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.

Detailed Pipeline Steps

1. **Data 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:

Importing Libraries

```
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**: Main PyTorch library for building and training neural networks.
- **torch.nn**: Module containing neural network layers and operations.
- **torch.nn.functional**: Provides functions like activation and loss functions that are stateless (not part of `nn.Module`).
- **pandas**: For data manipulation and analysis, specifically working with DataFrames.
- **numpy**: For numerical computing with arrays.
- **sklearn.model_selection.train_test_split**: Utility to split data into training and testing sets.
- **sklearn.preprocessing as pp**: Preprocessing utilities, like `LabelEncoder` to convert labels to numerical indices.
- **torch_geometric.nn.conv.MessagePassing**: A message-passing framework used in graph neural networks (GNNs).
- **torch_geometric.utils.degree**: Utility to calculate the degree (number of connections) of nodes in a graph.
- **random**: Standard Python library for random operations.

- **tqdm**: Progress bar library used to track the execution of loops.

Device Configuration

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

- Checks if a GPU (CUDA) is available. If so, sets the device to GPU (`cuda:0`). If not, it defaults to CPU (`cpu`).

LightGCNConv Class

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

- This defines a graph convolution layer, `LightGCNConv`, extending from `MessagePassing`.
- `super().__init__(aggr="add")`: Calls the constructor of the parent class (`MessagePassing`), using an aggregation method `"add"`. This means messages passed between nodes will be summed.

```
    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)
```

- **`forward`**: Defines how this layer processes inputs.
- **`edge_index`**: The connections between nodes (edges) represented as a two-row matrix (from/to nodes).

- `from_, to_ = edge_index`: Splits edge connections into two lists of node indices.
- `deg`: Degree of each node (how many edges connect to it).
- `deg_inv_sqrt`: The inverse square root of the degree. This normalization ensures that nodes with higher degrees don't overpower nodes with fewer connections.
- `norm`: A normalization factor applied to the message-passing update.
- `self.propagate`: Executes message passing between nodes, using the `norm` and node features `x`.

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

- `message`: The core of message passing. Each node sends its features to its neighbors, scaled by the normalization factor `norm`.

LightGCN Class

```
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 ra
nge(num_layers))

        self.init_parameters()
```

- `LightGCN`: Defines the Light Graph Convolutional Network model.
- `__init__`: Constructor that initializes the network's components.
    - `num_users`: Number of users in the dataset.
    - `num_items`: Number of items (e.g., movies) in the dataset.
    - `latent_dim`: Dimensionality of the user/item embeddings.
    - `num_layers`: Number of layers (how many times message passing is performed).
- `self.embedding`: An embedding layer that holds latent vectors for both users and items (combined size is `num_users + num_items`).
- `self.convs`: A list of `LightGCNConv` layers, repeated `num_layers` times.

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

- `init_parameters`: Initializes the embedding weights using a normal distribution with a standard deviation of `0.1`.

Forward Pass in LightGCN

```
    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
```

- `forward`: Defines how the embeddings are updated through each layer of the network.
    - `emb0 = self.embedding.weight`: Initial embeddings (before message passing).
    - `for conv in self.convs`: Iterates through each `LightGCNConv` layer, updating the embeddings via message passing.
    - `out = torch.mean(torch.stack(embs, dim=0), dim=0)`: Averages the embeddings over all layers to get the final output.
    - **Returns** both the initial embeddings and the output embeddings after propagation.

Encode Minibatch in LightGCN

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

- `encode_minibatch`: Encodes a batch of users and items.
    - `self(edge_index)`: Calls the forward pass to obtain updated embeddings.
    - `out[users], out[pos_items], out[neg_items]`: Fetches the embeddings for the specific users, positive items (items users interacted with), and negative items (items users did not interact with).

BPR Loss Function

```
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_score
s))

    return bpr_loss, reg_loss
```

- `compute_bpr_loss` : Calculates the Bayesian Personalized Ranking (BPR) loss used in recommendation systems.
    - `reg_loss` : Regularization loss that penalizes large embedding values (L2 regularization).
    - `pos_scores` / `neg_scores` : Calculates the dot product between user embeddings and positive/negative item embeddings.
    - `bpr_loss` : Measures the ranking loss by ensuring that the positive item score is higher than the negative item score.
    - Returns both the BPR loss and regularization loss.

Data Loader

