Assignment 5: Clothing Match Maker Assistant using RAG with GPT4o-Mini

Welcome to the Clothing Matchmaker App Jupyter Notebook! This project demonstrates the power of the GPT4o-Mini model in analyzing images of clothing items and extracting key features such as color, style, and type. The core of our app relies on this advanced image analysis model developed by OpenAI, which enables us to accurately identify the characteristics of the input clothing item.

GPT4o-Mini is a model that combines natural language processing with image recognition, allowing it to understand and generate responses based on both text and visual inputs.

Building on the capabilities of the GPT4o-Mini model, we employ a custom matching algorithm and the RAG technique to search our knowledge base for items that complement the identified features. This algorithm takes into account factors like color compatibility and style coherence to provide users with suitable recommendations. Through this notebook, we aim to showcase the practical application of these technologies in creating a clothing recommendation system.

Using the combination of GPT4o-Mini + RAG (Retrieval-Augmented Generation) offers several advantages:

1. **Contextual Understanding**: GPT4o-Mini can analyze input images and understand the context, such as the objects, scenes, and activities depicted. This allows for more accurate and relevant suggestions or information across various domains, whether it's interior design, cooking, or education.
2. **Rich Knowledge Base**: RAG combines the generative capabilities of GPT-4 with a retrieval component that accesses a large corpus of information across different fields. This means the system can provide suggestions or insights based on a wide range of knowledge, from historical facts to scientific concepts.
3. **Customization**: The approach allows for easy customization to cater to specific user needs or preferences in various applications. Whether it's tailoring suggestions to a user's taste in art or providing educational content based on a student's learning level, the system can be adapted to deliver personalized experiences.

Overall, the GPT4o-Mini + RAG approach offers a powerful and flexible solution for various fashion-related applications, leveraging the strengths of both generative and retrieval-based AI techniques.

Environment Setup

First we will install the necessary dependencies, then import the libraries and write some utility functions that we will use later on.

```
# pip install tenacity tqdm numpy typing tiktoken concurrent
```

```
Running cells with 'venv (Python 3.8.5)' requires the ipykernel
package.

Run the following command to install 'ipykernel' into the Python
environment.

Command:
'c:/Users/riman/VSCode/GenAI_Projects/Assignments/venv/Scripts/python.
exe -m pip install ipykernel -U --force-reinstall'
```

```
# # use approprite command if curl is not installed to download docs,
option: wget
# !curl -O https://raw.githubusercontent.com/anshupandey/Generative-
AI-for-Professionals/main/datasets/sample_clothes.zip
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:--
--:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:01
--:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:02
--:--:--     0
  0 30.0M    0 32768    0     0  12123      0  0:43:20  0:00:02
0:43:18 12140
 99 30.0M   99 29.7M    0     0  9194k      0  0:00:03  0:00:03
--:--:-- 9205k
100 30.0M  100 30.0M    0     0  9281k      0  0:00:03  0:00:03
--:--:-- 9292k
```

```
import zipfile

with zipfile.ZipFile("sample_clothes.zip", 'r') as zip_ref:
    zip_ref.extractall("sample_clothes")  # Creates a folder and
extracts files
```

Initializes a connection to Azure OpenAI service for chat completions.

Configuration

- Client Type: AzureChatOpenAI
- Endpoint: Azure-hosted OpenAI service at clothassistant8618180812.openai.azure.com
- Authentication: API key retrieved securely from Azure Key Vault
- API Version: 2024-12-01-preview
- Model: Uses gpt-4 model for chat completions

```
import pandas as pd
import numpy as np
import json
import ast
import tiktoken
import concurrent
from langchain_openai import AzureChatOpenAI
from tqdm import tqdm
from tenacity import retry, wait_random_exponential,
stop_after_attempt
from IPython.display import Image, display, HTML
from typing import List
import os


GPT_MODEL = "gpt-4o-mini"
EMBEDDING_MODEL = "text-embedding-ada-002"
EMBEDDING_COST_PER_1K_TOKENS = 0.00013
curr_path = os.getcwd()
# Make sure to initialize env variables: for azure openai endpoint,
key, api version


# write code to initialize the Azure OpenAI Client
```

Creating the Embeddings

We will now set up the knowledge base by choosing a database and generating embeddings for it. I am using the `sample_styles.csv` file for this in the data folder. This is a sample of a bigger dataset that contains ~44K items. This step can also be replaced by using an out-of-the-box vector database. For example, you can follow one of these cookbooks to set up your vector database.

```
styles_filepath = os.path.join(curr_path, "sample_clothes",
"sample_clothes", "sample_styles.csv")
styles_df = pd.read_csv(styles_filepath, on_bad_lines='skip')
print(styles_df.head())
print("Opened dataset successfully. Dataset has {} items of
clothing.".format(len(styles_df)))
```

```
      id gender masterCategory subCategory articleType baseColour
season  \
0  27152    Men        Apparel     Topwear      Shirts       Blue
Summer
1  10469    Men        Apparel     Topwear     Tshirts     Yellow
Fall
```

```
2  17169    Men        Apparel     Topwear      Shirts     Maroon
Fall
3  56702    Men        Apparel     Topwear      Kurtas       Blue
Summer
4  47062  Women        Apparel  Bottomwear     Patiala      Multi
Fall

     year   usage                        productDisplayName
0  2012.0  Formal       Mark Taylor Men Striped Blue Shirt
1  2011.0  Casual   Flying Machine Men Yellow Polo Tshirts
2  2011.0  Casual  U.S. Polo Assn. Men Checks Maroon Shirt
3  2012.0  Ethnic                 Fabindia Men Blue Kurta
4  2012.0  Ethnic        Shree Women Multi Colored Patiala
Opened dataset successfully. Dataset has 1000 items of clothing.
```

