FashionAI

Table of Contents

[Objective 2](#_Toc167817605)

[Dataset 2](#_Toc167817606)

[Playground 2](#_Toc167817607)

[Generative Search in Action 3](#_Toc167817608)

[System Design 3](#_Toc167817609)

[Layer 1: Build the vector store 3](#_Toc167817610)

[Layer 2: Search & Rerank Layer 5](#_Toc167817611)

[ Important Functions 5](#_Toc167817612)

[ SemanticSearchWithReranking() 6](#_Toc167817613)

[Layer 3: Generative Search 8](#_Toc167817614)

[ Important Functions 8](#_Toc167817615)

[ GenerativeSearch() 10](#_Toc167817616)

[Example experiments with different Queries 11](#_Toc167817617)

[**Query 1**: *“Maroon blazer for casual occasion from Van Heusen”* 11](#_Toc167817618)

[Sematic Search Result 11](#_Toc167817619)

[Generative Search result 12](#_Toc167817620)

[**Query 2**: *“White anarkali kurta with a rating of more than 4”* 13](#_Toc167817621)

[Sematic Search Result 13](#_Toc167817622)

[Generative Search result 14](#_Toc167817623)

[**Query 3**: *“Denim jacket with atleast 4 pockets, a rating of 4 and price less than 3000”* 15](#_Toc167817624)

[Sematic Search Result 15](#_Toc167817625)

[Generative Search result 16](#_Toc167817626)

[Conclusion 17](#_Toc167817627)

[Lessons learned 17](#_Toc167817628)

[Challenges Faced 17](#_Toc167817629)

## **Objective**

The goal is to develop an advanced search system that can intelligently sift through a vast collection of fashion product descriptions. This system should be able to understand a user's query and recommend the most suitable fashion items based on their preferences.

## **Dataset**

The dataset used for this project is [Myntra Fashion Product Dataset (kaggle.com)](https://www.kaggle.com/datasets/djagatiya/myntra-fashion-product-dataset). It includes thousands of fashion products, each with detailed information such as description, price, rating, brand name, and other attributes.

The dataset is organized as follows:

1. **Fashion Dataset v2.csv:** This CSV file contains each product in a row, with various details spread across columns.
2. **Image Folder:** This folder contains sample images for each product. Each image file is named according to the corresponding product's 'p\_id' value from the CSV.

## **Playground**

To experiment with the search tool, follow these steps:

**1. Open the Jupyter Notebook**: Ensure you have the Jupyter Notebook for this project open and run all the cells for the first time after launch.

**2. Navigate to the Playground Section:** Scroll to the end of the notebook until you find the section titled "Playground" or jump from the Table of Contents.

**3. Interact with the Search Tool:** In this section, you can test and experiment with the search tool. You can input various user queries to see how the system retrieves and recommends fashion products based on the descriptions, prices, ratings, brand names, and other attributes from the dataset.

In this section, two cells are provided to experiment with Semantic Search and Generative Search.

A screenshot of a computer

Description automatically generated

**4. Analyse Results:** Review the recommended products to understand how well the search tool matches the user query. This will help in assessing the accuracy and effectiveness of the search algorithm.

*> Make sure to set the OpenAI Api key in ‘OpenAI\_API\_Key.txt’ file <*

Feel free to experiment with different queries and explore different aspects of the dataset to fully understand the capabilities of the search system.

## **Generative Search in Action**

## System Design

**A screenshot of a computer screen

Description automatically generated**

### Layer 1: Build the vector store

1. **Loading and Exploring the CSV File**: The CSV file containing fashion product data is loaded and examined to understand its structure and contents.
2. **Exclusion of 'img' Column**: The 'img' column, which contains URIs for product images, is deemed unnecessary since the images are already available in the 'dataset/images' folder, named according to the 'p\_id' column.
3. **Extracting Description Data**: The 'description' column contains data in HTML format. This HTML data needs to be converted into plain text strings using Python.
4. **Portfolio:** The product information is currently fragmented across various columns, hindering comprehensive keyword searches. - Since our query focuses solely on documents, this fragmentation poses limitations.

There are two ways to address this:

**Option 1**: Centralize all the product details as metadata. Develop a chatbot to extract metadata values, facilitating precise filtering. Utilize the filtered results to construct a RAG model.

**Option 2**: Compile a cohesive document called as 'portfolio' incorporating all product details. Load this document entirely into the document section of the vector DB. This ensures vector DB is aware of additional keywords used in user queries.

* For the present task, I opted for option 2 for enhanced efficiency.

