Title: Graph-Based Recommendation System for Electronic Products

Objective

The primary goal of this project is to develop a recommendation system for electronic products using Graph Neural Networks (GNNs). By leveraging user-product interaction data, the system aims to predict user ratings for products they have not yet rated, thus providing personalized recommendations.

Data Description

The dataset used in this project is a CSV file (amazon-product-reviews) containing the following columns:

- `user_id`: Unique identifier for each user.
- `product_id`: Unique identifier for each product.
- `rating`: The rating given by the user to the product.
- `time_stamp`: The time when the rating was provided.

Methodology

1. **Data Preprocessing**:
   - Handle missing values.
   - Encode `user_id` and `product_id` using Label Encoding.
   - Split the dataset into training and validation sets.
2. **Graph Construction**:
   - Define nodes as users and items.
   - Create edges based on user-item interactions (ratings).
   - Represent the graph using multi-dimensional arrays (torch.tensor) to handle memory efficiently.
3. **Model Development**:
   - Choose a suitable GNN architecture (e.g., Graph Convolutional Networks (GCN), GraphSAGE, or Graph Attention Networks) I use GCN
   - Implement the GCN model using PyTorch Geometric.
   - Train and validate the model using the constructed graph data.
4. **Evaluation**:
   - Evaluate the model's performance using appropriate metrics.

Challenges

- Managing large datasets with limited computational resources.
- Ensuring efficient data preprocessing and graph construction.
- Selecting the appropriate GNN architecture and tuning hyperparameters for optimal performance.

Conclusion

The project demonstrates the application of GNNs in building an effective recommendation system for electronic products. By capturing complex user-item interaction patterns, the model provides personalized recommendations, enhancing user experience and engagement.

1. Data loading and preprocessing

1.1 import Libraries

```
import pandas as pd
import numpy as np
import torch
from torch_geometric.data import Data
from sklearn.preprocessing import LabelEncoder
# from torch_geometric.data import DataLoader
from torch_geometric.loader import DataLoader
from sklearn.model_selection import train_test_split
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
import torch.optim as optim
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score

import warnings
warnings.filterwarnings("ignore")
```

1.2 Preprocessing

```
file_path = 'ratings_Electronics (1).csv'
df = pd.read_csv(file_path)
df
```

	AKM1MP6P0OYPR	0132793040	5.0	1365811200
0	A2CX7LUOHB2NDG	0321732944	5.0	1341100800
1	A2NWSAGRHCP8N5	0439886341	1.0	1367193600
2	A2WNBOD3WNDNKT	0439886341	3.0	1374451200
3	A1GI0U4ZRJA8WN	0439886341	1.0	1334707200
4	A1QGNMC6O1VW39	0511189877	5.0	1397433600
...	...	...	...	...
7824476	A2YZI3C9MOHC0L	BT008UKTMW	5.0	1396569600
7824477	A322MDK0M89RHN	BT008UKTMW	5.0	1313366400
7824478	A1MH90R0ADMIK0	BT008UKTMW	4.0	1404172800
7824479	A10M2KEFPEQDHN	BT008UKTMW	4.0	1297555200
7824480	A2G81TMIOIDEQQ	BT008V9J9U	5.0	1312675200

