```
!pip install implicit
!pip install gradio
```

```
import implicit
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
import tensorflow as tf
from tqdm import tqdm
from sentence_transformers import SentenceTransformer
from tensorflow.keras import layers, Model, losses, optimizers, callbacks
from sklearn.metrics.pairwise import cosine_similarity
import pickle
from tensorflow.keras.models import load_model
import gradio as gr
```

```
orders = pd.read_csv("/content/drive/MyDrive/Recommender_Project/data/orders_small.csv")
order_products = pd.read_csv("/content/drive/MyDrive/Recommender_Project/data/order_products_small.csv")
products = pd.read_csv("/content/drive/MyDrive/Recommender_Project/data/products_small.csv")
aisles = pd.read_csv("/content/drive/MyDrive/Recommender_Project/data/aisles.csv")
departments = pd.read_csv("/content/drive/MyDrive/Recommender_Project/data/departments.csv")
```

```
orders.head()
```

	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_order
0	361493	27	prior	1	3	9	NaN
1	1662354	27	prior	2	2	17	6.0
2	965677	27	prior	3	3	8	1.0
3	2842504	27	prior	4	5	13	2.0
4	1007361	27	prior	5	2	15	4.0

```
products.head()
```

	product_id	product_name	aisle_id	department_id
0	1	Chocolate Sandwich Cookies	61	19
1	2	All-Seasons Salt	104	13
2	3	Robust Golden Unsweetened Oolong Tea	94	7
3	8	Cut Russet Potatoes Steam N' Mash	116	1
4	10	Sparkling Orange Juice & Prickly Pear Beverage	115	7

Next steps: Generate code with products | New interactive sheet

```
order_products.head()
```

	order_id	product_id	add_to_cart_order	reordered
0	4	46842	1	0
1	4	26434	2	1
2	4	39758	3	1
3	4	27761	4	1
4	4	10054	5	1

```
aisles.head()
```

	aisle_id	aisle
0	1	prepared soups salads
1	2	specialty cheeses
2	3	energy granola bars
3	4	instant foods
4	5	marinades meat preparation

Next steps: Generate code with aisles | New interactive sheet

```
1 departments.head()
```

