

def train():
    model.train()

    loss_all = 0
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        output = model(data)
        label = data.y.to(device)
        loss = crit(output, label)
        loss.backward()
        loss_all += data.num_graphs * loss.item()
        optimizer.step()
    return loss_all / len(train_dataset)

device = torch.device('cpu')
model = Net().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
crit = torch.nn.BCELoss()
```

In [11]:

```
from sklearn.metrics import roc_auc_score
def evaluate(loader):
    model.eval()

    predictions = []
    labels = []

    with torch.no_grad():
        for data in loader:

            data = data.to(device)
            pred = model(data).detach().cpu().numpy()

            label = data.y.detach().cpu().numpy()
            predictions.append(pred)
            labels.append(label)
    predictions = np.hstack(predictions)
    labels = np.hstack(labels)
    return roc_auc_score(labels,predictions)
```

In [12]:

```
from torch_geometric.loader import DataLoader
train_loader = DataLoader(train_dataset, batch_size=64)
val_loader = DataLoader(val_dataset, batch_size=64)
test_loader = DataLoader(test_dataset, batch_size=64)
for epoch in range(3):
    loss = train()
    train_acc = evaluate(train_loader)
    val_acc = evaluate(val_loader)
    test_acc = evaluate(test_loader)
    print('Epoch: {:03d}, Loss: {:.5f}, Train Auc: {:.5f}, Val Auc: {:.5f}, Test Auc: {:.5f}'.
          format(epoch, loss, train_acc, val_acc, test_acc))
```

```
Epoch: 000, Loss: 0.21942, Train Auc: 0.71395, Val Auc: 0.66066, Test Auc: 0.64988
Epoch: 001, Loss: 0.19286, Train Auc: 0.78787, Val Auc: 0.66170, Test Auc: 0.63384
Epoch: 002, Loss: 0.17371, Train Auc: 0.79901, Val Auc: 0.65447, Test Auc: 0.63127
```

Conclusion

In this attempt, I tried to use pytorch and pytorch geometric to simulate implementation graph neural network. Actually, the dataset is not designed for GNN, but we could see that GNN has a wide range of applications. And even for the small dataset, it can really have good performance. In this case, for 100000 data, it cost less time to train and can give a Test Auc near 0.65.