[7824481 rows x 4 columns]

```
df.rename(columns = {'AKM1MP6P0OYPR':'user_id',
'0132793040':'product_id', '5.0':'rating', '1365811200':'time_stamp'},
```

```
inplace = True)
df

                user_id  product_id  rating  time_stamp
0        A2CX7LUOHB2NDG  0321732944     5.0  1341100800
1        A2NWSAGRHCP8N5  0439886341     1.0  1367193600
2        A2WNBOD3WNDNKT  0439886341     3.0  1374451200
3        A1GI0U4ZRJA8WN  0439886341     1.0  1334707200
4        A1QGNMC6O1VW39  0511189877     5.0  1397433600
...                 ...         ...     ...         ...
7824476  A2YZI3C9MOHC0L  BT008UKTMW     5.0  1396569600
7824477  A322MDK0M89RHN  BT008UKTMW     5.0  1313366400
7824478  A1MH90R0ADMIK0  BT008UKTMW     4.0  1404172800
7824479  A10M2KEFPEQDHN  BT008UKTMW     4.0  1297555200
7824480  A2G81TMIOIDEQQ  BT008V9J9U     5.0  1312675200

[7824481 rows x 4 columns]

user_counts = df['user_id'].value_counts()
# we only need that users which use multiple times   # 4942648
multiple times and 2881833 one time
filtered_df =df[df['user_id'].isin(user_counts[user_counts >
1].index)]
filtered_df

                user_id  product_id  rating  time_stamp
0        A2CX7LUOHB2NDG  0321732944     5.0  1341100800
4        A1QGNMC6O1VW39  0511189877     5.0  1397433600
5        A3J3BRHTDRFJ2G  0511189877     2.0  1397433600
6        A2TY0BTJOTENPG  0511189877     5.0  1395878400
7        A34ATBPOK6HCHY  0511189877     5.0  1395532800
...                 ...         ...     ...         ...
7824474  A2R6Q6KJCYSVH7  BT008UKTMW     3.0  1343520000
7824476  A2YZI3C9MOHC0L  BT008UKTMW     5.0  1396569600
7824477  A322MDK0M89RHN  BT008UKTMW     5.0  1313366400
7824478  A1MH90R0ADMIK0  BT008UKTMW     4.0  1404172800
7824480  A2G81TMIOIDEQQ  BT008V9J9U     5.0  1312675200

[4942648 rows x 4 columns]

# shuffle data
# take a sample data of the population
result_df = filtered_df.sample(frac=0.002,
random_state=1).reset_index(drop=True)
result_df

             user_id  product_id  rating  time_stamp
0     A3554P57JKXVJN  B0085S0838     1.0  1397952000
1     A2CS4FEMSETJT1  B00IBCQJZO     5.0  1400198400
2     A2CIJE4EUZF2WW  B00C59X93O     5.0  1405209600
3     A15ZX3XV2L7QDH  B001V9LPT4     4.0  1359936000
```

```
4      AZGORQNAAGIMW  B00005NIMJ     3.0  1281657600
...              ...         ...     ...         ...
9880  A1GS1EX68K53ZS  B00007OO24     5.0  1214784000
9881  A32YO3HN3TPTNG  B001212ELY     5.0  1238457600
9882  A2K6OSG7JT8ZOA  B005455PW4     2.0  1402876800
9883  A3202C86H3AD1V  B0000O3LUKC    1.0  1257465600
9884  A3LZB8E6ZU78XD  B0001KWGOW     5.0  1133481600

[9885 rows x 4 columns]
```

```
result_df["rating"].value_counts()
```

```
rating
5.0    5729
4.0    1883
1.0     900
3.0     823
2.0     550
Name: count, dtype: int64
```

```
result_df.to_csv('filtered_data.csv', index=False)
```

```
# Load the data
data = pd.read_csv('filtered_data.csv')
```

```
data.head()
```

```
          user_id  product_id  rating  time_stamp
0  A3554P57JKXVJN  B0085S0838     1.0  1397952000
1  A2CS4FEMSETJT1  B00IBCQJZO     5.0  1400198400
2  A2CIJE4EUZF2WW  B00C59X93O     5.0  1405209600
3  A15ZX3XV2L7QDH  B001V9LPT4     4.0  1359936000
4   AZGORQNAAGIMW  B00005NIMJ     3.0  1281657600
```

```
data.isnull().sum()
```

```
user_id       0
product_id    0
rating        0
time_stamp    0
dtype: int64
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9885 entries, 0 to 9884
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   user_id     9885 non-null   object
 1   product_id  9885 non-null   object
 2   rating      9885 non-null   float64
```

```
 3   time_stamp  9885 non-null   int64
dtypes: float64(1), int64(1), object(2)
memory usage: 309.0+ KB
```

```
# convert categorical to numeric. Encode user IDs and product IDs
user_encoder = LabelEncoder()
item_encoder = LabelEncoder()

data['user_id']= user_encoder.fit_transform(data['user_id'])
data['product_id'] = item_encoder.fit_transform(data['product_id'])

data.head(2)
```

```
   user_id  product_id  rating  time_stamp
0     5546        6130     1.0  1397952000
1     3497        7825     5.0  1400198400
```

