Movie-Recommendation-Website-using-LightGCN Documentation.

Project Overview

This project is a movie recommendation system built using Flask for the web interface and LightGCN (a graph-based collaborative filtering model) for generating personalized movie recommendations. The system allows users to:

1. **Get movie recommendations** based on their user ID and preferred genres.
2. **Rate movies**, which updates the model dynamically with the new user ratings.

Pipeline Overview

1. **Data Loading and Preprocessing**
   - **Loading Data (`load_data()`)**: Movie and user ratings data are loaded into pandas DataFrames (`movies_df` and `ratings_df`). The movie dataset contains information about movie IDs, titles, and genres. The ratings dataset contains user-movie interactions in the form of ratings.
   - **Data Preprocessing (`preprocess_data()`)**:
     - User and movie IDs are converted into a numeric format using label encoders (`le_user` and `le_item`) for efficient training.
     - The ratings data is split into training and testing sets.
     - The number of unique users (`n_users`) and items (`n_items`) is calculated to define the matrix size for the collaborative filtering model.
2. **Model Initialization**
   - **LightGCN Model (`LightGCN`)**:
     - The model is initialized with the number of users and items, the embedding size (latent factors = 64), and the number of graph layers (3).
     - The LightGCN model learns latent factors for users and items by leveraging graph convolutions on the user-item interaction graph. This is particularly useful for collaborative filtering tasks.
     - The pre-trained weights are loaded from `lightgcn_model.pth` to avoid retraining from scratch.
3. **Web Application (Flask)**
   - **Homepage (`index`)**:
     - The homepage displays a list of available genres (extracted from `movies_df`).
     - Users can select genres and submit a request for movie recommendations.
   - **Recommendation Process (`recommend`)**:
     - When a user requests recommendations, their user ID and selected genres are passed to the `get_top_recommendations()` function.
     - The function filters the movies based on the selected genres and uses the LightGCN model to generate a list of recommended movies for the given user.
     - The recommendations are then passed to the results page, where the user can view them.
   - **Rating Process (`rate`)**:

- The user can rate a specific movie. Once they submit the rating, the system validates the input (to ensure valid ratings and IDs).
- The rating is added to the `ratings_df` DataFrame, and the `update_model_with_new_rating()` function updates the user-item interaction matrix.
- The model is updated with this new rating, which can improve future recommendations.

4. **LightGCN Model Workflow**
   - **Training**: In the initial setup (outside the provided code), the LightGCN model would have been trained on the user-item interactions in the `ratings_df` DataFrame. It learns the latent representations (embeddings) for both users and items by performing multiple layers of graph convolutions on the user-item graph.
   - **Recommendation**: Once trained, the model computes the similarity between users and items based on their embeddings. For a given user ID, the model retrieves the items (movies) with the highest predicted interaction values (ratings), resulting in personalized recommendations.
5. **Model Update with New Ratings** (`update_model_with_new_rating`)
   - Whenever a user rates a new movie, this function:
     - Adds the new rating to the `ratings_df`.
     - Re-encodes the user and movie IDs if new users/movies are added.
     - Updates the LightGCN model's weights to account for the new interaction.
     - After the update, the model continues to provide personalized recommendations based on the latest user interactions.

Evaluation of recommendation systems:

Here's a comparison table for six methods of recommendation system evaluation:

Evaluation Metric/Aspect	Collaborative Filtering	Content-Based Filtering	Matrix Factorization	SVD	NGCF (Neural Graph Collaborative Filtering)	Hybrid Recommendation Systems
Approach	User-item interaction	Item features similarity	Latent factors decomposition	Decomposes matrix into factors	Uses graph neural networks for recommendations	Combines multiple methods (e.g., Collaborative + Content)
Data Dependency	Requires user-item interaction data (ratings, clicks)	Requires item features (tags, keywords)	User-item interaction matrix	Interaction matrix decomposition	Requires interaction data + graph structure	Requires both interaction data and item features
Cold Start Problem	Yes (for new users/items)	Yes (for new items)	Less pronounced with latent factors	Reduced for unseen users/items with latent factor mapping	Can mitigate using graph structure	Mitigated (depends on method combination)
Scalability	Moderate, struggles with large datasets	Scales well with item features	Better scalability using latent factor models	Moderate	High scalability with large datasets	Depends on combined methods
Accuracy	High with sufficient data	Moderate, depends on feature quality	High with properly tuned models	High accuracy if properly factorized	High with structured graph representations	Generally higher accuracy by combining methods