Now we will generate embeddings for the entire dataset. We can parallelize the execution of these embeddings to ensure that the script scales up for larger datasets. With this logic, the time to create embeddings for the full `44K` entry dataset decreases from ~4h to ~2-3min.

```python
## Batch Embedding Logic

@retry(wait=wait_random_exponential(min=1, max=40),
stop=stop_after_attempt(10))
def get_embeddings(input: list):
    try:
        # write code to use openai client and create embeddings, use
variable 'response' to store embeddings




        return [data.embedding for data in response]
    except Exception as e:
        print(f"Error in get_embeddings: {str(e)}")
        raise


# Splits an iterable into batches of size n.
def batchify(iterable, n=1):
    l = len(iterable)
    for ndx in range(0, l, n):
        yield iterable[ndx : min(ndx + n, l)]


# Function for batching and parallel processing the embeddings
def embed_corpus(
    corpus: List[str],
    batch_size=64,
    num_workers=8,
```

```
    max_context_len=8191,
):
    # Encode the corpus, truncating to max_context_len
    encoding = tiktoken.get_encoding("cl100k_base")
    encoded_corpus = [
        encoded_article[:max_context_len] for encoded_article in
encoding.encode_batch(corpus)
    ]

    # Calculate corpus statistics: the number of inputs, the total
number of tokens, and the estimated cost to embed
    num_tokens = sum(len(article) for article in encoded_corpus)
    cost_to_embed_tokens = num_tokens / 1000 *
EMBEDDING_COST_PER_1K_TOKENS
    print(
        f"num_articles={len(encoded_corpus)}, num_tokens={num_tokens},
est_embedding_cost={cost_to_embed_tokens:.2f} USD"
    )

    # Embed the corpus
    with
concurrent.futures.ThreadPoolExecutor(max_workers=num_workers) as
executor:

        futures = [
            executor.submit(get_embeddings, text_batch)
            for text_batch in batchify(encoded_corpus, batch_size)
        ]

        with tqdm(total=len(encoded_corpus)) as pbar:
            for _ in concurrent.futures.as_completed(futures):
                pbar.update(batch_size)

        embeddings = []
        for future in futures:
            data = future.result()
            embeddings.extend(data)

        return embeddings


# Function to generate embeddings for a given column in a DataFrame
def generate_embeddings(df, column_name):
    # Initialize an empty list to store embeddings
    descriptions = df[column_name].astype(str).tolist()
    embeddings = embed_corpus(descriptions)

    # Add the embeddings as a new column to the DataFrame
    df['embeddings'] = embeddings
    print("Embeddings created successfully.")
```

Two options for creating the embeddings:

The next line will **create the embeddings** for the sample clothes dataset. This will take around 0.02s to process and another ~30s to write the results to a local .csv file. The process is using our `text-embedding-ada-002` model which is priced at `$0.00013/1K` tokens. Given that the dataset has around `1K` entries, the following operation will cost approximately `$0.001`. If you decide to work with the entire dataset of `44K` entries, this operation will take 2-3min to process and it will cost approximately `$0.07`.

If you would not like to proceed with creating your own embeddings, we will use a dataset of pre-computed embeddings. You can skip this cell and uncomment the code in the following cell to proceed with loading the pre-computed vectors. This operation takes ~1min to load all the data in memory.

```
generate_embeddings(styles_df, 'productDisplayName')
print("Writing embeddings to file ...")
styles_df.to_csv('sample_clothes/sample_styles_with_embeddings.csv',
index=False)
print("Embeddings successfully stored in
sample_styles_with_embeddings.csv")
```

```
num_articles=1000, num_tokens=8280, est_embedding_cost=0.00 USD

1024it [00:08, 120.35it/s]

Embeddings created successfully.
Writing embeddings to file ...
Embeddings successfully stored in sample_styles_with_embeddings.csv
```

```
# styles_df =
pd.read_csv('sample_clothes/sample_styles_with_embeddings.csv',
on_bad_lines='skip')

# # Convert the 'embeddings' column from string representations of
lists to actual lists of floats
# styles_df['embeddings'] = styles_df['embeddings'].apply(lambda x:
ast.literal_eval(x))

print(styles_df.head())
print("Opened dataset successfully. Dataset has {} items of clothing
along with their embeddings.".format(len(styles_df)))
```

```
      id gender masterCategory subCategory articleType baseColour
season  \
0  27152    Men        Apparel     Topwear      Shirts       Blue
Summer
1  10469    Men        Apparel     Topwear     Tshirts     Yellow
Fall
2  17169    Men        Apparel     Topwear      Shirts     Maroon
Fall
3  56702    Men        Apparel     Topwear      Kurtas       Blue
```

```
Summer
4  47062  Women        Apparel  Bottomwear     Patiala      Multi
Fall

     year   usage                        productDisplayName  \
0  2012.0  Formal       Mark Taylor Men Striped Blue Shirt
1  2011.0  Casual   Flying Machine Men Yellow Polo Tshirts
2  2011.0  Casual  U.S. Polo Assn. Men Checks Maroon Shirt
3  2012.0  Ethnic                  Fabindia Men Blue Kurta
4  2012.0  Ethnic        Shree Women Multi Colored Patiala

                                            embeddings
0  [0.006926022004336119, 0.00024994637351483107,...
1  [-0.04372986778616905, -0.008873915299773216, ...
2  [-0.02801556885242462, 0.05883694440126419, -0...
3  [-0.0036652961280196905, 0.02953275293111801, ...
4  [-0.05233072489500046, 0.01595483161509037, -0...
Opened dataset successfully. Dataset has 1000 items of clothing along
with their embeddings.
```

Building the Matching Algorithm

In this section, we'll develop a cosine similarity retrieval algorithm to find similar items in our dataframe. We'll utilize our custom cosine similarity function for this purpose. While the `sklearn` library offers a built-in cosine similarity function, recent updates to its SDK have led to compatibility issues, prompting us to implement our own standard cosine similarity calculation.

If you already have a vector database set up, you can skip this step. Most standard databases come with their own search functions, which simplify the subsequent steps outlined in this guide. However, we aim to demonstrate that the matching algorithm can be tailored to meet specific requirements, such as a particular threshold or a specified number of matches returned.