Split the Data

First, split the dataset into training, validation, and test sets.

```
train_data, temp_data = train_test_split(data, test_size=0.4,
random_state=1)
val_data, test_data = train_test_split(temp_data, test_size=0.5,
random_state=1)

train_data.shape, temp_data.shape, val_data.shape, test_data.shape
```

```
((5931, 4), (3954, 4), (1977, 4), (1977, 4))
```

```
train_data
```

```
      user_id  product_id  rating  time_stamp
5869     1903        6357     4.0  1394755200
1902     6751        6509     5.0  1381449600
8989     8742        2326     3.0  1374278400
7765     5189        6789     3.0  1404950400
8348     2765        4460     5.0  1363132800
...       ...         ...     ...         ...
2895     8175        2007     5.0  1251763200
7813     9080        3889     5.0  1390003200
905      5221        2517     4.0  1369440000
5192     2493        2769     4.0  1304380800
235      3429        2598     4.0  1244937600

[5931 rows x 4 columns]
```

```
train_data['rating'].value_counts()
```

```
rating
5.0    3398
```

```
4.0    1178
1.0     534
3.0     496
2.0     325
Name: count, dtype: int64
```

```
train_data['user_id'].values
```

```
array([1903, 6751, 8742, ..., 5221, 2493, 3429])
```

2 Graph Construction

```
num_users = data['user_id'].nunique()
num_items = data['product_id'].nunique()

num_nodes = num_users + num_items
num_nodes
```

```
17693
```

```
# Create node features
node_features = torch.eye(num_nodes)
node_features
```

```
tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 1., 0., 0.],
        [0., 0., 0.,  ..., 0., 1., 0.],
        [0., 0., 0.,  ..., 0., 0., 1.]])
```

train data to PyTorch Geometric format

```
# Create edge index from user-item interactions
train_edge_index = torch.tensor([train_data['user_id'].values,
train_data['product_id'].values], dtype=torch.long)

# Create edge attributes (ratings)
train_edge_attr = torch.tensor(train_data['rating'].values,
dtype=torch.float)


# Create the PyTorch Geometric data object
train_dataa = Data(edge_index=train_edge_index,
edge_attr=train_edge_attr)
train_dataa.x=node_features
train_dataa
```

```
Data(edge_index=[2, 5931], edge_attr=[5931], x=[17693, 17693])
```

validation and test data to PyTorch Geometric format

```
# Convert validation and test data to PyTorch Geometric format
val_edge_index = torch.tensor([val_data['user_id'].values,
val_data['product_id'].values], dtype=torch.long)
val_edge_attr = torch.tensor(val_data['rating'].values,
dtype=torch.float)

test_edge_index = torch.tensor([test_data['user_id'].values,
test_data['product_id'].values], dtype=torch.long)
test_edge_attr = torch.tensor(test_data['rating'].values,
dtype=torch.float)

# Create data objects for validation and test sets
val_dataa = Data(edge_index=val_edge_index, edge_attr=val_edge_attr,
x=node_features)
test_dataa = Data(edge_index=test_edge_index,
edge_attr=test_edge_attr, x=node_features)

print(f"{val_dataa} \n {test_dataa}")
```

```
Data(x=[17693, 17693], edge_index=[2, 1977], edge_attr=[1977])
 Data(x=[17693, 17693], edge_index=[2, 1977], edge_attr=[1977])
```

```
print(f"Max and min train edge values:
{train_edge_index.max().item(),train_edge_index.min().item()}, max
train edge attr:
{train_edge_attr.max().item(),train_edge_attr.min().item()}, Num
nodes: {num_users + num_items}")
print(f"Max and min val edge values:
{val_edge_index.max().item(),val_edge_index.min().item()}, Max and min
val attr:{val_edge_attr.max().item(),val_edge_attr.min().item()}, Num
nodes: {num_users + num_items}")
print(f"Max and min test edge values:
{test_edge_index.max().item(),test_edge_index.min().item()}, Max and
min val attr:
{test_edge_attr.max().item(),test_edge_attr.min().item()}, Num nodes:
{num_users + num_items}")
```

```
Max and min train edge values: (9818, 0), max train edge attr:(5.0,
1.0), Num nodes: 17693
Max and min val edge values: (9815, 0), Max and min val attr:(5.0,
1.0), Num nodes: 17693
Max and min test edge values: (9817, 3), Max and min val attr:(5.0,
1.0), Num nodes: 17693
```