	department_id	department
0	1	frozen
1	2	other
2	3	bakery
3	4	produce
4	5	alcohol

Next steps: Generate code with departments | New interactive sheet

Collaborative Filtering

```
# --- Build unique (user, product) interactions ---
# Merge orders and order_products to get user-product interactions
user_products = orders[['order_id','user_id']].merge(
    order_products[['order_id','product_id']], on='order_id'
)

# Map to contiguous indices for embeddings
user_ids = user_products['user_id'].unique()
product_ids = user_products['product_id'].unique()

user_map = {uid: i for i, uid in enumerate(user_ids)}
product_map = {pid: i for i, pid in enumerate(product_ids)}

user_products['user_idx'] = user_products['user_id'].map(user_map)
user_products['product_idx'] = user_products['product_id'].map(product_map)

# Get unique positive pairs (user_idx, product_idx)
positives = user_products[['user_idx','product_idx']].drop_duplicates().values
print("Unique positives:", len(positives))

# --- Negative sampling ---
# For each positive interaction, sample a fixed number of negative interactions (user, unpurchased item)
n_neg = 4
num_users = len(user_map)
num_items = len(product_map)

pairs = []
# Create a dictionary mapping user index to a set of item indices they have purchased
user_pos = user_products.groupby('user_idx')['product_idx'].apply(set).to_dict()

# Iterate through positive interactions and sample negatives
for u, i in tqdm(positives, desc="Sampling negatives"):
    pairs.append((u, i, 1))  # Add the positive interaction with label 1
    negs = 0
    while negs < n_neg:
        # Randomly sample an item index
        j = np.random.randint(0, num_items)
        # Check if the sampled item has not been purchased by the user
        if j not in user_pos[u]:
            pairs.append((u, j, 0)) # Add the negative interaction with label 0
            negs += 1

# Convert the list of pairs to a numpy array
pairs = np.array(pairs, dtype=np.int32)
```

```
# Separate users, items, and labels
users_arr, items_arr, labels_arr = pairs[:,0], pairs[:,1], pairs[:,2].astype(np.float32)

# Shuffle the data and split into training and validation sets
perm = np.random.permutation(len(users_arr))
users_arr, items_arr, labels_arr = users_arr[perm], items_arr[perm], labels_arr[perm]

split = int(0.9 * len(users_arr))
users_train, items_train, labels_train = users_arr[:split], items_arr[:split], labels_arr[:split]
users_val, items_val, labels_val = users_arr[split:], items_arr[split:], labels_arr[split:]

print("Train samples:", len(users_train), "Val samples:", len(users_val))

# Build tf.data.Dataset for efficient training
batch_size = 4096
train_ds = tf.data.Dataset.from_tensor_slices(((users_train, items_train), labels_train))\
    .shuffle(100000).batch(batch_size).prefetch(tf.data.AUTOTUNE)
val_ds = tf.data.Dataset.from_tensor_slices(((users_val, items_val), labels_val))\
    .batch(batch_size).prefetch(tf.data.AUTOTUNE)

# Print the number of unique users and items
num_users, num_items
```

```
Unique positives: 1637937
Sampling negatives: 100%|██████████| 1637937/1637937 [00:29<00:00, 56283.46it/s]
Train samples: 7370716 Val samples: 818969
(10000, 20000)
```

```
# === TensorFlow CF model (MF) + training ===

# Define hyperparameters
embedding_dim = 64     # Dimension of user and item embeddings; try 32, 64, or 128
lr = 1e-3              # Learning rate for the optimizer
epochs = 6             # Number of training epochs
model_save_path = "tf_cf_mf.h5" # Path to save the best model during training

# --- Model: simple Matrix Factorization (dot product + bias) ---
class MatrixFactorization(Model):
    def __init__(self, num_users, num_items, embed_dim):
        super().__init__()
        # Embedding layers for users and items
        self.user_emb = layers.Embedding(num_users, embed_dim, embeddings_initializer="he_normal")
        self.item_emb = layers.Embedding(num_items, embed_dim, embeddings_initializer="he_normal")
        # Bias layers for users and items
        self.user_bias = layers.Embedding(num_users, 1, embeddings_initializer="zeros")
        self.item_bias = layers.Embedding(num_items, 1, embeddings_initializer="zeros")
        # Sigmoid activation to output a probability-like score
        self.sig = layers.Activation("sigmoid")

    def call(self, inputs):
        user_idx, item_idx = inputs  # Input tensors: user and item indices (batch,)
        u = self.user_emb(user_idx)      # User embeddings (batch, embed_dim)
        v = self.item_emb(item_idx)      # Item embeddings (batch, embed_dim)
        ub = tf.squeeze(self.user_bias(user_idx), axis=-1)  # User biases (batch,)
        ib = tf.squeeze(self.item_bias(item_idx), axis=-1)  # Item biases (batch,)
        dot = tf.reduce_sum(u * v, axis=1)                # Dot product of embeddings (batch,)
        x = dot + ub + ib                                 # Add biases to the dot product
        out = self.sig(x)                                 # Apply sigmoid activation
        return out

# Instantiate the model
model = MatrixFactorization(num_users, num_items, embedding_dim)

# Compile the model
model.compile(
    optimizer=optimizers.Adam(learning_rate=lr),
    loss=losses.BinaryCrossentropy(),   # Use Binary Crossentropy for implicit feedback (binary classification)
    metrics=[tf.keras.metrics.BinaryAccuracy(name="acc")] # Metric to monitor during training
)

# Define callbacks for training
es = callbacks.EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True) # Stop training if validation loss doe
ck = callbacks.ModelCheckpoint(model_save_path, save_best_only=True, monitor="val_loss") # Save the best model based on valida

# Train the model
history = model.fit(train_ds, validation_data=val_ds, epochs=epochs, callbacks=[es, ck])

```

```
# Save the final model and extract embeddings
model.save(model_save_path, include_optimizer=False)
user_embeddings = model.user_emb.get_weights()[0]   # Extract user embeddings
item_embeddings = model.item_emb.get_weights()[0]   # Extract item embeddings

print("Training done. Embedding shapes:", user_embeddings.shape, item_embeddings.shape)


# === Recommend helper: returns product_ids and scores ===
# Build product_index_to_id array (col index -> product_id) from product_map
product_index_to_id = np.array(list(product_map.keys()))

# Build user -> set(item_idx) for masking (do this once)
user_seen = orders[['order_id','user_id']].merge(order_products[['order_id','product_id']], on='order_id')
user_seen['user_idx'] = user_seen['user_id'].map(user_map)
user_seen['item_idx'] = user_seen['product_id'].map(product_map)
user_seen = user_seen.dropna(subset=['user_idx','item_idx']).astype({'user_idx':int,'item_idx':int})
user_seen_map = user_seen.groupby('user_idx')['item_idx'].apply(set).to_dict()