The `find_similar_items` function accepts four parameters:

- `embedding`: The embedding for which we want to find a match.
- `embeddings`: A list of embeddings to search through for the best matches.
- `threshold` (optional): This parameter specifies the minimum similarity score for a match to be considered valid. A higher threshold results in closer (better) matches, while a lower threshold allows for more items to be returned, though they may not be as closely matched to the initial `embedding`.
- `top_k` (optional): This parameter determines the number of items to return that exceed the given threshold. These will be the top-scoring matches for the provided `embedding`.

```python
def cosine_similarity_manual(vec1, vec2):
    """Calculate the cosine similarity between two vectors."""

    # implement the function which returns cosine similarity between
two vectors.
    raise NotImplementedError("This function is not yet implemented.")
# remove this line
```

```
def find_similar_items(input_embedding, embeddings, threshold=0.5,
top_k=2):
    """Find the most similar items based on cosine similarity."""

    # Calculate cosine similarity between the input embedding and all
other embeddings
    similarities = [(index, cosine_similarity_manual(input_embedding,
vec)) for index, vec in enumerate(embeddings)]

    # Filter out any similarities below the threshold
    filtered_similarities = [(index, sim) for index, sim in
similarities if sim >= threshold]

    # Sort the filtered similarities by similarity score
    sorted_indices = sorted(filtered_similarities, key=lambda x: x[1],
reverse=True)[:top_k]

    # Return the top-k most similar items
    return sorted_indices

def find_matching_items_with_rag(df_items, item_descs):
   """Take the input item descriptions and find the most similar items
based on cosine similarity for each description."""

   # Select the embeddings from the DataFrame.
   embeddings = df_items['embeddings'].tolist()


   similar_items = []
   for desc in item_descs:

      # Generate the embedding for the input item
      input_embedding = get_embeddings([desc])

      # Find the most similar items based on cosine similarity
      similar_indices = find_similar_items(input_embedding,
embeddings, threshold=0.6)
      similar_items += [df_items.iloc[i] for i in similar_indices]

   return similar_items
```

Analysis Module

In this module, we leverage `GPT4o-Mini` to analyze input images and extract important features like detailed descriptions, styles, and types. The analysis is performed through a

straightforward API call, where we provide the URL of the image for analysis and request the model to identify relevant features.

To ensure the model returns accurate results, we use specific techniques in our prompt:

1. **Output Format Specification**: We instruct the model to return a JSON block with a predefined structure, consisting of:
   - `items` (str[]): A list of strings, each representing a concise title for an item of clothing, including style, color, and gender. These titles closely resemble the `productDisplayName` property in our original database.
   - `category` (str): The category that best represents the given item. The model selects from a list of all unique `articleTypes` present in the original styles dataframe.
   - `gender` (str): A label indicating the gender the item is intended for. The model chooses from the options `[Men, Women, Boys, Girls, Unisex]`.
2. **Clear and Concise Instructions**:
   - We provide clear instructions on what the item titles should include and what the output format should be. The output should be in JSON format, but without the `json` tag that the model response normally contains.
3. **One Shot Example**:
   - To further clarify the expected output, we provide the model with an example input description and a corresponding example output. Although this may increase the number of tokens used (and thus the cost of the call), it helps to guide the model and results in better overall performance.

By following this structured approach, we aim to obtain precise and useful information from the `GPT4o-Mini` model for further analysis and integration into our database.

```
from langchain_core.messages import HumanMessage

def analyze_image(image_base64, subcategories):
    # Prepare the prompt and image inputs for AzureChatOpenAI
    messages = [
        HumanMessage(
            content=[
                {
                    "type": "text",
                    "text": f"""Given an image of an item of clothing,
analyze the item and generate a JSON output with the following fields:
"items", "category", and "gender".
                            Use your understanding of fashion trends,
styles, and gender preferences to provide accurate and relevant
suggestions for how to complete the outfit.
                            The items field should be a list of items
that would go well with the item in the picture. Each item should
represent a title of an item of clothing that contains the style,
color, and gender of the item.
                            The category needs to be chosen between the
types in this list: {subcategories}.
```

```
                            You have to choose between the genders in
this list: [Men, Women, Boys, Girls, Unisex]
                            Do not include the description of the item
in the picture. Do not include the ```json ``` tag in the output.

                            Example Input: An image representing a
black leather jacket.

                            Example Output: {{"items": ["Fitted White
Women's T-shirt", "White Canvas Sneakers", "Women's Black Skinny
Jeans"], "category": "Jackets", "gender": "Women"}}"""
                },
                {
                    "type": "image_url",
                    "image_url": f"
{image_base64}"
                }
            ]
        )
    ]

    # Call the AzureChatOpenAI client to fetch send the messages and
get response and return the response
```

Testing the Prompt with Sample Images

To evaluate the effectiveness of our prompt, let's load and test it with a selection of images from our dataset. We'll use images from the `"data/sample_clothes/sample_images"` folder, ensuring a variety of styles, genders, and types. Here are the chosen samples:

- `2133.jpg`: Men's shirt
- `7143.jpg`: Women's shirt
- `4226.jpg`: Casual men's printed t-shirt

By testing the prompt with these diverse images, we can assess its ability to accurately analyze and extract relevant features from different types of clothing items and accessories.