3. Model Development

```
class GCN(torch.nn.Module):
    def __init__(self, num_features, hidden_channels, num_items):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.fc = torch.nn.Linear(hidden_channels * 2, num_items)
        self.item_embeddings = torch.nn.Embedding(hidden_channels,
num_items)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        x = F.relu(x)

    # Apply the final linear layer on the concatenated edge features
        edge_pred = self.fc(torch.cat([x[edge_index[0]],
x[edge_index[1]]], dim=1))
        return edge_pred.squeeze()

    def recommend(self, user_embedding, k=2):
        item_scores = torch.matmul(user_embedding,
self.item_embeddings.weight.t())
        _, recommended_items = torch.topk(item_scores, k)
        return recommended_items

hidden_channels = 32
num_items = data['product_id'].nunique()
num_features=num_users + num_items

model = GCN(num_features=num_features,
hidden_channels=hidden_channels, num_items=num_items)
model

GCN(
  (conv1): GCNConv(17693, 32)
  (conv2): GCNConv(32, 32)
  (fc): Linear(in_features=64, out_features=7874, bias=True)
  (item_embeddings): Embedding(32, 7874)
)

# Prepare the data loader
train_loader = DataLoader([train_dataa], batch_size=1, shuffle=True)

criterion = torch.nn.MSELoss()
model.train()
total_loss = 0
```

```
with torch.no_grad():
    for batch in train_loader:
        val_out = model(batch)
        loss = criterion(val_out, batch.edge_attr.view(-1, 1))
        total_loss += loss.item()
print(f"total loss of validate data {total_loss}")
```

```
total loss of validate data 18.559938430786133
```

Training Loop

```
# Prepare the data loader
train_loader = DataLoader([train_dataa], batch_size=1, shuffle=True)

# Define the loss function and optimizer
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

total_loss = 0
patience = 7
num_epochs = 200
no_improvement = 0
best=0
model.train()

# Training loop
for epoch in range(num_epochs):

    # if loss.item() <= 0.2:
    if no_improvement >= patience:
            print(f'Early stopping at epoch {epoch+1} as validation
loss did not improve for {patience} consecutive epochs.')
            break
    else:
        for batch in train_loader:
            # zeroing gradients after each iteration
            optimizer.zero_grad()
            # making a pridiction in forward pass
            out = model(batch)
            # calculating the loss between original and predicted data
points
            loss = criterion(out, batch.edge_attr.view(-1, 1))
            # backward pass for computing the gradients of the loss
w.r.t to learnable parameters
            loss.backward()
            # updateing the parameters after each iteration
            optimizer.step()
            total_loss += loss.item()
```

```
            if int(loss.item()) != best:
                no_improvement = 0
            else:
                no_improvement += 1
            best=int(loss.item())
        print(f'Epoch {epoch + 1}, Loss: {loss.item():.4f}')
print(total_loss)
```

```
Epoch 1, Loss: 18.5599
Epoch 2, Loss: 18.4314
Epoch 3, Loss: 18.2114
Epoch 4, Loss: 17.8371
Epoch 5, Loss: 17.2549
Epoch 6, Loss: 16.4051
Epoch 7, Loss: 15.2329
Epoch 8, Loss: 13.6980
Epoch 9, Loss: 11.7858
Epoch 10, Loss: 9.5257
Epoch 11, Loss: 7.0306
Epoch 12, Loss: 4.5580
Epoch 13, Loss: 2.5859
Epoch 14, Loss: 1.8403
Epoch 15, Loss: 2.8047
Epoch 16, Loss: 4.2061
Epoch 17, Loss: 4.4488
Epoch 18, Loss: 3.6058
Epoch 19, Loss: 2.4260
Epoch 20, Loss: 1.4976
Epoch 21, Loss: 1.0398
Epoch 22, Loss: 0.9939
Epoch 23, Loss: 1.1850
Epoch 24, Loss: 1.4388
Epoch 25, Loss: 1.6355
Epoch 26, Loss: 1.7152
Epoch 27, Loss: 1.6650
Epoch 28, Loss: 1.5045
Epoch 29, Loss: 1.2751
Epoch 30, Loss: 1.0312
Early stopping at epoch 31 as validation loss did not improve for 7
consecutive epochs.
205.4300863146782
```

```
val_dataa.x[val_edge_index[0]]
```

