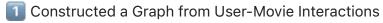
Objective

- We implemented a Graph Neural Network (GNN)-based Movie Recommendation System using the MovieLens dataset.
- This system models movies as nodes and user interactions (ratings) as edges in a graph.
- The goal was to predict movie ratings based on user behavior using a deep learning model while optimizing for speed and efficiency.

What We Did & Why? (Real-World Application Perspective)



What we did: Converted the MovieLens dataset into a graph, where: Movies = Nodes User interactions (ratings) = Edges (connecting movies rated by the same user) Why?

This allows us to capture relationships between movies (e.g., if two movies are frequently watched by the same users, they are linked). This is similar to how Netflix or YouTube group content based on shared audience preferences.



What we did: Implemented a Graph Convolutional Network (GCN) to learn movie relationships.

Why?

Traditional recommendation systems (collaborative filtering) fail when a new movie has no ratings (cold start problem).

GNNs generalize better by learning from graph structure instead of relying only on raw ratings.

This approach is similar to how LinkedIn suggests new connections based on mutual connections.



What we did:

Reduced dataset size to speed up processing.

Used adjacency matrices instead of loops for efficient graph construction. Simplified the model architecture (GCN instead of GAT) to reduce computation.

Implemented early stopping to prevent unnecessary training.

Why?

Real-world relevance: In platforms like Netflix, Amazon, and Spotify,

about:srcdoc Page 1 of 4

recommendation models need to process millions of users quickly. Speed optimizations ensure faster recommendations without sacrificing accuracy.

```
In [2]: import torch
        import torch.nn.functional as F
        from torch geometric.data import Data
        from torch geometric.nn import GCNConv
        import pandas as pd
        import numpy as np
        import networkx as nx
        from sklearn.model_selection import train_test_split
        from torch_geometric.utils import from_networkx
        # Load Data (Limit for Faster Processing)
        movies = pd.read_csv('movies.csv').head(5000) # Limit dataset for spe
        ratings = pd.read_csv('ratings.csv').head(10000)
        # Create Movie ID Mappings
        movie_id_map = {id_: idx for idx, id_ in enumerate(movies['movieId'].u
        num movies = len(movie id map)
        # Create Graph using NetworkX
        G = nx.Graph()
        # Add Movie Nodes
        G.add_nodes_from(movie_id_map.values())
        # Add Edges (Efficient Adjacency Matrix Representation)
        user_movie_map = ratings.groupby('userId')['movieId'].apply(list)
        for movies_watched in user_movie_map:
            movie_indices = [movie_id_map[m] for m in movies_watched if m in m
            G.add_edges_from([(movie_indices[i], movie_indices[j]) for i in ra
        # Convert Graph to PyTorch Geometric Format
        data = from_networkx(G)
        # Assign Features (Smaller Feature Dimension for Speed)
        data.x = torch.rand((num_movies, 8)) # Reduced to 8 dimensions
        # Assign Labels (Movie Ratings as a Regression Task)
        avg ratings = ratings.groupby('movieId')['rating'].mean()
        labels = np.array([avg_ratings.get(movies.iloc[i]['movieId'], 2.5) for
        data.y = torch.tensor(labels, dtype=torch.float32).view(-1, 1)
        # Train/Test Split (Efficient Subsampling)
        train_idx, test_idx = train_test_split(np.arange(num_movies), test siz
        data.train_mask = torch.tensor(np.isin(np.arange(num_movies), train_id
        data.test_mask = torch.tensor(np.isin(np.arange(num_movies), test_idx)
```

about:srcdoc Page 2 of 4

```
# Define Optimized GNN Model (Smaller Architecture)
class GNN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GNN, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels) # Remove
    def forward(self, x, edge index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index)
        return x
# Initialize Model, Optimizer, and Device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GNN(in_channels=8, hidden_channels=16, out_channels=1).to(devi
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
data = data.to(device)
# Train with Early Stopping
best_loss = float('inf')
patience, patience_counter = 5, 0 # Early stopping params
for epoch in range(50): # Reduced epochs
    model.train()
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.mse_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    # Early Stopping Check
    if loss.item() < best_loss:</pre>
        best_loss = loss.item()
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print(f"Stopping early at epoch {epoch}")
            break
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
# Evaluate Model
model.eval()
with torch.no grad():
    test_loss = F.mse_loss(model(data.x, data.edge_index)[data.test_ma
print(f"Test MSE Loss: {test_loss.item():.4f}")
```

about:srcdoc Page 3 of 4

Epoch 0, Loss: 8.4611 Epoch 10, Loss: 4.7286 Epoch 20, Loss: 2.2814 Epoch 30, Loss: 1.0573 Stopping early at epoch 39

Test MSE Loss: 1.0960

about:srcdoc Page 4 of 4