1. **Analysing Portfolio Lengths**: The maximum sentence length in the descriptions is found to be **2764** characters, and the maximum number of portfolios is less *than 1833* characters.

Since the maximum input lengths in our dataset are well within this limit, the model is suitable and there is *no need to split* (chunk) the product descriptions.

1. **Creating Metadata**: Metadata is created, containing only the 'p\_id' value to facilitate displaying the corresponding product image.
2. **Selecting a Suitable Embedding Model**: After researching, the model " [nomic-ai/nomic-embed-text-v1.5 · Hugging Face](https://huggingface.co/nomic-ai/nomic-embed-text-v1.5)" is chosen due to its good MTEB results of **62.28** and its support for text lengths up to **8192** characters.
3. **Storing Data in ChromaDB**: After completing the above steps, a collection named *'ProductDetails'* is created in ChromaDB. All product information along with the corresponding metadata is added to this collection.

This structured approach ensures the data is properly prepared, relevant columns are identified, and an appropriate model is selected for embedding the text data. Finally, the data is efficiently stored in ChromaDB for subsequent retrieval and use in the search system.

### Layer 2: Search & Rerank Layer

Layer 1 needs to be executed only for the first time and when we have a new data. From this layer, we retrieve the top 10 results for the query from the vector db and are reranked using cross encoder.

Setup a cache collection named *'ProductsCache'* in Chroma DB using the client.

#### Important Functions

* ***query\_from\_main\_or\_cache\_collection()***

The function *query\_from\_main\_or\_cache\_collection* is designed to efficiently search for query results by checking a cache before querying the main collection. Here is a step-by-step breakdown:

**1. Initialization:**

- Initialize empty lists (*ids, documents, distances, metadatas*) and an empty DataFrame (*results\_df*) to store query results.

**2. Cache Search:**

- Attempt to retrieve the query result from the *cache\_collection*.

- If the cache is empty or the retrieved result's distance exceeds a predefined *threshold(0.2)* , proceed to query the main collection.

**3. Main Collection Query:**

- Query the ‘*ProductDetails’* collection for the **top** **10** results.

- Store the query in the cache collection as a document for future reference, along with its corresponding results (*texts, ids, distances, and metadatas*).

- Construct a DataFrame (*results\_df*) from the main collection query results.

**4. Cache Result Handling:**

- If the cache contains a valid result within the distance threshold, extract the result data.

- Append the extracted data (*ids, documents, distances, metadatas*) to the respective lists.

- Create a DataFrame (*results\_df*) from the cached query results.

**5. Return Results:** Return the constructed DataFrame (*results\_df*) containing the query results.

This approach optimizes search efficiency by utilizing cached results when available and only querying the main collection when necessary.

* ***Improving Search Results with Cross-Encoder Reranking()***

Re-ranking the results obtained from a semantic search can often significantly enhance the relevance of the retrieved results. This process involves using a cross-encoder, which evaluates the relevance of each response in relation to the query. In this task, cross encoder model used is *‘****ms-marco-MiniLM-L-6-v2****’*.  
Here’s how this process works:

**1. Initial Retrieval:**

- Perform a semantic search to retrieve an initial set of results based on the query. This step quickly identifies potentially relevant documents.

**2. Cross-Encoder Scoring:**

- For re-ranking, pair the query with each of the retrieved responses.

- Pass each query-response pair into a cross-encoder model, which jointly processes the pair to capture the detailed interactions and context between them.

- The cross-encoder scores the relevance of each response with respect to the query, providing a refined relevance score.

**3. Re-ranking:**

- Use the relevance scores from the cross-encoder to reorder the initial set of retrieved results.

- Responses with higher relevance scores are moved to the top of the list, ensuring the most relevant documents are prioritized.

By leveraging the cross-encoder's ability to understand nuanced relationships between the query and each response, this re-ranking process leads to more accurate and contextually relevant search results.

#### SemanticSearchWithReranking()

This is the final function of this layer which aims to perform a semantic search with an additional step of re-ranking the results using a cross-encoder model.

Here's a detailed explanation of how it works:

**1. Query Input:**

- If no query is provided (query=None), the function prompts the user to enter a query.

**2. Initial Search:**

- The function calls *query\_from\_main\_or\_cache\_collection(query)* to retrieve an initial set of **top 10** semantic search results. This function queries either a main collection or a cache to get the results, which are returned in a DataFrame (*query\_results\_df*).