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
```

```
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
```

validating Loop

```
# Prepare the data loader
val_loader = DataLoader([val_dataa], batch_size=1, shuffle=True)

model.eval()
total_loss = 0
with torch.no_grad():
    for batch in val_loader:
        val_out = model(batch)
        loss = criterion(val_out, batch.edge_attr.view(-1, 1))
        total_loss += loss.item()
print(f"total loss of validate data {total_loss}")
```

```
total loss of validate data 3.7362334728240967
```

```
# Define the loss function and optimizer
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
total_loss = 0
patience = 29
num_epochs = 200
no_improvement = 0
best=0
model.eval()

# validate loop
for epoch in range(num_epochs):
    # if loss.item() <= 0.1:
    if no_improvement >= patience:
            print(f'Early stopping at epoch {epoch+1} as validation 
loss did not improve for {patience} consecutive epochs.')
            break
    else:
        for batch in val_loader:
            optimizer.zero_grad()
            out = model(batch)
            loss = criterion(out, batch.edge_attr.view(-1, 1))
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

            if int(loss.item()) != best:
                no_improvement = 0
            else:
```

```python
                no_improvement += 1
            best=int(loss.item())
        print(f'Epoch {epoch + 1}, Loss: {loss.item():.4f}')
print(total_loss)
```

```
Epoch 1, Loss: 3.7362
Epoch 2, Loss: 3.0082
Epoch 3, Loss: 2.6233
Epoch 4, Loss: 2.1615
Epoch 5, Loss: 1.8206
Epoch 6, Loss: 1.6260
Epoch 7, Loss: 1.4471
Epoch 8, Loss: 1.2487
Epoch 9, Loss: 1.0945
Epoch 10, Loss: 1.0113
Epoch 11, Loss: 0.9359
Epoch 12, Loss: 0.8393
Epoch 13, Loss: 0.7641
Epoch 14, Loss: 0.7229
Epoch 15, Loss: 0.6803
Epoch 16, Loss: 0.6213
Epoch 17, Loss: 0.5697
Epoch 18, Loss: 0.5434
Epoch 19, Loss: 0.5227
Epoch 20, Loss: 0.4879
Epoch 21, Loss: 0.4541
Epoch 22, Loss: 0.4349
Epoch 23, Loss: 0.4184
Epoch 24, Loss: 0.3935
Epoch 25, Loss: 0.3694
Epoch 26, Loss: 0.3559
Epoch 27, Loss: 0.3441
Epoch 28, Loss: 0.3247
Epoch 29, Loss: 0.3060
Epoch 30, Loss: 0.2947
Epoch 31, Loss: 0.2829
Epoch 32, Loss: 0.2666
Epoch 33, Loss: 0.2534
Epoch 34, Loss: 0.2442
Epoch 35, Loss: 0.2327
Epoch 36, Loss: 0.2201
Epoch 37, Loss: 0.2118
Epoch 38, Loss: 0.2044
Epoch 39, Loss: 0.1944
Epoch 40, Loss: 0.1863
Early stopping at epoch 41 as validation loss did not improve for 29
consecutive epochs.
32.4571525901556
```

testing Loop