```
def data_loader(data, batch_size, n_usr, n_itm):
    def sample_neg(x):
        while True:
            neg_id = random.randint(0, n_itm - 1)
            if neg_id not in x:
                return neg_id

    interected_items_df = (
        data.groupby("user_id_idx")["item_id_idx"].apply(lis
t).reset_index()
    )
    indices = [x for x in range(n_usr)]

    if n_usr < batch_size:
        users = [random.choice(indices) for _ in range(batch_
size)]
    else:
        users = random.sample(indices, batch_size)
    users.sort()
    users_df = pd.DataFrame(users, columns=["users"])

    interected_items_df = pd.merge(
        interected_items_df,
        users_df,
        how="right",
        left_on="user_id_idx",
        right_on="users",
    )
    pos_items = (
        interected_items_df["item_id_idx"]
        .apply(lambda x: random.choice(x) if isinstance(x, li
st) else x)
        .values


    )
    neg_items = interected_items_df["item_id_idx"].apply(samp
le_neg).values
```

```
    return torch.tensor(users), torch.tensor(pos_items), torc
h.tensor(neg_items)
```

- `data_loader`: Generates a batch of users, positive items, and negative items for training.
    - `sample_neg`: Helper function to sample a random item that the user has not interacted with.
    - `interected_items_df`: Groups the data by `user_id` and gathers the list of items each user has interacted with.
    - `random.sample` / `random.choice`: Randomly selects users and items for each batch.
    - Converts the users, positive items, and negative items into `torch.tensor` objects, which can be used for PyTorch training.

Evaluation Metrics

```
def eval_model(model, edge_index, users, test_df, k):
    item_embs = model(edge_index)[1][model.num_users:]
    user_embs = model(edge_index)[1][: model.num_users]

    train_user_item_dict = {
        row[0]: set(row[1:]) for row in test_df[["user_id_id
x", "item_id_idx"]].values
    }

    users_batch = torch.tensor(users).to(device)
    rating = torch.matmul(user_embs[users_batch], item_embs.
T)

    return calculate_metrics(users, rating, train_user_item_d
ict, k)
```

- `eval_model`: Evaluates the trained LightGCN model by calculating its performance on top-K recommendations.
    - `item_embs`: Fetches the learned item embeddings.
    - `user_embs`: Fetches the learned user embeddings.
    - `train_user_item_dict`: A dictionary mapping users to their interacted items.
    - `torch.matmul(user_embs[users_batch], item_embs.T)`: Multiplies user embeddings with item embeddings to compute a similarity score (rating).
    - `calculate_metrics`: Uses this rating matrix to compute evaluation metrics like recall, precision, etc.

---

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. Global Variables

- `model`: The LightGCN model is loaded globally at the start of the app using the trained weights (`lightgcn_model.pth`).
- `movies_df` and `ratings_df`: DataFrames storing the movie metadata and user ratings.
- Label encoders (`le_user`, `le_item`) and the number of users and items are initialized to preprocess the input data.

2. Homepage Route (`/`)

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

- Fetches a unique list of genres from the `movies_df`.
- Passes the list of genres to the `index.html` template, where users can select genres for movie recommendations.

3. Recommendation Route (`/recommend`)

```
@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,
    )
    return render_template(
        "results.html", recommendations=recommendations.to_di
ct(orient="records")
    )
```

- Takes the `user_id` and `genres` from the form submission.
- Calls the `get_top_recommendations` function to fetch the top recommendations for the user.
- Displays the recommendations in the `results.html` template, passing them as a list of records.

4. Rating Route (`/rate`)

```
@app.route("/rate", methods=["POST"])
def rate():
    global model  # Declare model as global to access it
    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"))
    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")
    return redirect(url_for("index"))
```

- Takes the `user_id`, `movie_id`, and `rating` values from the form submission.
- Validates that the inputs are valid (user and movie IDs must be digits, and the rating must be between 1 and 5).
- Calls the `update_model_with_new_rating` function to update the LightGCN model with the new rating.
- After updating the model and saving the new state, it flashes a success message and redirects the user back to the homepage.

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.

Notes:

- **Model Persistence**: Every time a new rating is added, the model is updated in memory. However, the model is not saved again to disk (unless explicitly done in `update_model_with_new_rating()`).
- **Templates**: We will need corresponding HTML templates (`index.html` and `results.html`) to render the user interface for selecting genres and showing the recommendations.
- **Error Handling**: Basic validation for user and movie IDs and the rating value ensures invalid input doesn't crash the app.