**3. Preparing Inputs for Re-Ranking:**

- The function prepares inputs for the cross-encoder model by pairing the query with each of the retrieved responses. This is done by creating a list of lists (*cross\_inputs*), where each inner list contains the query and a response document.

**4. Re-Ranking with Cross-Encoder:**

- The cross-encoder model (*cross\_encoder*) is used to predict relevance scores for each query-response pair. These scores reflect how relevant each response is to the query.

- The predicted scores are stored in the DataFrame (*query\_results\_df*) under a new column called ‘*Reranked\_scores’*.

**5. Selecting Top Results:**

- The DataFrame is sorted by the re-ranked scores in descending order to prioritize the most relevant responses.

- The function selects the top n entries (top\_n) based on these scores, where top\_n defaults to 3.

**6. Return Top Results:**

- The top 3 re-ranked results are returned as the final output of the function.

**Summary**

This function enhances the relevance of search results by:

- Performing an initial semantic search to get a broad set of results.

- Using a cross-encoder model to re-evaluate and score these results based on their relevance to the query.

- Returning the top 3 results based on the refined relevance scores.

This two-step process ensures that the results are both comprehensive and highly relevant to the user's query.

### Layer 3: Generative Search

Once we have obtained the final top semantic search results with re-ranking, we can enhance the user experience by leveraging GPT-3.5 to generate a direct answer to the query.

#### Important Functions

* ***generate\_response()***

This function utilizes OpenAI's GPT-3.5's ChatCompletion model to generate a response to a user query about fashion products. It considers the user's role, the query, and the search results from a DataFrame, and follows specific guidelines to ensure the generated response is relevant, informative, and compliant with the provided instructions.

**Input Parameters**:

*query*: The user's query for which a response needs to be generated.

*results\_df:* A DataFrame containing search results from a corpus of fashion products. This DataFrame serves as the basis for generating the response.

**Output**:

Once the response is generated by the ChatCompletion model, the function extracts the content of the response and splits it into lines using the newline character ('\n').

The final response is returned as a list of lines, making it easily readable and accessible for further processing or display.

* ***display\_image()***

This function is designed to display an image corresponding to a product result from the provided DataFrame (*result\_df*). Let's break down its functionality:

**1. Input Parameter:**

- *result\_df*: A DataFrame containing information about the product result, including metadata *‘p\_id’*

**2. Metadata Extraction:**

- The function extracts the metadata associated with the product result from the DataFrame. This metadata typically includes *‘p\_id’*, which is needed to locate the corresponding image file.

**3. Metadata Conversion:**

- The extracted metadata, represented as a string, is converted to a dictionary format using the *ast.literal\_eval* function. This allows easier access to specific metadata values.

**4. Image Number Extraction:**

- From the metadata dictionary, the function retrieves the image number (*'p\_id'*) associated with the product result.

**5. Image Path Construction:**

- Using the image number, the function constructs the file path to the corresponding image file. The path is typically based on a predefined directory structure where images are stored.

**6. Image Display:**

- The function utilizes the display function to showcase the image in the Jupyter Notebook environment. The image is loaded from the constructed file path and displayed with a specified width of "250" pixels.

*Additional Notes:*

- The images are stored in a directory named "*dataset/images*" and follow a specific naming convention, where the image file name corresponds to the ‘*p\_id*’ of the product followed by the ".jpg" extension.

- It's important to ensure that the DataFrame contains accurate metadata, especially the ‘p\_id’, to successfully locate and display the corresponding image.

Thus, the function enables users to visualize the products associated with the search results, enhancing the understanding and interaction with the data.

* ***print\_response\_and\_display\_image()***

The function serves as a convenient utility for presenting the generated response and displaying the associated image of a suggested product. By combining textual information with visual representation, this function enhances the user experience and facilitates better understanding and interaction with the suggested products.

*Input:*

- *result\_df*: This parameter represents the DataFrame containing information about the suggested product, including metadata such as 'p\_id'

- *response*: This parameter contains the generated response that needs to be printed.

- *result\_text*: This parameter specifies the text to be used as a header before printing the response. It defaults to "Response:" if not provided.

#### GenerativeSearch()

The final function of this layer performs a generative search by combining semantic search with reranking and generating a response.

Let's delve into its functionality:

**1. Input Parameter:**

- *query*: This parameter allows the user to input a query directly. If no query is provided, the function prompts the user to input one via the console.

**2. Semantic Search with Reranking:**

- The function calls the *SemanticSearchWithReranking* function to perform semantic search with reranking for the given query. It retrieves the **top 3** most relevant results and stores them in a DataFrame named *semantic\_search\_df*. The index of the DataFrame is reset to ensure proper iteration.