```
# Prepare the data loader
test_loader = DataLoader([test_dataa], batch_size=1, shuffle=True)

model.eval()
total_loss = 0
with torch.no_grad():
    for batch in test_loader:
        test_out = model(batch)
        loss = criterion(test_out, batch.edge_attr.view(-1, 1))
        total_loss += loss.item()
print(f"total loss of test data {total_loss}")
```

```
total loss of test data 3.1904726028442383
```

```
# Define the loss function and optimizer
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
total_loss = 0
patience = 13
num_epochs = 200
no_improvement = 0
best=0
model.eval()

# testing loop
for epoch in range(num_epochs):
    if no_improvement >= patience:
            print(f'Early stopping at epoch {epoch+1} as validation
loss did not improve for {patience} consecutive epochs.')
            break
    else:
        for batch in test_loader:
            optimizer.zero_grad()
            out = model(batch)
            loss = criterion(out, batch.edge_attr.view(-1, 1))
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

            if int(loss.item()) != best:
                no_improvement = 0
            else:
                no_improvement += 1
            best=int(loss.item())
        print(f'Epoch {epoch + 1}, Loss: {loss.item():.4f}')
print(total_loss)
```

```
Epoch 1, Loss: 3.1905
Epoch 2, Loss: 2.7598
Epoch 3, Loss: 2.3278
Epoch 4, Loss: 2.0663
Epoch 5, Loss: 1.8326
Epoch 6, Loss: 1.5941
Epoch 7, Loss: 1.4489
Epoch 8, Loss: 1.2959
Epoch 9, Loss: 1.1369
Epoch 10, Loss: 1.0333
Epoch 11, Loss: 0.9280
Epoch 12, Loss: 0.8124
Epoch 13, Loss: 0.7367
Epoch 14, Loss: 0.6621
Epoch 15, Loss: 0.5779
Epoch 16, Loss: 0.5247
Epoch 17, Loss: 0.4749
Epoch 18, Loss: 0.4203
Epoch 19, Loss: 0.3919
Epoch 20, Loss: 0.3597
Epoch 21, Loss: 0.3270
Epoch 22, Loss: 0.3109
Epoch 23, Loss: 0.2837
Epoch 24, Loss: 0.2629
Early stopping at epoch 25 as validation loss did not improve for 13
consecutive epochs.
25.75902023911476
```

4. Evaluation

```
val_dataa

Data(x=[17693, 17693], edge_index=[2, 1977], edge_attr=[1977])

model.eval()
with torch.no_grad():
    val_out = model(val_dataa)
    test_out = model(test_dataa)

# Calculate evaluation metrics
val_mse = mean_squared_error(val_edge_attr.numpy(), val_out[:,:1]
[:,0].numpy())
val_mae = mean_absolute_error(val_edge_attr.numpy(), val_out[:,:1]
[:,0].numpy())
val_R2_score= r2_score(val_edge_attr.numpy(), val_out[:,:1]
[:,0].numpy())

test_mse = mean_squared_error(test_edge_attr.numpy(), test_out[:,:1]
```

```
[:,0].numpy())
test_mae = mean_absolute_error(test_edge_attr.numpy(), test_out[:,:1]
[:,0].numpy())
test_R2_score= r2_score(test_edge_attr.numpy(), test_out[:,:1]
[:,0].numpy())


print(f'Validation MSE: {val_mse}, Validation MAE: {val_mae},
validation r2 score: {val_R2_score*100}')
print(f'Test MSE: {test_mse}, Test MAE: {test_mae}, Test r2 score:
{test_R2_score*100}')
```