Explainability	Hard to explain recommendations	Easier to explain based on item features	Harder to interpret	Difficult due to complex decompositions	Difficult due to complex neural graph structure	Depends on method used for explanation
Sparsity Handling	Struggles with sparse data	Works well with feature-rich items	Handles sparsity well	Effective for sparse matrices	Effective in sparse environments	Handles sparsity by combining methods
Personalization	High, based on user similarity	High, based on item preference	High personalization based on latent factors	High, personalized by decomposed factors	Highly personalized through graph structures	Very high personalization through hybrid combinations
Computation Complexity	Moderate	Low	Moderate to High	Moderate	High (due to GNN computation)	Varies depending on combined models
Examples	Netflix, Amazon	Pandora (Music Recommendation)	Google Matrix Factorization	Singular Value Decomposition (SVD) in recommendation systems	Graph-based recommendation platforms	Netflix hybrid approach combining collaborative filtering and content-based

Detailed Pipeline Steps

1. **ta Loading**:
   - The data is loaded at the beginning of the Flask app, calling `load_data()` which loads movie metadata and user ratings into DataFrames.
2. **Data Preprocessing**:
   - The user and movie IDs are converted into numeric IDs using label encoders.
   - The ratings data is split into training and testing sets.
3. **Model Loading**:
   - A pre-trained LightGCN model is loaded from the `lightgcn_model.pth` file and is set to evaluation mode (`model.eval()`), meaning it's ready to generate recommendations without being retrained.
4. **User Requests Recommendations**:
   - On the homepage (`/`), the user enters their user ID and selects genres.
   - When they submit the form, the `recommend` route processes this request:
     - The selected genres are filtered from the `movies_df` DataFrame.
     - The `get_top_recommendations()` function is called with the user ID and filtered genres to generate the top movie recommendations using the LightGCN model.
     - These recommendations are passed to the results template (`results.html`) and displayed to the user.
5. **User Rates a Movie**:
   - A user submits a rating for a specific movie through the `/rate` route.
   - The input is validated (to ensure the rating is between 1 and 5, and that IDs are valid).
   - The `update_model_with_new_rating()` function adds the new rating to the `ratings_df` DataFrame.
   - The function then updates the model, allowing it to learn from the new interaction. The updated model is ready for future recommendations.

Core Components

1. **DataFrames**:
   - `movies_df`: Contains movie metadata, such as titles, genres, etc.
   - `ratings_df`: Stores user-item interactions (ratings), which are updated as new ratings are submitted by users.
2. **LightGCN Model**:
   - A graph-based recommendation model that performs well in collaborative filtering tasks by learning user and item embeddings.
   - It generates personalized recommendations by finding items that match a user's preferences based on their historical interactions.
3. **Flask App**:
   - Provides a user-friendly interface where users can request movie recommendations based on their preferences and submit new ratings.
   - The app dynamically updates the model as users submit new ratings, ensuring the system adapts to user feedback over time.

Flow Diagram (Simplified)

1. **User Inputs** (Homepage):
   - User ID
   - Selected Genres
2. **LightGCN Model**:
   - Retrieves and preprocesses the user-movie interaction matrix.
   - Generates top recommendations based on selected genres and user history.
3. **Flask App**:
   - Displays recommendations.
   - Allows users to submit new ratings.
4. **Model Update**:
   - The model updates itself after receiving new ratings, improving future recommendations.

Summary

- **Input**: User ID and genres for recommendations; movie ID and rating for rating.
- **Process**: The LightGCN model generates recommendations and updates based on new ratings.
- **Output**: Personalized movie recommendations for users.