```

```
Epoch 1/6
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - acc: 0.8446 - loss: 0.5221WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 55s 30ms/step - acc: 0.8446 - loss: 0.5220 - val_acc: 0.8616 - val_loss: 0.3261
Epoch 2/6
1797/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - acc: 0.8646 - loss: 0.3184WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 55s 30ms/step - acc: 0.8646 - loss: 0.3184 - val_acc: 0.8698 - val_loss: 0.3033
Epoch 3/6
1796/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - acc: 0.8718 - loss: 0.2964WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 54s 30ms/step - acc: 0.8718 - loss: 0.2964 - val_acc: 0.8730 - val_loss: 0.2921
Epoch 4/6
1797/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - acc: 0.8764 - loss: 0.2833WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 51s 28ms/step - acc: 0.8764 - loss: 0.2833 - val_acc: 0.8775 - val_loss: 0.2817
Epoch 5/6
1797/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - acc: 0.8831 - loss: 0.2687WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 84s 29ms/step - acc: 0.8831 - loss: 0.2687 - val_acc: 0.8816 - val_loss: 0.2718
Epoch 6/6
1798/1800 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - acc: 0.8897 - loss: 0.2533WARNING:absl:You are saving your model as an HDF5 file via
1800/1800 ━━━━━━━━━━━━━━━━━━━━ 54s 30ms/step - acc: 0.8897 - loss: 0.2533 - val_acc: 0.8843 - val_loss: 0.2645
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is
Training done. Embedding shapes: (10000, 64) (20000, 64)
```

```
def recommend_for_user_CF(user_id, top_k=10, return_scores=True):
    # Check if the user ID exists in our mapping
    if user_id not in user_map:
        raise ValueError(f"User ID {user_id} not found in the user map.")

    u_idx = user_map[user_id]          # Get the internal index for the user
    u_vec = user_embeddings[u_idx]     # Get the user's embedding vector (embed_dim,)

    # Calculate scores for all items by taking the dot product of user embedding with all item embeddings
    scores = item_embeddings @ u_vec   # (num_items,)

    # Mask items the user has already seen (purchased)
    seen = user_seen_map.get(u_idx, set()) # Get the set of item indices the user has seen
    scores[list(seen)] = -np.inf       # Set scores of seen items to negative infinity to exclude them from top-k

    # Get the indices of the top-k highest scores
    top_idx = np.argpartition(-scores, range(top_k))[:top_k]

    # Sort the top-k indices by their scores in descending order
    top_idx = top_idx[np.argsort(-scores[top_idx])]

    # Get the original product IDs for the top-k indices
    prod_ids = product_index_to_id[top_idx]

    # Return product IDs and scores if requested, otherwise just product IDs
    if return_scores:
        return list(prod_ids), list(scores[top_idx])
    return list(prod_ids)

```

```
# Example usage: recommend for user 27 (real Instacart ID)
example_uid = 27
pids, sc = recommend_for_user_CF(example_uid, top_k=10)
print("TF-MF recommendations (product_id, score):")
```

```
# Iterate through the recommended product IDs and their scores
for pid, s in zip(pids, sc):
    # Find the product name using the product_id from the products DataFrame
    name = products.loc[products['product_id']==pid,'product_name'].values[0]
    # Print the product name, ID, and recommendation score
    print(f"- {name} (id={pid}) score={s:.4f}")
```

```
TF-MF recommendations (product_id, score):
- Sparkling Lemon Water (id=21709) score=4.1237
- Clementines, Bag (id=28199) score=3.7631
- Organic Whole String Cheese (id=22035) score=3.7572
- Peach Pear Flavored Sparkling Water (id=26620) score=3.7144
- Vitamin C Super Orange Dietary Supplement (id=2238) score=3.6471
- Kiwi Sandia Sparkling Water (id=12576) score=3.5266
- Dark Chocolate Nuts & Sea Salt Bars (id=45645) score=3.5177
- Earl Grey Tea (id=48946) score=3.4537
- Organic Reduced Fat 2% Milk (id=5785) score=3.4254
- Hass Avocados (id=12341) score=3.4170
```

