Fashion AI – Outfit Builder (Cross-sell) AI (Personal Stylist)

This notebook illustrates an **Outfit Builder (Cross-sell) AI use case** using the **Myntra Fashion Product Dataset** from Kaggle:
🔗 [Myntra Fashion Dataset on Kaggle]

Overview: Outfit Builder (Cross-sell) AI

The **Outfit Builder** is a personalized **fashion recommendation system** that mimics the role of a stylist. It enables intelligent cross-selling by suggesting complementary fashion items that go well together.

- **User Query Example:**
  "What goes well with this denim jacket?"
- **Retrieval:**
  - Myntra dataset was embedded using chromaDB (vector DB)
  - System fetches **candidate products** based on the intent clarification layer using LLM and embeds them using chroma vector database
- **Generation:**
  - AI suggests **styled outfit combinations** in natural language:
    "Pair this denim jacket with black skinny jeans and white sneakers for a casual street look. Or try it over a floral dress for a playful contrast."

Technical Implementation

The pipeline is organized into **modular layers** for interpretability and scalability:

1. Data Exploration – Know Your Data

- Understand the Myntra dataset structure: categories, attributes (color, style, price, material, etc.), product images, and text descriptions.
- Identify metadata useful for outfit pairing like p_attributes

2. Data Preparation – Chunking Strategies

- Clean text descriptions and metadata.
- Applied concatenation on product, name, description and p_attributes for embedding the text for search layers
- Ensure embeddings capture **textual features (concatenated columns)**.

3. Embedding Layer with Vector DB

- Convert the concatenated data into **dense vector representations** using `text-embedding-ada-002` model in chromaDB
- Store vectors in the chromaDB
- Enables fast **semantic search** of relevant items given a user query

4. User Query Embeddings Layer +Cache Implementation

- Parse user query (text).
- Using GPT 3.5 turbo model to understand the intent of the user query - To understand the intent of the user query like if it's for matching or finding an alternative for a piece cloth
- Implemented cache collections to not spend time on already worked on user queries. This helps to lesser the traffic on querying the historic requirements.
- Generate corresponding embeddings. Example: *"What goes well with a denim jacket?"* → embedding compared against DB to fetch relevant items.

5. Re-Ranking Layer

- Initial retrieval may fetch many candidates.
- Used a **cross-encoder (e.g., `cross-encoder/ms-marco-MiniLM-L-6-v2`)** to re-rank results.
- Ensures top items are semantically aligned with the **intent** (e.g., casual vs. formal outfits).

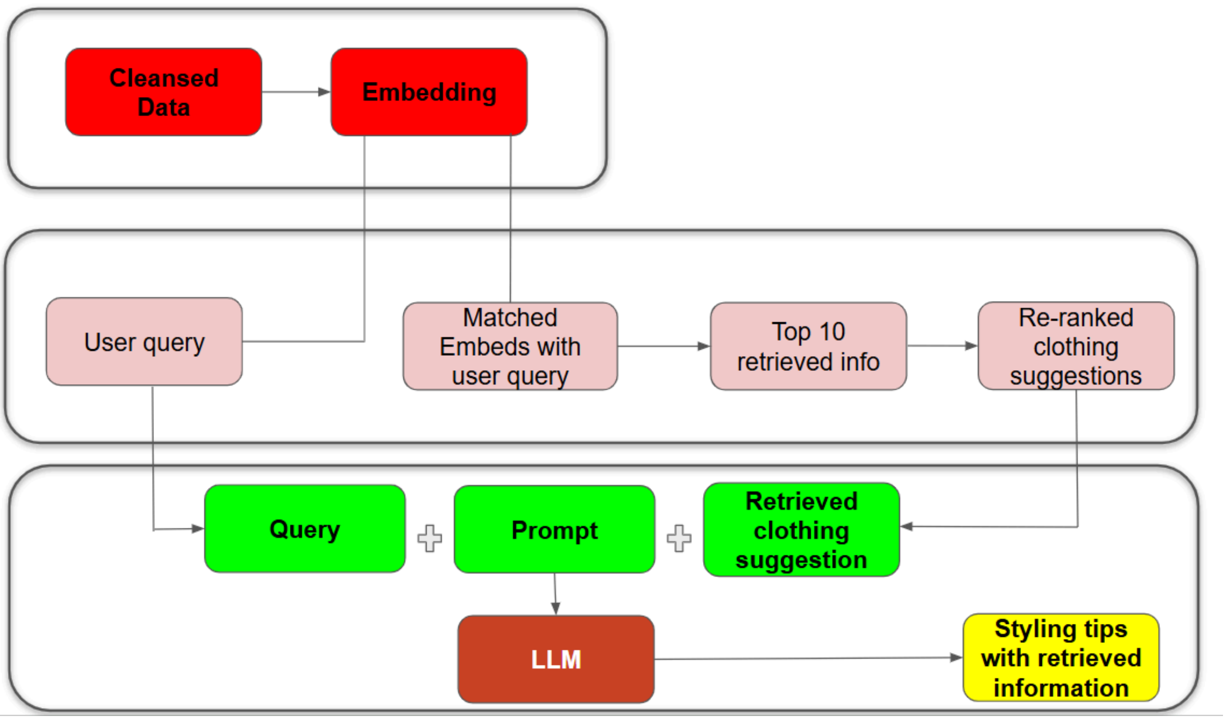
6. Generative Search Layer

- Use **LLM (e.g., GPT-4-0613)** to:
  - Synthesize results into natural outfit suggestions.
  - Incorporate **style attributes** (occasion, season, budget) with the picture displayed based on the search layer
  - Example Output:
    "Try styling your denim jacket with beige chinos and loafers for a smart-casual vibe. Alternatively, layer it over a hoodie with joggers for an athleisure look."

7. RAG Pipeline – End-to-End

- Combine all layers into a **Retrieval-Augmented Generation (RAG) pipeline**:
    - **Input:** User text or image query.
    - **Step 1:** Retrieve relevant product embeddings.
    - **Step 2:** Re-rank with semantic similarity.
    - **Step 3:** Generate outfit recommendations.
- Pipeline ensures **fast, relevant, stylist-like suggestions**.

Process workflow:



Cleansed Data
Embedding
User query
Matched Embeds with user query
Top 10 retrieved info
Re-ranked clothing suggestions
Query
Prompt
Retrieved clothing suggestion
LLM
Styling tips with retrieved information

Challenges and lessons learnt:

1. Concatenated text to be embedded was throwing an error due to the maximum token limit exceeded. To overcome that, I processed the embedding in batches of 100s
2. The objective of the search system is to assist the user to put together a good outfit. So adding an additional layer of intent clarification helped to define the search better. Without an intent clarification layer, the suggestions were not aligned with the user's request.
3. The final generative layer hallucinates with the first three products of the original dataset as the suggestion regardless of the user query statement. Changing the GPT model from 3.5 turbo to GPT 4o solved this problem.

The stress testing results, layer wise results are added as the screenshot in the folder along with the codes