This project allows users to interact with a recommendation system where the model is continually updated based on user feedback (new ratings), ensuring more accurate recommendations over time.

model.py file:

1. Imports

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import numpy as np
```

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing as pp
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import degree
import random
from tqdm import tqdm
```

- `torch`: The PyTorch library, used for tensor computations and creating neural networks.
- `torch.nn`: Provides basic building blocks for creating neural networks, such as layers and loss functions.
- `torch.nn.functional`: Contains functions for different activation layers and other functions for model building.
- `pandas` **(pd)**: A library for data manipulation and analysis.
- `numpy` **(np)**: A library for numerical computations.
- `sklearn`: The Scikit-learn library, used for machine learning operations like data preprocessing and splitting datasets.
- `torch_geometric`: A library designed for deep learning on graphs and other irregularly structured data.
- `tqdm`: A library for progress bars in loops, particularly useful when training machine learning models.

Device Initialization:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

- **Purpose**: Initializes the device for computations, checking if CUDA-enabled GPUs are available. If not, it defaults to using the CPU.

`LightGCNConv` Class:

This class defines the LightGCN convolutional layer, a key component for aggregating graph-based information.

```
class LightGCNConv(MessagePassing):
    def __init__(self, **kwargs):
        super().__init__(aggr="add")
```

- **Purpose**: Initializes a message-passing layer for graph data, aggregating messages by summing them (`aggr="add"`).
- **Message Passing**: This method propagates node features across edges and aggregates them.

`forward` Method:

This method computes the forward pass for a LightGCN layer by normalizing node features.

```
def forward(self, x, edge_index):
    from_, to_ = edge_index
    deg = degree(to_, x.size(0), dtype=x.dtype)
    deg_inv_sqrt = deg.pow(-0.5)
    deg_inv_sqrt[deg_inv_sqrt == float("inf")] = 0
    norm = deg_inv_sqrt[from_] * deg_inv_sqrt[to_]

    return self.propagate(edge_index, x=x, norm=norm)
```

- **Parameters**:

- `x`: Node feature matrix.
- `edge_index`: Graph connectivity in COO format, representing edges.

- **Degree Normalization**: It calculates the degree of each node and then normalizes the messages passed to neighboring nodes by their degrees (inverse square root).
- **Propagation**: Uses the `propagate` method to pass messages between nodes based on edges.

`message` Method:

Defines how messages are passed during graph convolutions.

```
def message(self, x_j, norm):
    return norm.view(-1, 1) * x_j
```

- **Purpose**: Multiplies the node features by the normalized values (`norm`) during the message-passing phase.
- **Parameters**:
  - `x_j`: Neighbor node features.
  - `norm`: Normalization factor calculated in the `forward` pass.

`LightGCN` Class:

Defines the full LightGCN model, which uses multiple convolutional layers to learn embeddings.

```
class LightGCN(nn.Module):
    def __init__(self, num_users, num_items, latent_dim, num_layers):
        super(LightGCN, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.latent_dim = latent_dim
        self.num_layers = num_layers

        self.embedding = nn.Embedding(num_users + num_items, latent_dim)
        self.convs = nn.ModuleList(LightGCNConv() for _ in range(num_layers))

        self.init_parameters()
```

- **Parameters**:
  - `num_users`: The number of users.
  - `num_items`: The number of items.
  - `latent_dim`: The dimensionality of the embedding space for users and items.
  - `num_layers`: The number of layers for the graph convolutions.
- **Embeddings**:
  - Embeds both users and items in a common latent space (`nn.Embedding`).
  - The number of embeddings is the sum of users and items.
- **Convs**: A list of `LightGCNConv` layers, where the number of layers is controlled by `num_layers`.

`init_parameters` Method:

Initializes the embeddings' weights using a normal distribution.

```
def init_parameters(self):
    nn.init.normal_(self.embedding.weight, std=0.1)
```

- **Purpose**: Initializes the embedding weights with a standard deviation of 0.1.