Content Based Filtering

```
# Merge products with aisle and department names to create a richer product description
products_full = products.merge(aisles, on="aisle_id").merge(departments, on="department_id")

# Create an enriched text column by concatenating product name, aisle, and department
# This text will be used to generate content-based embeddings
products_full['full_text'] = (
    products_full['product_name'].fillna("") +  # Use fillna("") to handle potential missing values gracefully
    " [aisle: " + products_full['aisle'].fillna("") + "]" +
    " [dept: " + products_full['department'].fillna("") + "]"
)

# Display the head of the dataframe to show the new 'full_text' column
print(products_full[['product_name','full_text']].head(5))
```

```
                                product_name  \
0                    Chocolate Sandwich Cookies
1                             All-Seasons Salt
2           Robust Golden Unsweetened Oolong Tea
3              Cut Russet Potatoes Steam N' Mash
4  Sparkling Orange Juice & Prickly Pear Beverage

                                       full_text
0  Chocolate Sandwich Cookies [aisle: cookies cak...
1  All-Seasons Salt [aisle: spices seasonings] [d...
2  Robust Golden Unsweetened Oolong Tea [aisle: t...
3  Cut Russet Potatoes Steam N' Mash [aisle: froz...
4  Sparkling Orange Juice & Prickly Pear Beverage...
```

```
# Load a pre-trained sentence transformer model
# This model will be used to convert text descriptions into numerical vectors (embeddings)
st_model = SentenceTransformer('all-MiniLM-L6-v2')

# Prepare the product texts and their corresponding product IDs
product_texts = products_full['full_text'].tolist()
product_ids = products_full['product_id'].tolist()

# Encode the enriched product texts into embeddings
# show_progress_bar=True displays a progress bar during encoding
# convert_to_numpy=True ensures the output is a NumPy array
# normalize_embeddings=True scales the embeddings to unit length, for cosine similarity
item_embeddings_content = st_model.encode(
    product_texts,
    show_progress_bar=True,
    convert_to_numpy=True,
    normalize_embeddings=True
)

# Print the shape of the generated embeddings
print("Embeddings shape:", item_embeddings_content.shape)
```

```
Batches: 100%  625/625 [00:06<00:00, 126.93it/s]
Embeddings shape: (20000, 384)
```

```
# Map product_id -> row index in embedding matrix
product_id_to_index = {pid: i for i, pid in enumerate(product_ids)}
```

```
def recommend_for_user_CBF(user_id, top_k=10):
    # Find the orders placed by the user
    user_orders = orders[orders['user_id']==user_id]['order_id']

    # Find all unique products purchased by the user across their orders
    user_products_hist = order_products[order_products['order_id'].isin(user_orders)]['product_id'].unique()

    # If the user has no purchase history, return an empty list
    if len(user_products_hist) == 0:
        return []

    # Get the indices of the purchased items in the content-based embedding matrix
    # Filter out products that might not be in the products_full dataframe used for embeddings
    hist_indices = [product_id_to_index[pid] for pid in user_products_hist if pid in product_id_to_index]

    # Calculate the user's profile by averaging the content embeddings of their purchased items
    user_profile = item_embeddings_content[hist_indices].mean(axis=0).reshape(1, -1)

    # Compute the cosine similarity between the user profile and all item embeddings
    sims = cosine_similarity(user_profile, item_embeddings_content)[0]

    # Mask already purchased items by setting their similarity scores to a very low value
    for idx in hist_indices:
        sims[idx] = -1 # Use -1 as cosine similarity is in [-1, 1]

    # Get the indices of the top-k items with the highest similarity scores
    top_idx = np.argpartition(-sims, range(top_k))[:top_k]

    # Sort the top-k indices based on their similarity scores in descending order
    top_idx = top_idx[np.argsort(-sims[top_idx])]

    # Get the original product IDs and their corresponding similarity scores for the top-k indices
    rec_ids = [product_ids[i] for i in top_idx]
    rec_scores = [sims[i] for i in top_idx]

    # Return the recommendations as a list of (product_id, score) tuples
    return list(zip(rec_ids, rec_scores))
```

```
# Example: get content-based recommendations for user 27
recs = recommend_for_user_CBF(27, top_k=10)

print("Content-based recommendations for user 27 (with aisle+dept):")
# Iterate through the recommendations and print details
for pid, score in recs:
    # Find the product details (name, aisle, department) using the product_id
    product_info = products_full.loc[products_full['product_id']==pid].iloc[0]