We need a utility function to encode the .jpg images in base64

```
import base64

def encode_image_to_base64(image_path):
    with open(image_path, 'rb') as image_file:
        encoded_image = base64.b64encode(image_file.read())
        return encoded_image.decode('utf-8')

# Set the path to the images and select a test image
image_path = os.path.join(curr_path, "sample_clothes",
"sample_clothes", "sample_images")
test_images = ["2133.jpg", "7143.jpg", "4226.jpg"]

# Encode the test image to base64
reference_image = os.path.join(image_path, test_images[0])
encoded_image = encode_image_to_base64(reference_image)
encoded_image
```


```
'/9j/4AAQSkZJRgABAQAAZABkAAD/7AARRHVja3kAAQAEAAAAZAAA/
9sAQwABAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQECAgEBAgEBAQICAgI
CAgICAgECAgICAgICAgIC/
9sAQwEBAQEBAQEBAgEBAQICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgI
CAgICAgICAgICAgICAgIC/8AAEQgAUAA8AwERAAIRAQMRAf/
EAB0AAAEEAwEBAAAAAAAAAAAAAAAoABwgJAQULBgL/
xAAxEAABBAICAgECBAUEAwAAAAACAQMEBQYHCBEAEgkTIQoUFTFBUWFxkTJygaEiI0L/
xAAcAQABBAMBAAAAAAAAAAAAAAAABAUGBwIDCAH/
xAA4EQABAwIEBAMFBgYDAAAAAAABAgMRAAQFEiExBkFRYRNxgQcUIpGxIzJCoeHwM1Jigs
HRFXKy/9oADAMBAAIRAxEAPwA/
jwopeFFeK2PsTCtR4Dme0NjZDX4ngWvsYu8xzDJLQzCBSY5j1e/
Z29lI+mBG4LUOO6qNtibrp+rTQG4YCvqUlSglIkqrwmBNCzO/
i2eIr2yskxWs0fsqRgNfk7Nfh+zZV3XQo2c4sIGszI0xmVUty8Qle5B9GLPU2+myGZKimX
TbiiwQoEqfymNDllJPYzMDmYohekAeU/XT/
ca70TlobeOtuSundd731Bft5NrbaGMwcqxS5BpWDfgTPqNPRZkYiVYdpFnsTIkxhVUmJUF
5klVQ7VA4hTa1IVunpt2I7Eaivdeehp3PMKKXhRS8KKXhRS8KKG+/
FDYrsbLvjxw2nwvMMgx/
F7DlDqGt25RU9mFNAzDX02Jlbo12STOveXURsqgY7Lbhqv0XZrEV50SWM0iaLt9dvZXbjM
eOltRRPNQBIAHMmJjnEc6dsCsmr/F7C2uJFqt1vxSDGVsrSFKJ/
CIMZuUzQ6eifio+P+94+3Wz82u32HL/
AB1WarM7fZ8FyvrbeeI+rNEv6k0zMs47rvs0wQvPukP01BBU18g9jinFF8pfvKlpetiFIa
TbhKlwdQv8RTHIRM6V0g7wV7OLG1S80y27a3oKDcuXqlpRmBALJnKFg7TJEajeb7vwwuL7
S1XpHmDobNr29t8K03yaCj1axPE5FCxSX2FQcguLLFLTtW3YNhYSo0mVEYMmok36xEv15L
5LO2cSaxJhh1DfhOhCStJmUlU/
CZ6EKgHUCJ3Fc88RYDcYFfvsL+O28Z1tl2RDqWin4xG4yrSCoaFQUAZSYJ/
82VH6XhRS8KKXhRS8KKgP8nuC6czrgnyRDe13GxnXeGa4utp2GRzZ510GjtdXMHm2OT7B9
qK847W/
r1RXsyWGWjfksyzjsIjroKia8tverd1iSlS0mCDBBgwZ+vbmN6dMGxI4TiVrfeGl1tpQC0
qBUFNkgLEDcxOXcZokEaEKfjDbaRzTAtqS8e4340dLQZ1h+w9c4PYZhVLHIanDiq7eS3KO
3ZnuHKb+m/
LshjG20CGj6PSWhAoVatX6jcvv3akOMoAkABUJSU+ERIJUSdFAEqnckg11TaLwl3DfGsOH
```