`forward` Method:

Defines the forward pass of the LightGCN model.

```
def forward(self, edge_index):
    emb0 = self.embedding.weight
    embs = [emb0]

    emb = emb0
    for conv in self.convs:
        emb = conv(x=emb, edge_index=edge_index)
        embs.append(emb)

    out = torch.mean(torch.stack(embs, dim=0), dim=0)

    return emb0, out
```

- **Parameters**:
  - `edge_index`: The adjacency matrix in COO format representing the graph structure.
- **Embeddings**:
  - `emb0`: The initial embedding.
  - `embs`: A list of embeddings for each layer.
  - Each `LightGCNConv` layer updates the embedding, and the results are averaged across layers to get the final embedding `out`.

`encode_minibatch` Method:

Encodes users and items for a specific mini-batch.

```
def encode_minibatch(self, users, pos_items, neg_items, edge_index):
    emb0, out = self(edge_index)
    return (
        out[users],
        out[pos_items],
        out[neg_items],
        emb0[users],
        emb0[pos_items],
        emb0[neg_items],
    )
```

- **Parameters**:
  - `users`: Batch of users.

- `pos_items`: Positive items interacted with by the users.
- `neg_items`: Negative items not interacted with by the users.
- `edge_index`: The adjacency matrix for the graph.

- **Returns**: Encoded user and item embeddings for both the final layer (`out`) and the initial embedding layer (`emb0`).

`compute_bpr_loss` Function:

Calculates the Bayesian Personalized Ranking (BPR) loss for the model.

```
def compute_bpr_loss(users, users_emb, pos_emb, neg_emb, user_emb0, pos_emb0, neg_emb0):
    reg_loss = (
        (1 / 2)
        * (user_emb0.norm().pow(2) + pos_emb0.norm().pow(2) + neg_emb0.norm().pow(2))
        / float(len(users))
    )

    pos_scores = torch.mul(users_emb, pos_emb).sum(dim=1)
    neg_scores = torch.mul(users_emb, neg_emb).sum(dim=1)

    bpr_loss = torch.mean(F.softplus(neg_scores - pos_scores))

    return bpr_loss, reg_loss
```

- **Purpose**:
  - The BPR loss function encourages the model to rank positive interactions higher than negative ones.
  - `reg_loss`: L2 regularization on embeddings to prevent overfitting.
  - `pos_scores` and `neg_scores`: Dot products between user embeddings and positive/negative item embeddings.
  - `bpr_loss`: Softplus is applied to the difference between negative and positive scores to calculate loss.

`data_loader` Function:

Generates batches of users, positive items, and negative items for training.

```
def data_loader(data, batch_size, n_usr, n_itm):
    ...
```

- **Purpose**: Samples a batch of users, their positive interactions, and random negative items (non-interacted items) for training.
- **Negative Sampling**: Ensures that the negative items selected for each user are not ones they have interacted with.

`train` Function:

This function handles the entire training process for the LightGCN model.

```
def train(model, optimizer, data, edge_index, batch_size, n_usr, n_itm):
    model.train()
    total_loss = 0

    for users, pos_items, neg_items in data_loader(data, batch_size, n_usr, n_itm):
        optimizer.zero_grad()
```

```
        users_emb, pos_emb, neg_emb, user_emb0, pos_emb0, neg_emb0 = model.encode_minibatc
h(
            users, pos_items, neg_items, edge_index
        )
        loss, reg_loss = compute_bpr_loss(
            users, users_emb, pos_emb, neg_emb, user_emb0, pos_emb0, neg_emb0
        )
        loss += reg_loss * 1e-4

        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    return total_loss / len(data)
```

- **Purpose**: Trains the model for one epoch.
- **Parameters**:
  - `model`: The LightGCN model to be trained.
  - `optimizer`: Optimizer used for adjusting model weights (e.g., Adam).
  - `data`: The training data (user-item interaction pairs).
  - `edge_index`: The graph structure for users and items.
  - `batch_size`: Number of samples processed in one iteration.
  - `n_usr` and `n_itm`: Number of users and items respectively.
- **Steps**:
  - Sets the model to training mode (`model.train()`).
  - For each batch of users, positive items, and negative items (generated by `data_loader`):
    - `optimizer.zero_grad()`: Clears the gradients from the previous batch.
    - `encode_minibatch`: Gets embeddings for users, positive items, and negative items.
    - `compute_bpr_loss`: Computes the BPR loss and regularization.
    - `loss.backward()`: Backpropagates the loss to compute gradients.
    - `optimizer.step()`: Updates the model's parameters using the computed gradients.
    - Accumulates the loss across all batches.
- **Returns**: The average training loss for the entire epoch.