    name = product_info['product_name']
    aisle = product_info['aisle']
    dept = product_info['department']
    # Print the recommendation with its details and score
    print(f"- {name} | aisle={aisle}, dept={dept} (sim={score:.3f})")
```

```
Content-based recommendations for user 27 (with aisle+dept):
- Organic Nuts & Chocolate Beverage | aisle=nuts seeds dried fruit, dept=snacks (sim=0.793)
- Strawberry | aisle=yogurt, dept=dairy eggs (sim=0.784)
- Organic Jelly Beans | aisle=candy chocolate, dept=snacks (sim=0.782)
- Fruit Medley, Organic | aisle=fruit vegetable snacks, dept=snacks (sim=0.780)
- Nutri-Grain Strawberry | aisle=breakfast bars pastries, dept=breakfast (sim=0.776)
- Yogurt, Lowfat, Organic, Vanilla Bean | aisle=yogurt, dept=dairy eggs (sim=0.766)
- Organic Strawberry Energy Chews | aisle=candy chocolate, dept=snacks (sim=0.763)
- Organic Chunky Strawberry Fruit Bars | aisle=ice cream ice, dept=frozen (sim=0.763)
- Fruit Spread, Organic, Strawberry | aisle=spreads, dept=pantry (sim=0.763)
- Organic Mixed Berry Snack Bar | aisle=energy granola bars, dept=snacks (sim=0.760)
```

```
def recommend_hybrid(user_id, top_k=10, alpha=0.6):
    """
    Generates hybrid recommendations for a user by combining CF and CBF scores.

    Args:
        user_id (int): The ID of the user for whom to generate recommendations.
        top_k (int): The number of top recommendations to return.
        alpha (float): The weighting factor for CF scores (0.0 to 1.0).
                       A higher alpha gives more weight to collaborative filtering.
```

```
    Returns:
        list of tuples: A list of (product_id, hybrid_score) for the top_k recommended products.
                        Returns an empty list if the user has no purchase history.
    """
    # Check if the user ID exists in our mapping (important for CF part)
    if user_id not in user_map:
        # If user not in CF model, fallback to pure CBF or popularity if no history
        print(f"Warning: User ID {user_id} not found in CF user map. Using only CBF if history exists.")
        # Proceed to CBF part, which handles users with no history
        return recommend_for_user_CBF(user_id, top_k=top_k) # CBF returns (pid, sim)

    u_idx = user_map[user_id]

    # --- CF scores ---
    # Get the user's embedding from the CF model
    u_vec_cf = user_embeddings[u_idx]
    # Calculate CF scores for all items using the dot product
    scores_cf = item_embeddings @ u_vec_cf  # (num_items,)

    # --- CBF scores ---
    # Find the orders placed by the user to get purchase history for CBF
    user_orders = orders[orders['user_id']==user_id]['order_id']
    user_products_hist = order_products[order_products['order_id'].isin(user_orders)]['product_id'].unique()

    # We'll return empty if no history for CBF, relying on the initial check for CF.
    if len(user_products_hist) == 0:
        print(f"Warning: User ID {user_id} has no purchase history for CBF.")
        return []

    # Get the indices of the purchased items in the content-based embedding matrix
    # Filter out products that might not be in the products_full dataframe used for embeddings
    hist_indices = [product_id_to_index[pid] for pid in user_products_hist if pid in product_id_to_index]

    # Calculate the user's content-based profile by averaging embeddings of purchased items
    user_profile = item_embeddings_content[hist_indices].mean(axis=0).reshape(1, -1)
    # Compute cosine similarity between the user profile and all item content embeddings
    scores_cbf = cosine_similarity(user_profile, item_embeddings_content)[0]

    # --- Combine scores ---
    # Calculate the final hybrid score as a weighted sum of CF and CBF scores
    scores = alpha * scores_cf + (1 - alpha) * scores_cbf

    # Mask already purchased items by setting their scores to negative infinity
    # This prevents recommending items the user has already bought
    seen = set(hist_indices) # Use indices from CBF history as a proxy for seen items
    for idx in seen:
        scores[idx] = -np.inf

    # Get the indices of the top-k items with the highest hybrid scores
    top_idx = np.argpartition(-scores, range(top_k))[:top_k]

    # Sort the top-k indices based on their hybrid scores in descending order
    top_idx = top_idx[np.argsort(-scores[top_idx])]

    # Get the original product IDs and their corresponding hybrid scores for the top-k indices
    rec_ids = [product_ids[i] for i in top_idx]
    rec_scores = [scores[i] for i in top_idx]

    # Return the recommendations as a list of (product_id, score) tuples
    return list(zip(rec_ids, rec_scores))
```

```
# Example: get hybrid recommendations for user 27 with alpha=0.6
recs = recommend_hybrid(27, top_k=10, alpha=0.6)

print("Hybrid recommendations for user 27:")