W14d42cZloDIUYPjZzOVWk+GQFzJIhJosL4guUPFjdOB7b1tpwXcX3Xq/
NIttyC19bxJcCyjTs1gJKwjMsdekkrWR4BNxmDFiQp8X0VJFNIamxoskhF2W4RbttYew62
AS5/EIJMODdJJ2KRoBsYnrXOnHmJ31/
xHeNXgLbVopQt0FISAy4fFChGqgsrKsx1ggaAAVcd441DaXhRS8KKXhRUCfkO+Rbj78bGk
F3HvWxsZsy8sJGN6z1xjDceTmuzcybgO2CUdEzKdBiuro8Rv8xZ2kxxuFXRlQnCdkvRIsl
RbWy7lzIghIGpJ2A/yegH0k1gtYQJiT+/lXOt+TX56eb/
yEUWS6lnXmNaL44ZUDsS10rrWO5IXLadl781Fg7L2LctFZZk2jjcZXIsNunrHXWgI651RB
UdV2DbOVKFZlKBEq56awNhvHM9zSfxFKEq0jkPy/
c+leW+KLY1hltjkGq8mze5x5a+qeujjyJseNh11j7TphbO3rgQvqtusx248l+OElpuYEOW
rsY0aVxyveIcKucGt/f7Z8i3dcS2tJAJaC5gpXvlMKASYKDokmRV+
+zHG7PiG6dwjFGAby0ZNw2oKhu4DZSF+K0YAWkFKioHI4kEuJkGWKz3nxsPWHMvN96cRtk
bH1u8KpgdHs3C7EaC4usMxlipiVMi1x1lHI95i1jdV9rYHXT2ZTP5adE+rGI2kFqbcLWCs
Ow5FveW3jN3fxrBglCjqk5dzA0URCkmYBqsOPsdY4g4iu721X9izDLaxoFoblOYEclGSnq
kjyolH4t/
xOe7ci2zrjQ3O+nw7NcZ2FldFhEPkDilVGwXKMNuMosYdDj9hneKVTa1GT40d3Jgsy5Nex
Uy4TU9yaTU1tkmvHa7wi3W047ay26gFWT7yVASSATqDvGpBjYTUPbeWkgOHMkbnmPl58x6
0coKoSdp/NUX+iiqoSf8ACov+PIxS2s+FFLworncfi/t8W97zQ4+6OYmyTxjUPHs81dr/
AKhpHay3cGcWrdnL9AX7PpjWucXBCVFUQcJP9Jr242qi00pYH3iZPYR9P81pWZUROkUJ3A
tGbiO056oKk2P1GjRF6VUTowVft0qIir/
Jft+6eOzN03cNpJ0J3B+o6z86TKBTsJp3NfZllWF1ud1+JPyor+VYlMrbKfEAClVtLFR9q
4kI717sMPUNxcQnjFUUW7T277RCTa9aW99brtbhvxmSpDkf1MLDiCeZhQg9RvNbrPELvDL
lV1ZPG3uFNPM5gdclw2WnE/3IUY5gwd68TVV6IAC02qoiECIIqqooD0a9In3NBQf7J/
RPFrQnlMb+dI1aADbTl05VtWZciBasR4pm1IhxltCkRzJJMWUyKyK1WFTr1lI8204C99p7
CqJ313vbX9sGgNBqrt0Hrp6Gg/
dM8/8ANdmjjhPy614+aLs9gT3LTPLHTmsJ+a2boNA5Y5bMwahk5HOMGHTADdt3ZjhIBkHs
a+pKPXkFuAkPvhAhAWoDykxTk3OREmTA+lPR5prOsEqoJKKdqiKqJ/
Nevsn+fCiuXd+IkxHkXtH5YuVV1aaozuTjWMUessYwkqqgl38Mda47rqnfp8lZKjbkqxXz
b2wy+X296OCswxNAQBBFTN9YNTavXjTbyk5si1hKucEZoB000J3ilSMFxi6t/
frXDLi5tcxT4jTSnEgjdJyAkHnqPKh4KOPdYpJnUN5DWqmvQ4zkqJZRGDmw25QQMhq1ZI2
jOueer5UNwvRQcVmSrTvQk4HmdivIpSJASsHvzkQeU9uXaaaXUkSFAhSSQRsQdiCOo6HY7
06EEY8xhp1uZCD8rSS35Qy1SF9SSw+40lfFJUcWZYnERhxpEQEcR0kVRUFVZGyQpCSCISD
Oka9B1MCe89aTER6x5+enKthVPy4kqNIglLhymjIWpMM347jr7jToC00/
HNCF8mnDbURXsmxVOlT373tkpVKfhXPLSZHWfTyrw6gSJSZ7/
v8AetSu4bcN9mcs9kZ7CxLIsLxNnE4ldazCzFy8KRJrHoAiydLX1dQ/
+pNdQ5QgZOMsqcdRQ+xXqP4hxLbYJfOouWHH1rgpCMuwSnQqUQPyI7VMOGuC8Q4paedtLp
m2at1BKy6V5gTJkJQkyPMiur3werbal4gcbKO7mUM+xotM4DRHKxgbFqgOPS0ESqgjWRrU
lfiRxgw44/
QMiRggJhsiabAlaTcN3Z96aQUN3HxgEgkBWsEiASJ3503Xti5ht5dYe8oLcsnFNFQBSCUE
pkAyQDEjsalR55SasL30vX79fb+/8P28KKB/+QvNLXMOWvKnZEJoq9+023J4/
SwaIikRMdwStq9TN2rQuKinKWZBmzvoIo+6WAgCe6L3WuKpdxHH3QDkQw4hmOeXQFQ9SSe
ddOcJIZwLgWwuAM7lww9dqVIAlWdYT5hICR3566BtcvcWo8f5ecmsZxua9+lYXyD29gtQq
st9u0+vs5usFqX1JohEXHKvHohn0nSuESr91+9s2zDZDZQspW2IIjpoDM8xHauXLlxTjjr
qxq8pSz5qJUfzNN1UMQxiHHdklFBmPKcj9R3JZypAMq4EZz0dFWUed6En19xa79lAk6TyQ
MBPhpTMAbc56/qdqQEzqRv+n6963sK0mxIjrcWXMZZsBb/
NxWJTzLEpI8gn4Zym2y9XvpPEhNqqdgqqoknZdrUKOQK1k8vXTzifrWMa+VEjcIsOpMa0B
wv39gF3XVuV51p3lXqPaVO8Liu2mR6D5KFe4fZxSb/
e2cwTfNbEJk09SYrYZ+wiqqlTe0FCkPsPpORWmv8ATkE+mlXf7HnQp7FrZRBQUoVHUhR8u
