Recommendation System - Setup Guide

This guide walks you through building a hybrid recommendation system using the LightFM library and MongoDB. The system blends collaborative filtering and content-based filtering techniques and is deployed through a Flask backend. It is designed to provide accurate, personalized product recommendations and is easily extendable for web applications.

Recommendation Techniques

- Collaborative filtering makes recommendations based on user behavior. It identifies patterns among users who have interacted with similar items, if users with similar preferences in the past will like similar items in the future.
- Content-Based Filtering recommends items by comparing the features of items and the preferences or characteristics of users. It helps with the 'cold start' problem (new users or new products) by relying on metadata such as user location or product category.

Data Requirements

To build an effective recommendation system, the following data volumes are ideal:

Type	Target Volume
Users	≥ 1,000 (Ideal: 5K–50K)
Products	≥ 500 (Ideal: 1K–10K)
Interactions	≥ 10× users (Ideal: 100K+)

However, for a functional demo, the minimum required data is:

Type	Target Volume
Users	~ 100
Products	~ 100
Interactions	~ 1000+ (ideally 5–10 interactions per user)

To get started with building a Recommendation System

1. Uploading Data to MongoDB

Prepare your sample data as CSV files. Use the provided script \`insert_csv_data_into_mongodb.py\` to upload the data to MongoDB. This script currently handles the following collections:

- users
- products
- user_product_interactions

2. Recommendation Model

To get started open the script `recommendation_system.py`. We will be using LightFM to create a recommendation mode.

1. Load Data from MongoDB:
    a. Establish a connection to the MongoDB database.
    b. Retrieve data from the relevant collections and store them appropriately.

```
# Load data (update collection names)
users = list(db["users"].find())
products = list(db["products"].find())
interactions = list(db["user_product_interactions"].find())
```

2. Create user_ids, product_ids, and product_lookup mappings. These will be critical during training and prediction.
3. User and Product Features: The model's quality greatly improves by adding user and product metadata. Examples of features:
    a. User: city, age group, browsing history, user type
    b. Product: category, brand, price range, color, season

    In LightFM, we pass these features to the model using `build_user_features` and `build_item_features`, allowing the model to learn better embeddings and deal with sparse interaction matrices.
4. Model Training: We use LightFM's `warp` loss for implicit data. It focuses on ranking items correctly rather than predicting exact scores.
    Other available loss functions:
    a. `logistic`: Binary classification loss
    b. `bpr`: Bayesian Personalized Ranking
    c. `warp-kos`: An extension of WARP with better sampling

    Each has its use-cases, but `warp` often performs better when the goal is ranking.

5. Cosine Similarity for Similar Products: Cosine similarity is computed between item embeddings to find similar products. This can power features like 'Customers also viewed' or suggest alternatives to out-of-stock items.
6. Saving the Model and Artifacts: Saving the trained model and matrices with `pickle` and `scipy.sparse` allows reusability without retraining. You can deploy the model in production or reload it for scheduled batch predictions.

    Saved Artifacts:

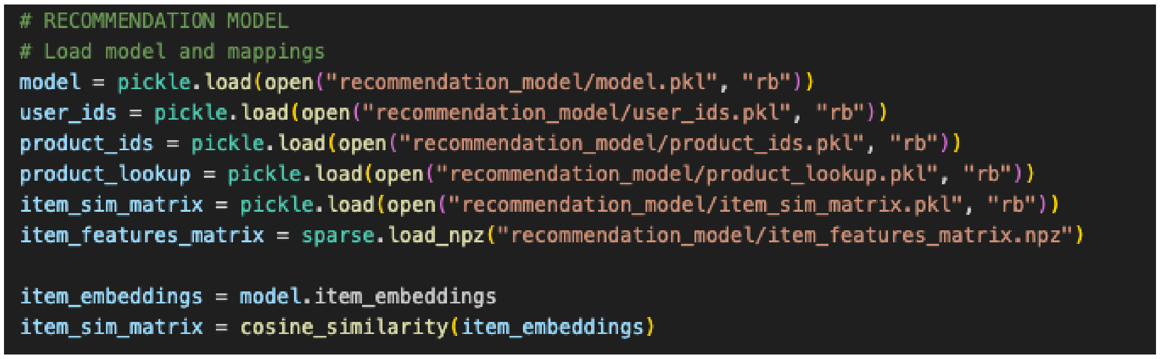
    a. Trained LightFM model
    b. User and product IDs
    c. Lookup dictionary