# Iterate through the recommendations and print details
for pid, score in recs:
    # Find the product details (name, aisle, department) using the product_id from products_full
    product_info = products_full.loc[products_full['product_id']==pid,'product_name'].values[0]
    aisle = products_full.loc[products_full['product_id']==pid,'aisle'].values[0]
    dept = products_full.loc[products_full['product_id']==pid,'department'].values[0]
    # Print the recommendation with its details and hybrid score
    print(f"- {product_info} | aisle={aisle}, dept={dept} (hybrid_score={score:.3f})")
```

```
Hybrid recommendations for user 27:
- Snack Bags | aisle=more household, dept=household (hybrid_score=2.891)
- Minis Original Saltine Crackers | aisle=crackers, dept=snacks (hybrid_score=2.788)
- Peach Sparkling Energy Water | aisle=energy sports drinks, dept=beverages (hybrid_score=2.723)
- Berry Punch Flavored Fruit Drink | aisle=refrigerated, dept=beverages (hybrid_score=2.504)
- Caramel Sauce | aisle=ice cream toppings, dept=snacks (hybrid_score=2.501)
- Organic Rice Vinegar | aisle=asian foods, dept=international (hybrid_score=2.478)
- Bean and Cheese Gorditas | aisle=prepared meals, dept=deli (hybrid_score=2.474)
- Yerba Mate Sparkling Classic Gold | aisle=tea, dept=beverages (hybrid_score=2.464)
- 24/7 Performance Cat Litter | aisle=cat food care, dept=pets (hybrid_score=2.447)
- Organic Yukon Gold Potato Bag | aisle=fresh vegetables, dept=produce (hybrid_score=2.385)
```

```python
class HybridRecommender:
    """
    A class to combine Collaborative Filtering (CF) and Content-Based Filtering (CBF)
    recommendations.
    """
    def __init__(self, cf_model=None, user_embeddings=None, item_embeddings_cf=None,
                 item_embeddings_cbf=None, product_ids=None,
                 user_map=None, product_map=None, product_id_to_index=None,
                 alpha=0.6):
        """
        Initializes the HybridRecommender with necessary models, embeddings, and mappings.

        Args:
            cf_model (tf.keras.Model, optional): The trained Collaborative Filtering model. Defaults to None.
            user_embeddings (np.ndarray, optional): User embeddings from the CF model. Defaults to None.
            item_embeddings_cf (np.ndarray, optional): Item embeddings from the CF model. Defaults to None.
            item_embeddings_cbf (np.ndarray, optional): Item embeddings from the CBF model. Defaults to None.
            product_ids (np.ndarray, optional): Array of original product IDs. Defaults to None.
            user_map (dict, optional): Mapping from original user_id to internal index. Defaults to None.
            product_map (dict, optional): Mapping from original product_id to internal index (for CF). Defaults to None.
            product_id_to_index (dict, optional): Mapping from original product_id to internal index (for CBF). Defaults to N
            alpha (float, optional): Weighting factor for CF vs CBF scores (0.0 to 1.0). Defaults to 0.6.
        """
        self.cf_model = cf_model
        self.user_embeddings = user_embeddings
        self.item_embeddings_cf = item_embeddings_cf
        self.item_embeddings_cbf = item_embeddings_cbf
        self.product_ids = product_ids
        self.user_map = user_map
        self.product_map = product_map # Keep if needed for other CF related lookups
        self.product_id_to_index = product_id_to_index # For CBF index lookups
        self.alpha = alpha

    # ---------------------------
    # Save and Load
    # ---------------------------
    def save(self, path="recommender"):
        """
        Saves the recommender components to disk.

        Args:
            path (str, optional): Base path for saving files. Defaults to "recommender".
        """
        # Save embeddings
        np.save(f"{path}_user_embeddings.npy", self.user_embeddings)
        np.save(f"{path}_item_embeddings_cf.npy", self.item_embeddings_cf)
        np.save(f"{path}_item_embeddings_cbf.npy", self.item_embeddings_cbf)
        np.save(f"{path}_product_ids.npy", self.product_ids)

        # Save mappings and alpha
        with open(f"{path}_mappings.pkl", "wb") as f:
            pickle.dump({
                "user_map": self.user_map,
                "product_map": self.product_map,
                "product_id_to_index": self.product_id_to_index,
                "alpha": self.alpha
            }, f)

        # Save CF model (optional, depends on if you need to load the full model later)
        if self.cf_model is not None:
            try:
                self.cf_model.save(f"{path}_cf_model.h5", include_optimizer=False)
            except Exception as e:
                print(f"Warning: Could not save TF model: {e}")