X1FGyfGDsWRlmg52HzTcJ/
WuUyKiCjv+sKDI4MfKqxov49NTLC4ZD+CNxwFPsPmHDVyq4w1CV/eYUU+Y3B/

```
MgdhTf7UMNTY8Se8tpCUYmyh0gfziW1/PKknuTVknkgquK+TXpE/3An+TFF/
6XwooCKkiyty8pcu1veGNta7R+SNaqXGmk84LsKx5UlLuHXRRfZIrdBVzVNE6UW2C/
gnlc2ClPcQ3BJlZuSTPRKjJ9Amumbzw7b2bswmGmsPSAe6mkxv1UrT1oTHltmFVlfM/
lXl9MoHjubcn+RWQUjgqiNuV13ufNrCA52gogqseQ39kRE6cTpOuvLXtiEOtk/
ceEa9dte81y44CQRMEf6/
f502MU44uCk78ykcI89GvyRxweSWcF8YKOFKFQOKkz6KvIiI4rKGjSo4oKj8ElOUEQEzt1
I08x15xtScg9enXrrW7iyP/X6r/
wDMQgLtEVET1T9v5L7evX9vFyCAkJ30M1rIgn971d98c+fBfcQdnwRaKTdcVeU+B7pYbV0
jdiaT5OYg1oPYkqIyi/avibj1xo1+WfXoJ5HHMlFV7KvfaBbrewtu7QP4C42/
CNJ8tT9asz2XXqbTiP3dasqb5pSBrHxJhQ8zp86OM+IW1S0qN7kKAjazdVTGfTtEcCVjN6
2Toov7D9SOQp/sVO/I5wwR7vcBMAAoOmm6alHtbSoXuCLUZzsvAf2u/
rtpVy3koqoawqdp/bpft/Re0/7TwooO/
iJjWLD8tfJiXOrnVoOM27+c25pTkavctpEMaN+zYoZsWua/8p02NM2E67GYFF+o/
WNiKKS9eQrCbdB4lxNSRq044qf+4SI9Co/Wr54iv3G/ZPgaVEhV4m3ZJ6hEr/
8ALQ57GufhaG1kci2nOPOSSs7Swu2JxAjMl0rOZInJMJtO0akmr4k612vqTiiir6ovlpoZ
S60puJj4gRvtuO/Uc659WTM8/r+
+Rp9eM+qLrkTtXB9N007Hq3JswDM6+vm5RKtodElpjmA5Xl0dZMqkiPyGn3xxxWoyI0TJy
32GpShGN4wUpuUtWodeSVFo5VAbyfhCh6kH0oaYU+
+llCgFObHlsTB36RTXFINyJEFjsXbAGDVD9FMGjbF91XBT7CSKXSoiqiKnSd+Of4YSZKvp
zPry+VJgOtXa/Ali1hsbnpB4/
pjd9kuA8j9AcgdD7VYpGY0lvH8ByfBXLxjYNwE2S0yMCh2FjWCTm3lL6zM9yGUMTmkwBNe
NWzL+G3DDhHhrAHzMad9TS7D7p2xvrW8YMPWy0rT5pM/Lke1Gn/CEWW0De/tf7DYKvz/
DSxHE8sgOqIkORa+yTYeD5GbQovRsJbVp+pD2Ki+CovRD3VnCyFsf8jaO/
wAS2cCD5JzAHyiKub2pvNX1lwjijHxM3rDygY/m8FUdjqZFX8eS2qdrBft/yiJ/
HpVVERf89eFFAvRdtZ9qzkr8qXKPj5f1b9RDpuV2d2ke9n1IS7PXljeSnbt3HrZJJHCnlZ
Y/UHRyWkeVJkgo7oONtu+kSw5Dpx3EfcCiXSSpTgkEkg7AgkA6EyQQSBEGug8TYsG/
Zvgh4lbeVb2QZ8Nu3IQ6XFIKGgVKSUoSpJMkjSCrUwKFv3NxAsGsdi7N15jsjFYUtGktaV
q9lZbDaluxweWW1ayK1h047jv1UJs2nfRwSVHuiVBtS1wu7RZMOZ/
GuACVuNgJGpJTCBIyhIymSTImdYHOmI3tmu+uFWlqqxslkBtpbqnlJASkHM6pKCoqVKpyp
icoBiaiBXVGe4zaV0+tOzxvJWprpVl1S2kqneF0YMgJLlfeV8gDguuVrskCRDA1F4mlRRN
UXMtLU2rOiVFQnT4VEfl3jka0B5IUkoXlIGmpBHKpD6R4obU3K9HlxIcXF8bZcBqVkWQOi
2jbZGIODXVn1AdsHkc9RRO2m1UkT3VO1R+s8MubhKFqSLZrkpwwTp+FO55a6DnSNy8aRoi
Xl6CE6x5nl5a+lGYfBFp7WHC3fNd+gyDsbXeOJPa9yPNcpOKtnPlvmzkWMU8FuLGQcZrEy
inYZJiN2Eorps5rkp2LFVrXjWGtDBnF5ZuGCleaTMAwofyhJSqY1MoBnU17bvuKu0Qr7Iy
mI0OYSD1JCgBuBCiOVWHfG1ubJ4vK/Jj2LUW9bK5I12Q29Vbzql2DTZXfyLm9zpb/
ABGYkdBuaErCvzmA6/G7jA/XABuEYKo0Zg/jtYpdOvkgYlmUNIBKVGCnnEBQPkDzrpbj/
DLN3g7D28NUgnhUstvJCgVIDzTYUhev3wVNLIPxQpWkURZ+/kvqg6//2Q==='

# Select the unique subcategories from the DataFrame
unique_subcategories = styles_df['articleType'].unique()

# Analyze the image and return the results
analysis = analyze_image(encoded_image, unique_subcategories)
image_analysis = json.loads(analysis)

# Display the image and the analysis results
display(Image(filename=reference_image))
print(image_analysis)
```

```
{'items': ["Men's White Casual T-shirt", "Men's Dark Wash Jeans",
"Men's Casual Sneakers"], 'category': 'Shirts', 'gender': 'Men'}
```