d. Item similarity matrix
e. Feature matrices

3 . Flask Backend Integration

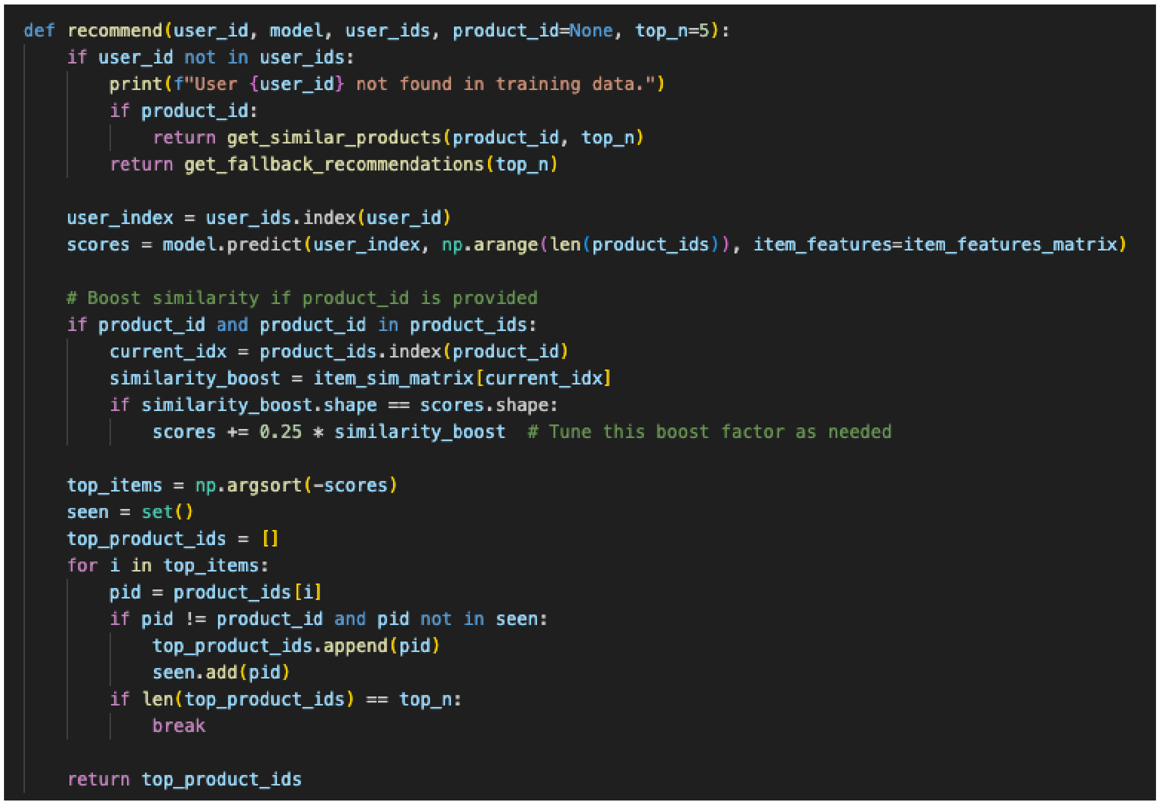
The trained model is deployed using a Flask API. The server loads the model and exposes endpoints for serving recommendations.