`test` Function:

Evaluates the performance of the model using Hit Rate (HR) and Normalized Discounted Cumulative Gain (NDCG).

```
def test(model, edge_index, test_data, train_user_dict, test_user_dict, top_k=10):
    model.eval()
    hr, ndcg = [], []

    emb0, out = model(edge_index)
    users_list = list(test_user_dict.keys())
```

```
    for user in users_list:
        seen = set(train_user_dict[user])

        predictions = torch.matmul(out[user], out.t()).cpu().detach().numpy()
        predictions[seen] = -np.inf

        rank = np.argpartition(predictions, -top_k)[-top_k:]
        rank = rank[np.argsort(predictions[rank])[::-1]]

        hr.append(np.isin(test_user_dict[user], rank))
        if len(test_user_dict[user]) == 0:
            continue

        ndcg.append(
            np.reciprocal(np.log2(np.where(rank == test_user_dict[user][0])[0] + 2))
            if test_user_dict[user][0] in rank
            else 0
        )

    return np.mean(hr), np.mean(ndcg)
```

- **Purpose**: Tests the model and computes performance metrics for recommendations.
- **Parameters**:
  - `model`: The trained LightGCN model.
  - `edge_index`: The graph structure (edges) for users and items.
  - `test_data`: The testing dataset (user-item interactions).
  - `train_user_dict`: Dictionary where each user maps to the items they interacted with in the training data.
  - `test_user_dict`: Dictionary where each user maps to the items they interacted with in the test data.
  - `top_k`: Number of top recommendations to evaluate.
- **Steps**:
  - Puts the model into evaluation mode (`model.eval()`).
  - **Embedding Calculation**: Retrieves user and item embeddings from the model.
  - **For Each User**:
    - `seen`: Items the user has already interacted with in the training set are ignored in predictions.
    - **Predictions**: Computes scores for all items by taking the dot product of the user embedding and all item embeddings.
    - **Top-K Ranking**: Sorts the predicted item scores and selects the top `k` items.
    - **Hit Rate (HR)**: Checks if any of the user's test items are in the top `k` predicted items.
    - **NDCG**: Measures the ranking quality of the predicted items by giving higher weights to the correctly predicted items in higher positions.
- **Returns**: The average **Hit Rate (HR)** and **NDCG** for all users.

`data_loader` Helper Function:

Generates mini-batches of users, positive items (interacted), and negative items (not interacted).

```
def data_loader(data, batch_size, n_usr, n_itm):
    all_pos = {}
    for u, i in data:
        if u not in all_pos:
            all_pos[u] = []
        all_pos[u].append(i)

    data = np.array(list(data))

    for start in range(0, len(data), batch_size):
        batch = data[start : start + batch_size]
        users = batch[:, 0]
        pos_items = batch[:, 1]
        neg_items = []

        for u in users:
            while True:
                neg_item = np.random.randint(n_itm)
                if neg_item not in all_pos[u]:
                    neg_items.append(neg_item)
                    break

        yield torch.LongTensor(users), torch.LongTensor(pos_items), torch.LongTensor(neg_i
tems)
```

- **Purpose**: Prepares data batches for training by randomly sampling negative items for users that they haven't interacted with.
- **Parameters**:
  - `data`: List of user-item interaction pairs.
  - `batch_size`: Number of samples per batch.
  - `n_usr`: Total number of users.
  - `n_itm`: Total number of items.
- **Steps**:
  - Creates a dictionary, `all_pos`, to store all positive items (interacted items) for each user.
  - For each batch:
    - Extracts **users** and their **positive items**.
    - For each user, randomly selects a **negative item** that they have not interacted with.
    - Yields the batch of users, positive items, and negative items as tensors.