Next, we process the output from the image analysis and use it to filter and display matching items from our dataset. Here's a breakdown of the code:

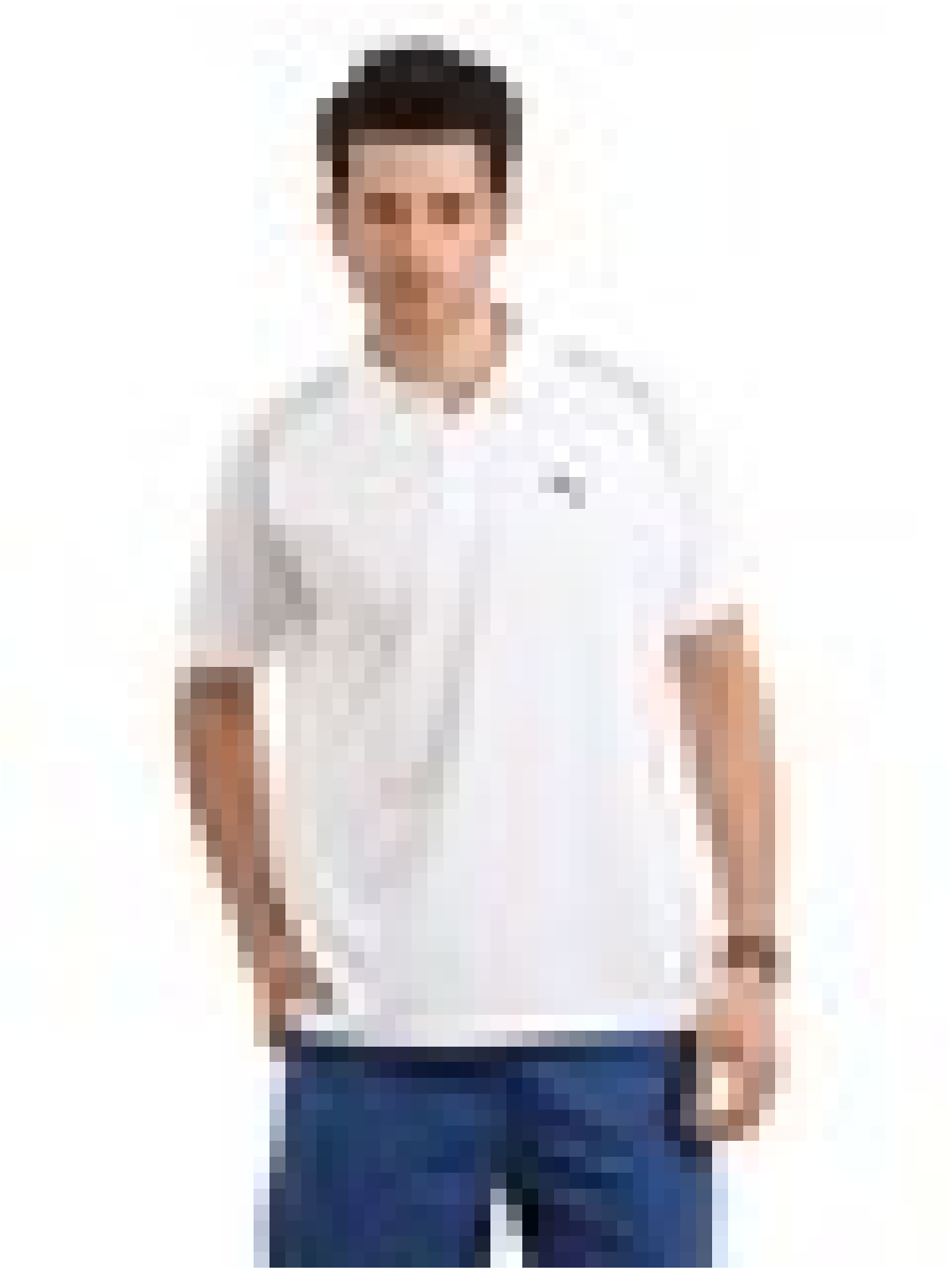
1. **Extracting Image Analysis Results**: We extract the item descriptions, category, and gender from the `image_analysis` dictionary.
2. **Filtering the Dataset**: We filter the `styles_df` DataFrame to include only items that match the gender from the image analysis (or are unisex) and exclude items of the same category as the analyzed image.
3. **Finding Matching Items**: We use the `find_matching_items_with_rag` function to find items in the filtered dataset that match the descriptions extracted from the analyzed image.
4. **Displaying Matching Items**: We create an HTML string to display images of the matching items. We construct the image paths using the item IDs and append each image to the HTML string. Finally, we use `display(HTML(html))` to render the images in the notebook.

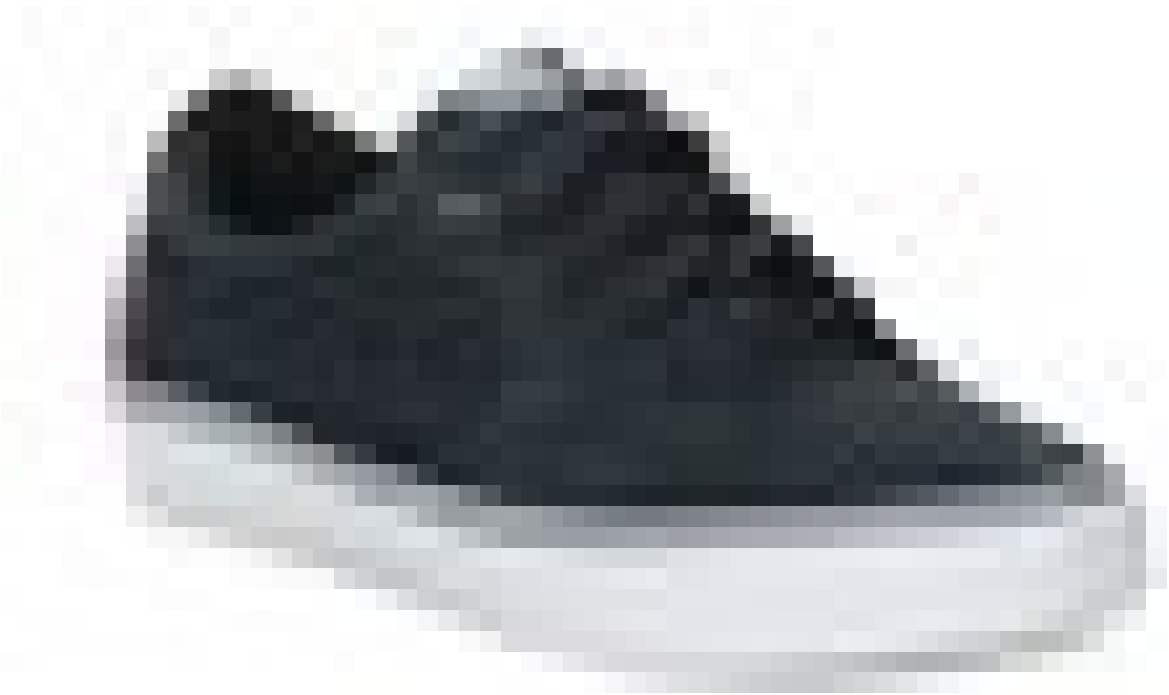
This cell effectively demonstrates how to use the results of image analysis to filter a dataset and visually display items that match the analyzed image's characteristics.