1. Load the saved model and artifacts.



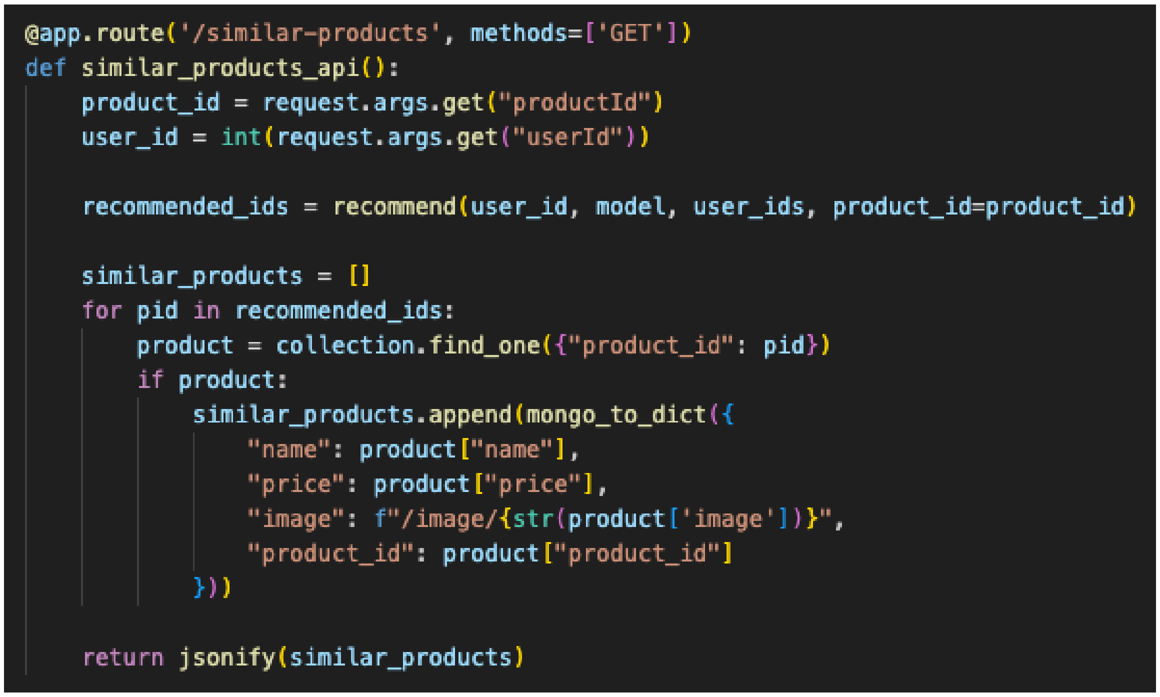
```
# RECOMMENDATION MODEL
# Load model and mappings
model = pickle.load(open("recommendation_model/model.pkl", "rb"))
user_ids = pickle.load(open("recommendation_model/user_ids.pkl", "rb"))
product_ids = pickle.load(open("recommendation_model/product_ids.pkl", "rb"))
product_lookup = pickle.load(open("recommendation_model/product_lookup.pkl", "rb"))
item_sim_matrix = pickle.load(open("recommendation_model/item_sim_matrix.pkl", "rb"))
item_features_matrix = sparse.load_npz("recommendation_model/item_features_matrix.npz")

item_embeddings = model.item_embeddings
item_sim_matrix = cosine_similarity(item_embeddings)
```

2. Define functions for getting recommendations



```
def recommend(user_id, model, user_ids, product_id=None, top_n=5):
    if user_id not in user_ids:
        print(f"User {user_id} not found in training data.")
        if product_id:
            return get_similar_products(product_id, top_n)
        return get_fallback_recommendations(top_n)

    user_index = user_ids.index(user_id)
    scores = model.predict(user_index, np.arange(len(product_ids)), item_features=item_features_matrix)

    # Boost similarity if product_id is provided
    if product_id and product_id in product_ids:
        current_idx = product_ids.index(product_id)
        similarity_boost = item_sim_matrix[current_idx]
        if similarity_boost.shape == scores.shape:
            scores += 0.25 * similarity_boost  # Tune this boost factor as needed

    top_items = np.argsort(-scores)
    seen = set()
    top_product_ids = []
    for i in top_items:
        pid = product_ids[i]
        if pid != product_id and pid not in seen:
            top_product_ids.append(pid)
            seen.add(pid)
        if len(top_product_ids) == top_n:
            break

    return top_product_ids
```

3. Define endpoint for recommendations



```python
@app.route('/similar-products', methods=['GET'])
def similar_products_api():
    product_id = request.args.get("productId")
    user_id = int(request.args.get("userId"))

    recommended_ids = recommend(user_id, model, user_ids, product_id=product_id)

    similar_products = []
    for pid in recommended_ids:
        product = collection.find_one({"product_id": pid})
        if product:
            similar_products.append(mongo_to_dict({
                "name": product["name"],
                "price": product["price"],
                "image": f"/image/{str(product['image'])}",
                "product_id": product["product_id"]
            }))

    return jsonify(similar_products)
```

The `recommend` function combines model predictions with cosine similarity to return a mix of collaborative and content-based suggestions. If the user or product is unknown, fallback recommendations based on popularity are returned.