Helper Functions:

`negative_sampling` Function:

Performs negative sampling by selecting items that the user hasn't interacted with.

```
def negative_sampling(pos_data, num_users, num_items, num_negatives):
    negative_data = []

    all_pos = {u: set() for u in range(num_users)}
    for u, i in pos_data:
        all_pos[u].add(i)

    for u, pos_items in all_pos.items():
        for _ in range(num_negatives):
            while True:
                neg_item = np.random.randint(num_items)
                if neg_item not in pos_items:
                    negative_data.append([u, neg_item])
                    break

    return negative_data
```

- **Purpose**: For each user, generates negative samples by randomly selecting items the user has not interacted with.
- **Parameters**:
  - `pos_data`: Positive user-item interaction pairs.
  - `num_users`: Total number of users.
  - `num_items`: Total number of items.
  - `num_negatives`: Number of negative samples to generate per user.
- **Steps**:
  - `all_pos`: Stores all items the user has interacted with.
  - For each user, it randomly selects a negative item until the required number of negative samples is reached.
- **Returns**: A list of negative user-item pairs.

---

app.py file:

The provided `app.py` file uses Flask to create a web interface for a LightGCN-based movie recommendation system. Here's an overview of the key functionality and how the app is structured:

Key Features

1. **Homepage (`/`)**
   - Loads the available genres from the `movies_df` DataFrame.
   - Displays these genres on the homepage using the `index.html` template.
2. **Recommend Movies (`/recommend`)**
   - A user submits a form with their user ID and selected genres.
   - The app fetches top movie recommendations for the user, filtering by genres.
   - The results are displayed using the `results.html` template.
3. **Rate Movies (`/rate`)**
   - A user can submit a form to rate a specific movie.

- The app updates the LightGCN model with the new user rating and re-trains the model.
- A success message is shown to the user.

Breakdown of the Code

1. Imports and Setup

```
from flask import Flask, render_template, request, redirect, url_for, flash
import pandas as pd
import torch
from model import (
    load_data,
    LightGCN,
    preprocess_data,
    get_top_recommendations,
    update_model_with_new_rating,
)
```

- **Flask**: A web framework for creating web applications.
- **pandas**: A library for data manipulation and analysis.
- **torch**: A library for deep learning (part of PyTorch).
- **model**: This imports specific functions and classes from a `model.py` file that likely contains the logic for the LightGCN model and data handling.

2. Application Initialization

```
app = Flask(__name__)
app.secret_key = "your_secret_key"
```

- `app = Flask(__name__)`: Initializes the Flask application.
- `app.secret_key`: A secret key for session management and security (like protecting against CSRF attacks).

3. Global Model Declaration

```
global model
```

- Declares a global variable `model` to store the LightGCN model instance.

4. Load Model and Data

```
movies_df, ratings_df = load_data()
train_df, test_df, n_users, n_items, le_user, le_item = preprocess_data(ratings_df)
model = LightGCN(n_users, n_items, latent_dim=64, num_layers=3).to("cpu")
model.load_state_dict(torch.load("lightgcn_model.pth", map_location="cpu"))
model.eval()
```

- **Loading Data**: Calls `load_data()` to load movie and ratings data into `movies_df` and `ratings_df` DataFrames.
- **Preprocessing**: Calls `preprocess_data()` to prepare the data for training, returning:
  - `train_df`: Training DataFrame.

- `test_df`: Testing DataFrame.
- `n_users`: Total number of users.
- `n_items`: Total number of items.
- `le_user` and `le_item`: Label encoders for users and items.

- **Model Initialization**: Creates an instance of the `LightGCN` model with specified parameters (64 latent dimensions and 3 layers) and moves it to the CPU.
- **Load Model Weights**: Loads the pre-trained model weights from a file (`lightgcn_model.pth`) and sets the model to evaluation mode (`model.eval()`).