```

```
    @classmethod
    def load(cls, path="recommender"):
        """
        Loads the recommender components from disk.

        Args:
            path (str, optional): Base path where files were saved. Defaults to "recommender".

        Returns:
            HybridRecommender: An instance of the HybridRecommender.
        """
        # Load embeddings
        user_embeddings = np.load(f"{path}_user_embeddings.npy")
        item_embeddings_cf = np.load(f"{path}_item_embeddings_cf.npy")
        item_embeddings_cbf = np.load(f"{path}_item_embeddings_cbf.npy")
        product_ids = np.load(f"{path}_product_ids.npy")

        # Load mappings
        with open(f"{path}_mappings.pkl", "rb") as f:
            maps = pickle.load(f)

        # Load CF model
        cf_model = None
        try:
            # compile=False to avoid needing the original training setup
            cf_model = load_model(f"{path}_cf_model.h5", compile=False)
        except:
            # Handle cases where the model file might not exist or is corrupted
            print(f"Warning: Could not load TF model from {path}_cf_model.h5")
            pass

        return cls(cf_model=cf_model,
                   user_embeddings=user_embeddings,
                   item_embeddings_cf=item_embeddings_cf,
                   item_embeddings_cbf=item_embeddings_cbf,
                   product_ids=product_ids,
                   user_map=maps["user_map"],
                   product_map=maps.get("product_map"), # Use .get for backward compatibility if product_map wasn't saved
                   product_id_to_index=maps["product_id_to_index"],
                   alpha=maps.get("alpha", 0.6)) # Use .get for backward compatibility


    # ---------------------------
    # Recommend
    # ---------------------------
    def recommend(self, user_id, top_k=10, orders=None, order_products=None):
        """
        Generates hybrid recommendations for a user.

        Args:
            user_id (int): The ID of the user for whom to generate recommendations.
            top_k (int, optional): The number of top recommendations to return. Defaults to 10.
            orders (pd.DataFrame, optional): DataFrame containing order information. Required for CBF history lookup.
            order_products (pd.DataFrame, optional): DataFrame containing order product information. Required for CBF history

        Returns:
            list of tuples: A list of (product_id, hybrid_score) for the top_k recommended products.
                            Returns an empty list if the user is not in the user map or has no purchase history for CBF.
        Raises:
            ValueError: If orders or order_products DataFrames are not provided for CBF calculation.
        """
        # Ensure user exists in the CF map (needed for user embedding)
        if user_id not in self.user_map:
            print(f"User ID {user_id} not found in user map.")
            return []

        u_idx = self.user_map[user_id]

        # --- CF scores ---
        # Calculate CF scores for all items using the dot product of user embedding with all item embeddings
        u_vec_cf = self.user_embeddings[u_idx]
        scores_cf = self.item_embeddings_cf @ u_vec_cf  # (num_items,)

        # --- CBF scores ---
        # CBF requires user purchase history, which is in the provided DataFrames
        if orders is None or order_products is None:
```

```
            raise ValueError("Need orders and order_products DataFrames to compute CBF")

        # Find the orders placed by the user
        user_orders = orders[orders['user_id']==user_id]['order_id']

        # Find all unique products purchased by the user across their orders
        user_products_hist = order_products[order_products['order_id'].isin(user_orders)]['product_id'].unique()

        if len(user_products_hist) == 0:
            print(f"User ID {user_id} has no purchase history for CBF.")
            return []

        # Get the indices of the purchased items in the content-based embedding matrix
        # Filter out products that might not be in the products_full dataframe used for embeddings
        hist_indices = [self.product_id_to_index[pid] for pid in user_products_hist if pid in self.product_id_to_index]

        # Calculate the user's content-based profile by averaging embeddings of purchased items
        user_profile = self.item_embeddings_cbf[hist_indices].mean(axis=0).reshape(1, -1)
        # Compute cosine similarity between the user profile and all item content embeddings
        scores_cbf = cosine_similarity(user_profile, self.item_embeddings_cbf)[0]

        # --- Combine scores ---
        # Calculate the final hybrid score as a weighted sum of CF and CBF scores
        scores = self.alpha * scores_cf + (1 - self.alpha) * scores_cbf

        # Mask already purchased items by setting their scores to negative infinity
        # This prevents recommending items the user has already bought
        # Use indices from CBF history as a proxy for seen items
        seen_indices = set(hist_indices)
        scores[list(seen_indices)] = -np.inf

        # --- Get Top-K recommendations ---
        # Get the indices of the top-k items with the highest hybrid scores
        top_idx = np.argpartition(-scores, range(top_k))[:top_k]

        # Sort the top-k indices based on their hybrid scores in descending order
        top_idx = top_idx[np.argsort(-scores[top_idx])]

        # Get the original product IDs and their corresponding hybrid scores for the top-k indices
        rec_ids = [self.product_ids[i] for i in top_idx]
        rec_scores = [scores[i] for i in top_idx]

        return list(zip(rec_ids, rec_scores))
```

```
# Create and save recommender
recommender = HybridRecommender(
    cf_model=model,
    user_embeddings=user_embeddings,
    item_embeddings_cf=item_embeddings,
    item_embeddings_cbf=item_embeddings_content,
    product_ids=np.array(product_ids),
    user_map=user_map,
    product_map=product_map,
    product_id_to_index=product_id_to_index,
    alpha=0.6
)

recommender.save("/content/drive/MyDrive/Recommender_Project/models/my_recommender")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is
```

