Understanding Meta-data in PTUPCDR Code

Src = Movies and TV
Tgt = CDs and Vinyl

1. First instance of meta_data starts from **split** function which is called from main function. This function gives the meta-data and stores it in **train_meta['pos_seq']** dataframe. The data is saved in **/train_meta.csv** file.

Preprocessing.py /main(self):

train_src, train_tgt, train_meta, test = self.split(src, tgt)

Preprocessing.py / split(self, src, tgt):

pos_seq_dict = self.get_history(src, co_users)

➔

This function will return the pos_seq_dict dictionary mapping pos_seq_dict[uid] = pos. Basically it will give the uid's from source data for the users who have rated greater than 3 for the corresponding item.

```
def get_history(self, data, uid_set):
    pos_seq_dict = {}
    for uid in tqdm.tqdm(uid_set):
        pos = data[(data.uid == uid) & (data.y > 3)].iid.values.tolist()
        pos_seq_dict[uid] = pos
    return pos_seq_dict
```

train_meta = tgt[tgt['uid'].isin(co_users - test_users)]

➔ train_meta consist of values from target set where uids present in (co_users-test_users) set. It's a dataframe.

train_meta['pos_seq'] = train_meta['uid'].map(pos_seq_dict)

➔ pos_seq_dict is a dictionary where the keys are the unique values in the 'uid' column and the values are the corresponding values to be mapped, this line of code will create a new column called 'pos_seq' in the train_meta DataFrame. The values in this new column will be the mapped values from the pos_seq_dict dictionary based on the matching 'uid' values. So basically we will have data with uid == key value of pos_seq_dict in train_meta['pos_seq'].

return train_src, train_tgt, **train_meta**, test

Preprocessing.py / save(self, train_src, train_tgt, train_meta, test):

```
train_meta.to_csv(output_root + '/train_meta.csv', sep=',', header=None, index=False)
```

2. Get_data function loads the metadata into data_meta variable. The use of meta data starts from CDR function where the base model for PTUPCDR been trained on this data. In MFBasedModel function, for the train_meta if-else code snippet, the main algorithm for PTUPCDR is written.

run.py/ get_data(self):

```
data_meta = self.read_log_data(self.meta_path, self.batchsize_meta, history=True)
```

➔ Again the data is passed through read_log_data function which will transformed the data into tensor, combining uid, iid and ratings.

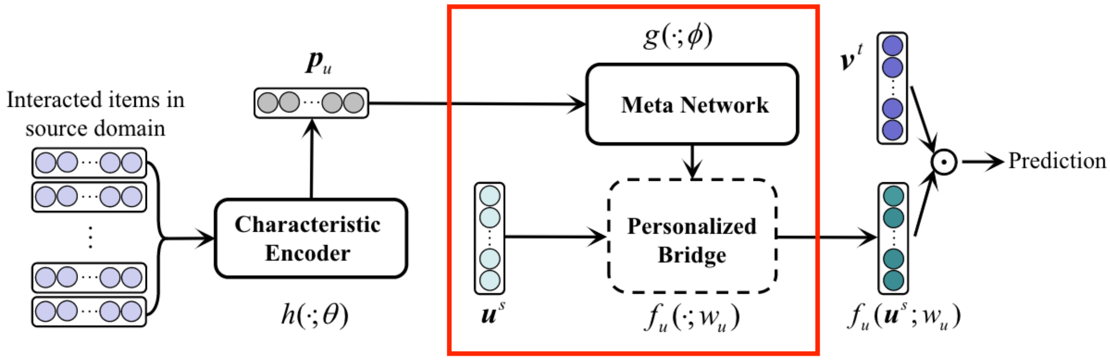
run.py/ CDR(..., data_meta,...):

```
# Under PTUPCDR model
self.train(data_meta, model, criterion, optimizer_meta, i, stage='train_meta')
```

run.py/ train(self, **data_loader**, model, criterion, optimizer, epoch, stage, mapping=False):

```
model.train()
for X, y in tqdm.tqdm(data_loader, smoothing=0, mininterval=1.0):
    if mapping:
        src_emb, tgt_emb = model(X, stage)
        loss = criterion(src_emb, tgt_emb)
    else:
        pred = model(X, stage)
        loss = criterion(pred, y.squeeze().float())
    model.zero_grad()
    loss.backward()
    optimizer.step()
```

models.py/MFBasedModels/forward (self, x, stage)/:



➔ This is the execution of the above entire diagram of the PTUPCDR model. Embeddings are created using using pytorch neural network models like this **torch.nn.Embedding(uid_all, emb_dim).** Then **Meta network** is created by selecting transferable features from source item-id embeddings and feeding them into the mapping function which maps the transferrable features of the embeddings and the data x. This is done using **MetaNet** function which is a neural network made up of sequential nn layers like linear, relu and softmax. Then the **Personalized bridged** is created by feeding this Meta Network output and source user embeddings to batch-multiplication bnm function which multiplies them to form the personalized bridge for each user. The supporting functions for this code are given below for reference.

```
elif stage in ['train_meta', 'test_meta']:
        iid_emb = self.tgt_model.iid_embedding(x[:, 1].unsqueeze(1))
        uid_emb_src = self.src_model.uid_embedding(x[:, 0].unsqueeze(1))
        ufea = self.src_model.iid_embedding(x[:, 2:])
        mapping = self.meta_net.forward(ufea, x[:, 2:]).view(-1, self.emb_dim,
self.emb_dim)
        uid_emb = torch.bmm(uid_emb_src, mapping)
        emb = torch.cat([uid_emb, iid_emb], 1)
        output = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
         return output
```

Functions for reference:

```
class MetaNet(torch.nn.Module):
   def __init__(self, emb_dim, meta_dim):
      super().__init__()
      self.event_K = torch.nn.Sequential(torch.nn.Linear(emb_dim, emb_dim), torch.nn.ReLU(),
```

```
                                torch.nn.Linear(emb_dim, 1, False))
        self.event_softmax = torch.nn.Softmax(dim=1)
        self.decoder = torch.nn.Sequential(torch.nn.Linear(emb_dim, meta_dim), torch.nn.ReLU(),
                                torch.nn.Linear(meta_dim, emb_dim * emb_dim))

    def forward(self, emb_fea, seq_index):
        mask = (seq_index == 0).float()
        event_K = self.event_K(emb_fea)
        t = event_K - torch.unsqueeze(mask, 2) * 1e8
        att = self.event_softmax(t)
        his_fea = torch.sum(att * emb_fea, 1)
        output = self.decoder(his_fea)
        return output.squeeze(1)


class LookupEmbedding(torch.nn.Module):

    def __init__(self, uid_all, iid_all, emb_dim):
        super().__init__()
        self.uid_embedding = torch.nn.Embedding(uid_all, emb_dim)
        self.iid_embedding = torch.nn.Embedding(iid_all + 1, emb_dim)

    def forward(self, x):
        uid_emb = self.uid_embedding(x[:, 0].unsqueeze(1))
        iid_emb = self.iid_embedding(x[:, 1].unsqueeze(1))
        emb = torch.cat([uid_emb, iid_emb], dim=1)
        return emb


class MFBasedModel(torch.nn.Module):
    def __init__(self, uid_all, iid_all, num_fields, emb_dim, meta_dim_0):
        super().__init__()
        self.num_fields = num_fields
        self.emb_dim = emb_dim
        self.src_model = LookupEmbedding(uid_all, iid_all, emb_dim)
        self.tgt_model = LookupEmbedding(uid_all, iid_all, emb_dim)
        self.aug_model = LookupEmbedding(uid_all, iid_all, emb_dim)
        self.meta_net = MetaNet(emb_dim, meta_dim_0)
        self.mapping = torch.nn.Linear(emb_dim, emb_dim, False)

    def forward(self, x, stage):
        if stage == 'train_src':
            emb = self.src_model.forward(x)
            x = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
```

```
            return x
        elif stage in ['train_tgt', 'test_tgt']:
            emb = self.tgt_model.forward(x)
            x = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
            return x
        elif stage in ['train_aug', 'test_aug']:
            emb = self.aug_model.forward(x)
            x = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
            return x
        elif stage in ['train_meta', 'test_meta']:
            iid_emb = self.tgt_model.iid_embedding(x[:, 1].unsqueeze(1))
            uid_emb_src = self.src_model.uid_embedding(x[:, 0].unsqueeze(1))
            ufea = self.src_model.iid_embedding(x[:, 2:])
            mapping = self.meta_net.forward(ufea, x[:, 2:]).view(-1, self.emb_dim, self.emb_dim)
            uid_emb = torch.bmm(uid_emb_src, mapping)
            emb = torch.cat([uid_emb, iid_emb], 1)
            output = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
            return output
        elif stage == 'train_map':
            src_emb = self.src_model.uid_embedding(x.unsqueeze(1)).squeeze()
            src_emb = self.mapping.forward(src_emb)
            tgt_emb = self.tgt_model.uid_embedding(x.unsqueeze(1)).squeeze()
            return src_emb, tgt_emb
        elif stage == 'test_map':
            uid_emb = self.mapping.forward(self.src_model.uid_embedding(x[:,
0].unsqueeze(1)).squeeze())
            emb = self.tgt_model.forward(x)
            emb[:, 0, :] = uid_emb
            x = torch.sum(emb[:, 0, :] * emb[:, 1, :], dim=1)
            return x
```