**3. Response Generation and Display:**

- The function iterates over each result in the semantic\_search\_df DataFrame.

- For the first result (index 0), it generates a response using the *generate\_response* function and prints the response along with displaying the corresponding image using the *print\_response\_and\_display\_image* function. This result is labelled as the "Top product".

- For subsequent results, it checks if the current result's reranked score is close enough to the previous result's reranked score. If it is within a certain threshold (0.5 in this case), it generates a response and displays the corresponding image, labelling the result as a "Similar product". If the rerank score difference is larger than the threshold, the loop is terminated to avoid suggesting further similar products.

**4. Previous Reranked Score Tracking:**

- The function keeps track of the reranked score of the previous result to compare it with the reranked score of the current result in the next iteration. This allows for determining the similarity between consecutive results.

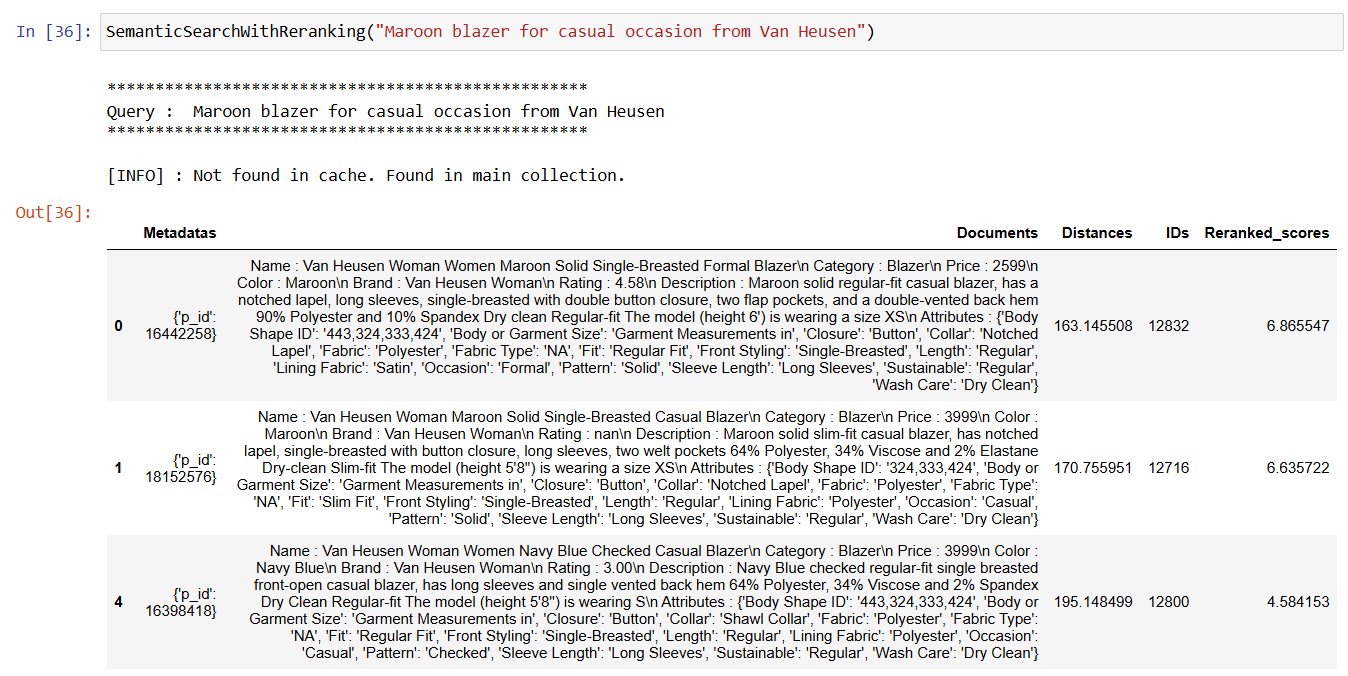
***Summary:***

The **GenerativeSearch** function is the main function of this whole search system which orchestrates the process of performing generative search by integrating semantic search with reranking and response generation. It iterates over the top results obtained from semantic search, generates responses based on the query and retrieved information, and displays the corresponding images. This iterative process provides users with relevant product suggestions and allows for exploration of similar products based on reranked scores.