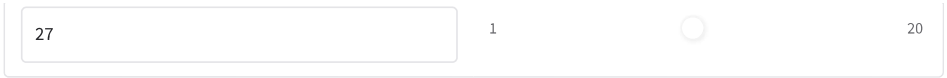
```
# Load recommender
recommender_loaded = HybridRecommender.load(
    "/content/drive/MyDrive/Recommender_Project/models/my_recommender"
)

# --- Precompute most popular products (for fallback) ---
popular_items = order_products["product_id"].value_counts().index.tolist()

def recommend_with_history(user_id, top_k=10):
    try:
        user_id = int(user_id)

        # --- Get recommendations ---
        recs = recommender_loaded.recommend(
```

```
            user_id, top_k=top_k, orders=orders, order_products=order_products
        )

        # --- Past purchases (last order) ---
        last_order = orders[orders["user_id"] == user_id]["order_id"].max()
        if pd.isna(last_order):
            past_items = []
        else:
            past_items = order_products[order_products["order_id"] == last_order]["product_id"].tolist()

        # --- Build past purchases table ---
        past_data = []
        for pid in past_items[:top_k]:
            row = products_full.loc[products_full["product_id"] == pid].iloc[0]
            past_data.append([row["product_name"], row["aisle"], row["department"]])
        past_df = pd.DataFrame(past_data, columns=["Product", "Aisle", "Department"])
        if past_df.empty:
            past_df = pd.DataFrame([["No past purchases found.", "-", "-"]],
                                   columns=["Product", "Aisle", "Department"])

        # --- Build recommendations table (with fallback) ---
        rec_data = []
        if recs:
            for pid, score in recs:
                row = products_full.loc[products_full["product_id"] == pid].iloc[0]
                rec_data.append([row["product_name"], row["aisle"], row["department"], f"{score:.3f}"])
        else:
            rec_data.append(["⚠️ No personalized recs", "-", "-", "-"])
            rec_data.append(["🔥 Popular items:", "-", "-", "-"])
            for pid in popular_items[:top_k]:
                row = products_full.loc[products_full["product_id"] == pid].iloc[0]
                rec_data.append([row["product_name"], row["aisle"], row["department"], "-"])
        rec_df = pd.DataFrame(rec_data, columns=["Product", "Aisle", "Department", "Score"])

        return past_df, rec_df

    except Exception as e:
        return pd.DataFrame([[f"❌ Error: {str(e)}", "-", "-"]],
                            columns=["Product", "Aisle", "Department"]), pd.DataFrame()
```

```
Warning: Could not load TF model from /content/drive/MyDrive/Recommender_Project/models/my_recommender_cf_model.h5
```

```
# --- Gradio UI ---
with gr.Blocks() as demo:
    gr.Markdown("# 🛒 Instacart Personalized Recommender")
    gr.Markdown("Compare a user's **past purchases** with their **new recommendations** (with fallback).")

    with gr.Row():
        user_id_in = gr.Number(value=27, label="User ID")
        topk_in = gr.Slider(1, 20, value=10, step=1, label="Top-K Recommendations")

    with gr.Row():
        past_out = gr.Dataframe(headers=["Product", "Aisle", "Department"], label="Past Purchases", interactive=False)
        rec_out = gr.Dataframe(headers=["Product", "Aisle", "Department", "Score"], label="Recommended Products", interactive=

    btn = gr.Button("Get Recommendations")
    btn.click(fn=recommend_with_history, inputs=[user_id_in, topk_in], outputs=[past_out, rec_out])

demo.launch(share=True)  # share=True for public link
```

```
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://98ee3c93e67de01756.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the workin
```



Past Purchases

Product	Aisle
Natural Artisan Water	water seltzer
Natural Artesian Water	water seltzer
Pure Coconut Water	juice nectars
Sparkling Water Grapefru	water seltzer

Recommended Products

Aisle	Department	Score
more household	household	2.891
crackers	snacks	2.788
energy sports drin	beverages	2.723
refrigerated	beverages	2.504