5. Index Route

```
@app.route("/")
def index():
    available_genres = list(set("|".join(movies_df["genres"]).split("|")))
    return render_template("index.html", genres=available_genres)
```

- **Route**: Maps the root URL (`/`) to the `index` function.
- **Available Genres**: Extracts unique genres from the `movies_df` DataFrame and prepares them for rendering in the template.
- **Render Template**: Renders the `index.html` template, passing the list of available genres.

6. Recommend Route

```
@app.route("/recommend", methods=["POST"])
def recommend():
    user_id = request.form.get("user_id")
    selected_genres = request.form.get("genres").split(",")

    recommendations, _ = get_top_recommendations(
        int(user_id),
        selected_genres,
        model,
        movies_df,
        ratings_df,
        le_user,
        le_item,
        n_users,
    )

    print(recommendations.to_dict(orient="records"))  # Add this line

    return render_template(
        "results.html", recommendations=recommendations.to_dict(orient="records")
    )
```

- **Route**: Maps the `/recommend` URL to the `recommend` function for POST requests.
- **Get User ID and Genres**: Retrieves the user ID and selected genres from the submitted form.

- **Get Recommendations**: Calls `get_top_recommendations()` to get movie recommendations based on the user ID and selected genres.
- **Debug Output**: Prints the recommendations to the console (useful for debugging).
- **Render Results**: Renders the `results.html` template, passing the recommendations as a list of dictionaries.

7. Rate Route

```
@app.route("/rate", methods=["POST"])
def rate():
    global model, ratings_df, n_users  # Declare model, ratings_df, and n_users as global
    user_id = request.form.get("user_id")
    movie_id = request.form.get("movie_id")
    rating = request.form.get("rating")

    # Validate inputs
    if not user_id.isdigit() or not movie_id.isdigit() or not (1 <= float(rating) <= 5):
        flash("Invalid input values!", "error")
        return redirect(url_for("index"))

    print(f"user_id: {user_id}, movie_id: {movie_id}, rating: {rating}")  # Debug output

    try:
        model, ratings_df, n_users = update_model_with_new_rating(
            model,
            ratings_df,
            int(user_id),
            int(movie_id),
            float(rating),
            n_users,
            le_user,
            le_item,
        )
        flash("Rating added successfully!", "success")
    except Exception as e:
        flash(f"An error occurred: {str(e)}", "error")  # Capture the error message

    return redirect(url_for("index"))
```

- **Route**: Maps the `/rate` URL to the `rate` function for POST requests.
- **Global Variables**: Declares `model`, `ratings_df`, and `n_users` as global to modify them.
- **Get Inputs**: Retrieves user ID, movie ID, and rating from the form.
- **Input Validation**: Checks if the user ID and movie ID are digits and if the rating is between 1 and 5. If not, it shows an error message.
- **Debug Output**: Prints the input values to the console.
- **Update Model**: Calls `update_model_with_new_rating()` to update the model with the new rating.
  - If successful, it flashes a success message; if there's an error, it flashes the error message.
- **Redirect**: Redirects back to the index page.

8. Running the Application

```
if __name__ == "__main__":
    app.run(debug=True)
```

- **Run the App**: If the script is executed directly, the Flask application will start in debug mode, allowing for live code updates and error messages in the browser.

Summary

- **Web Framework**: Uses Flask to handle web requests and serve HTML templates.
- **Model Loading**: Loads and prepares a LightGCN model for movie recommendations.
- **Routes**: Defines several routes to display the main page, handle recommendations, and process user ratings.
- **Data Handling**: Uses pandas for data manipulation and PyTorch for model operations.

This application structure allows users to input their preferences, receive movie recommendations, and rate movies, effectively interacting with a recommendation model.

What Happens in the Helper Functions

1. `load_data()`: Loads movie and rating data (likely from CSV files or a database).
2. `preprocess_data()`: Prepares the data for training the LightGCN model by encoding user and item IDs and splitting into training and testing sets.
3. `get_top_recommendations()`: Uses the trained LightGCN model to generate movie recommendations for a user based on their interaction history and selected genres.
4. `update_model_with_new_rating()`: Adds a new rating to the dataset, updates the user-item matrix, and retrains or fine-tunes the model with the new data.