## **Example experiments with different Queries**

### **Query 1**: *“Maroon blazer for casual occasion from Van Heusen”*

#### Sematic Search Result



#### Generative Search result

A person in a red jacket

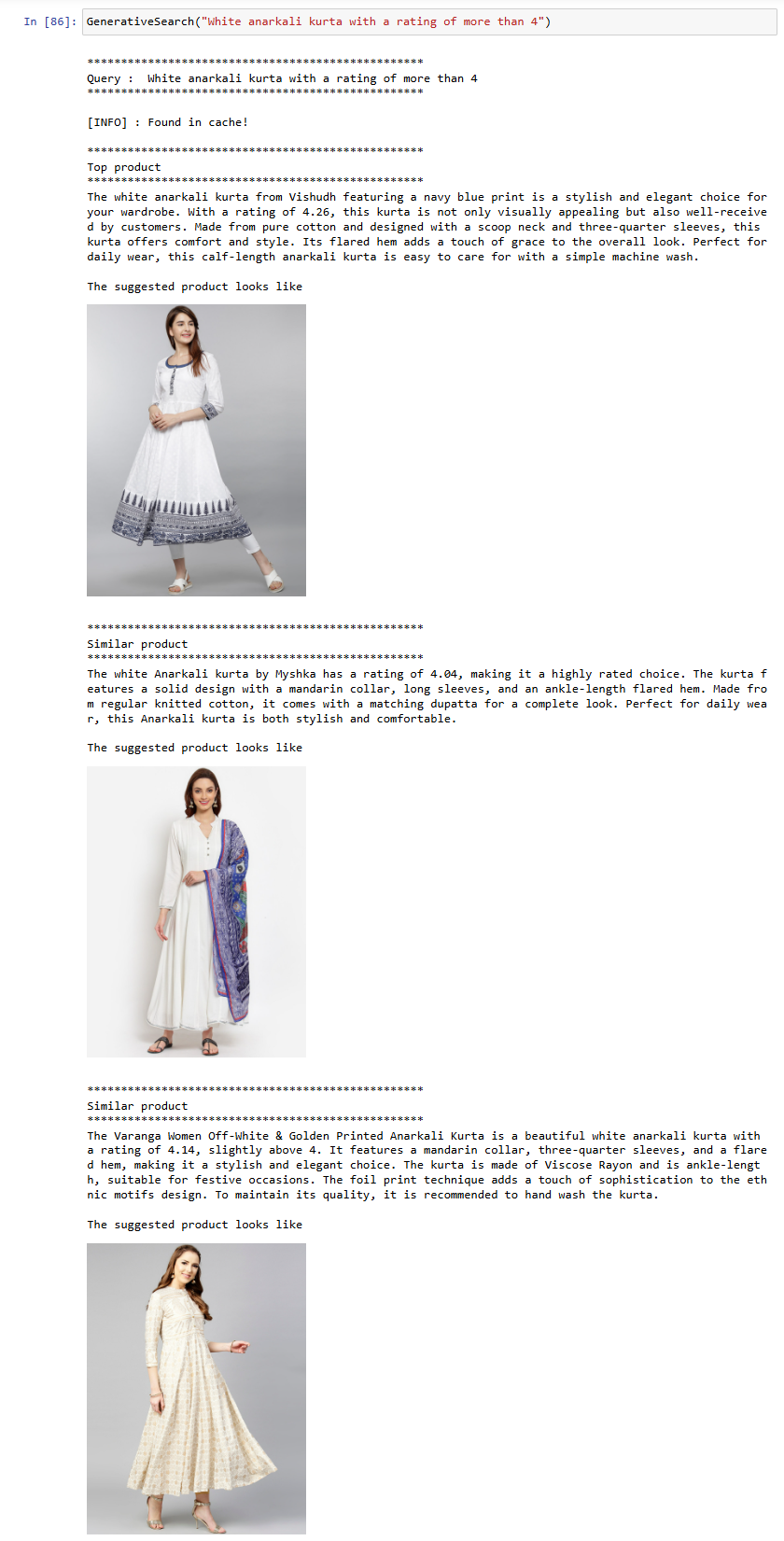
Description automatically generated

### **Query 2**: *“White anarkali kurta with a rating of more than 4”*

#### Sematic Search Result



#### Generative Search result

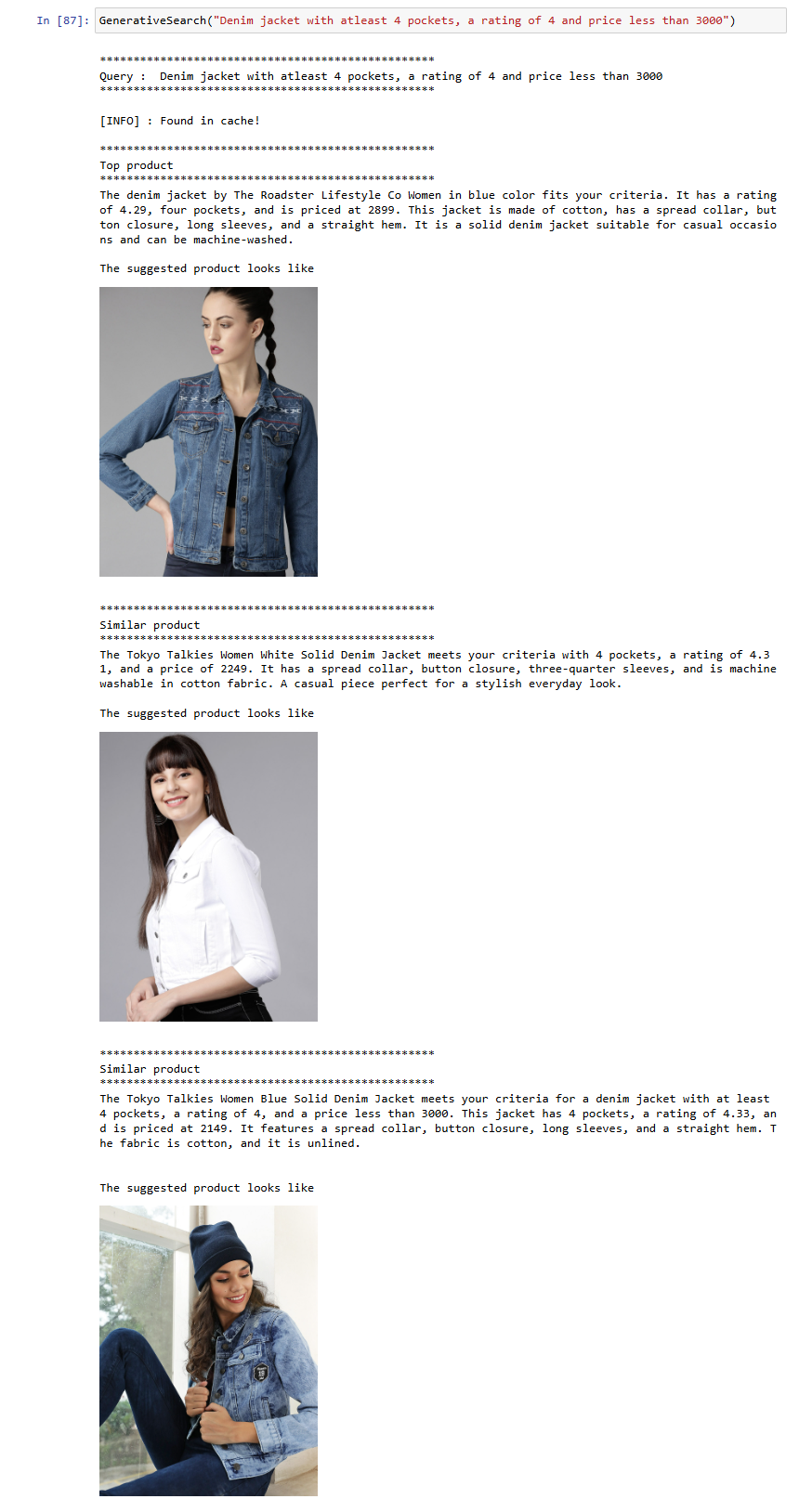


### **Query 3**: *“Denim jacket with atleast 4 pockets, a rating of 4 and price less than 3000”*

#### Sematic Search Result



#### Generative Search result



## Conclusion

Finally, with the function GenerativeSearch(), user can input a query to fetch a product from the vast database using natural language.

### Lessons learned

* Explored various models to implement embedding.
* Explored various ways to prepare the document for query.
* Explored different prompts to generate appropriate response for the user query.
* Implemented cache technique for faster response.
* Hands-on experience with Chroma DB.
* Explored various datasets to experiment with RAG apart from this fashion dataset.

### Challenges Faced

* In this scenario, I did not come with the use of chunking a single product data. This is an area to explore for the same problem statement.
* When using OpenAI embedding function, faced an error with ‘$’ character input. But I did not choose the model anyways since I could find something else with similar performance.
* There is still a room to explore the search query. For an example, if the user queries for “red t-shirt” it always gives the same set of results.