```
Validation MSE: 1.0632094144821167, Validation MAE:
0.8342692255973816, validation r2 score: 36.85176372528076
Test MSE: 0.24447304010391235, Test MAE: 0.329874187707901, Test r2
score: 86.07448935508728
```

```
train_dataa
```

```
Data(edge_index=[2, 5931], edge_attr=[5931], x=[17693, 17693])
```

```
result_df['user_id']
```

```
0       A3554P57JKXVJN
1       A2CS4FEMSETJT1
2       A2CIJE4EUZF2WW
3       A15ZX3XV2L7QDH
4        AZGORQNAAGIMW
             ...
9880    A1GS1EX68K53ZS
9881    A32YO3HN3TPTNG
9882    A2K6OSG7JT8ZOA
9883    A3202C86H3AD1V
9884    A3LZB8E6ZU78XD
Name: user_id, Length: 9885, dtype: object
```

```
result_df['user_id'].unique()[4]
```

```
'AZGORQNAAGIMW'
```

```
result_df['user_id'].shape
```

```
(9885,)
```

Making predictions with `user_id` for the first `k` products

```
def make_recommendations(user_id, k=3):
    model.eval()
```

```
    with torch.no_grad():
        # Get all user embeddings (from the training graph)
        all_user_embeddings = model(train_dataa)
        # all_user_embeddings = model(train_graph_data.x,
train_graph_data.edge_index)

        # Extract the embedding for the specific user_id
        user_idx = user_encoder.transform([user_id])[0]

        user_embedding = all_user_embeddings[user_idx].unsqueeze(0)

        # Get recommendations (assuming model.recommend returns
indices)
        recommended_items = model.recommend(user_embedding, k=k)

        # Flatten the recommended_items array
        recommended_items = recommended_items.flatten()

        print("Raw recommended items (indices):",
recommended_items.cpu().numpy())

        # Convert to numpy array for processing
        recommended_item_indices = recommended_items.cpu().numpy()

        # Filter valid indices (should be within the range of item
indices)
        valid_indices = [idx for idx in recommended_item_indices if 0
<= idx < len(item_encoder.classes_)]

        print("Valid recommended items (indices):", valid_indices)

        if not valid_indices:
            print(f"No valid recommendations for user {user_id}")
            return []

        # Inverse transform and decode the recommended items
        recommended_items =
item_encoder.inverse_transform(valid_indices)

        return recommended_items


user_id = 'A2CIJE4EUZF2WW'
recommended_items = make_recommendations(user_id)
print(f'Recommended items for user {user_id}:', recommended_items)
```

```
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Recommended items for user A2CIJE4EUZF2WW: ['B00000K4BB' 'B000001OKH'
'B00001P584']
```

```
# Create a list to store user IDs and recommended items
user_recommendations = []
WWINF=[]

# Iterate over the first 10 unique users in the dataset
for user in result_df['user_id'].unique()[:100]:
    try:
        # Ensure the user is known
        user_idx = user_encoder.transform([user])
        # Make recommendations for the current user
        recommended_items = make_recommendations(user, k=3)
        # Append the user ID and recommended items to the list
        user_recommendations.append([user] + list(recommended_items))
    except IndexError:
        WWINF.append(user)
    except ValueError as e:
        print(f"Skipping user {user} due to: {e}")
    except KeyError as e:
        print(f"User {user} is not recognized by the encoder. Skipping
user.")

# Define the column names for the DataFrame
columns = ['user_id'] + [f'recommended_item_{i+1}' for i in range(3)]

# Create a DataFrame from the list of user recommendations
user_recommendations_df = pd.DataFrame(user_recommendations,
columns=columns)

# Save the DataFrame to a CSV file
user_recommendations_df.to_csv('user_recommendations.csv',
index=False)

# user_encoder.inverse_transform([850])[0]
# print(WWINF)
```

```
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
```

```
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
```

```
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
Raw recommended items (indices): [26  7 29]
Valid recommended items (indices): [26, 7, 29]
```

```python
user_encoder.inverse_transform([850])[0]
```

```
'A1BD5JY0P5E1P1'
```