**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)

1. 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)/:

A diagram of a bridge

Description automatically generated with medium confidence

🡺 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