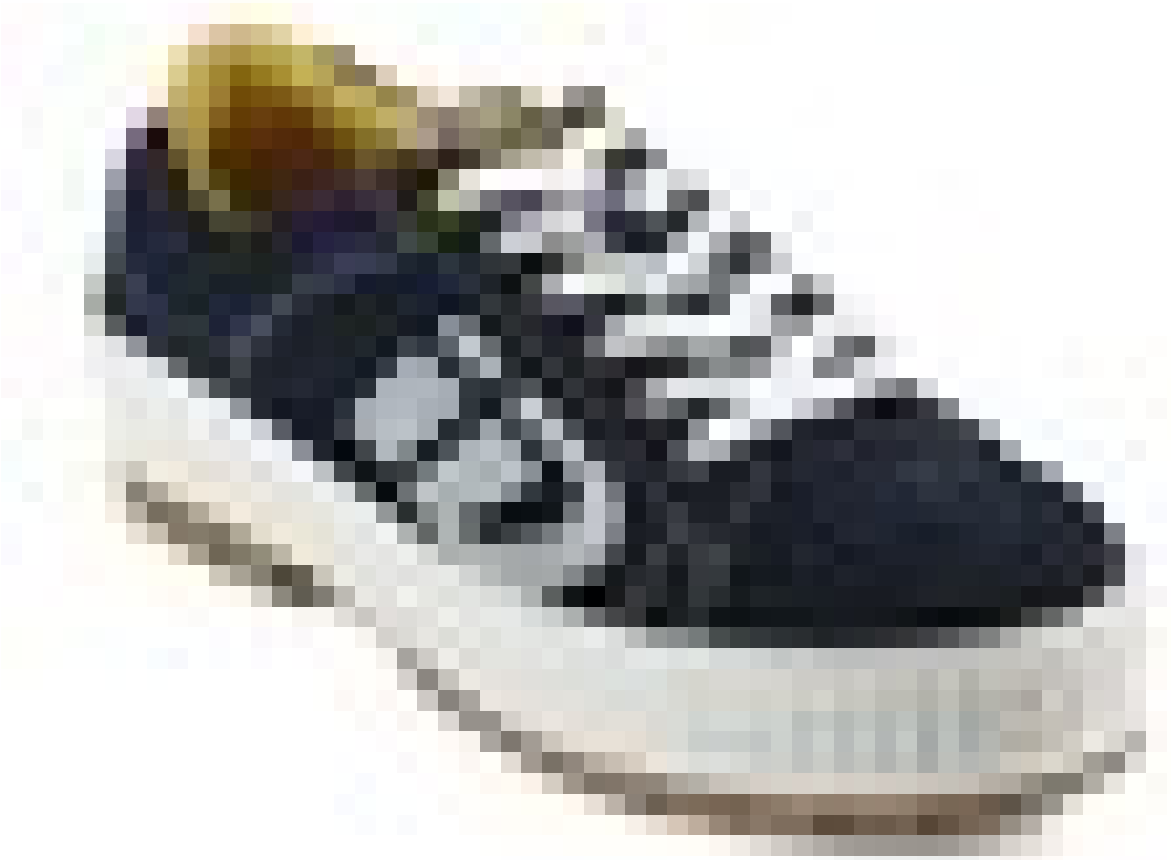
```
# Extract the relevant features from the analysis
item_descs = image_analysis['items']
item_category = image_analysis['category']
item_gender = image_analysis['gender']


# Filter data such that we only look through the items of the same
gender (or unisex) and different category
filtered_items = styles_df.loc[styles_df['gender'].isin([item_gender,
'Unisex'])]
filtered_items = filtered_items[filtered_items['articleType'] !=
item_category]
print(str(len(filtered_items)) + " Remaining Items")

# Find the most similar items based on the input item descriptions
matching_items = find_matching_items_with_rag(filtered_items,
item_descs)
```

```
# Display the matching items (this will display 2 items for each
description in the image analysis)
html = ""
paths = []
for i, item in enumerate(matching_items):
    item_id = item['id']

    # Path to the image file
    image_path = os.path.join(curr_path, "sample_clothes",
"sample_clothes", "sample_images", f'{item_id}.jpg')
    paths.append(image_path)
    html += f'<img src="{image_path}"
style="display:inline;margin:1px"/>'

# Print the matching item description as a reminder of what we are
looking for
print(item_descs)
# Display the image
for pth in paths:
  display(Image(filename=pth))
```

```
513 Remaining Items
["Men's White Casual T-shirt", "Men's Dark Wash Jeans", "Men's Casual
Sneakers"]
```







Guardrails

In the context of using Large Language Models (LLMs) like GPT4o-Mini, "guardrails" refer to mechanisms or checks put in place to ensure that the model's output remains within desired parameters or boundaries. These guardrails are crucial for maintaining the quality and relevance of the model's responses, especially when dealing with complex or nuanced tasks.

Guardrails are useful for several reasons:

1. **Accuracy**: They help ensure that the model's output is accurate and relevant to the input provided.
2. **Consistency**: They maintain consistency in the model's responses, especially when dealing with similar or related inputs.
3. **Safety**: They prevent the model from generating harmful, offensive, or inappropriate content.
4. **Contextual Relevance**: They ensure that the model's output is contextually relevant to the specific task or domain it is being used for.

In our case, we are using GPT4o-Mini to analyze fashion images and suggest items that would complement an original outfit. To implement guardrails, we can **refine results**: After obtaining initial suggestions from GPT4o-Mini, we can send the original image and the suggested items back to the model. We can then ask GPT4o-Mini to evaluate whether each suggested item would indeed be a good fit for the original outfit.

This gives the model the ability to self-correct and adjust its own output based on feedback or additional information. By implementing these guardrails and enabling self-correction, we can enhance the reliability and usefulness of the model's output in the context of fashion analysis and recommendation.

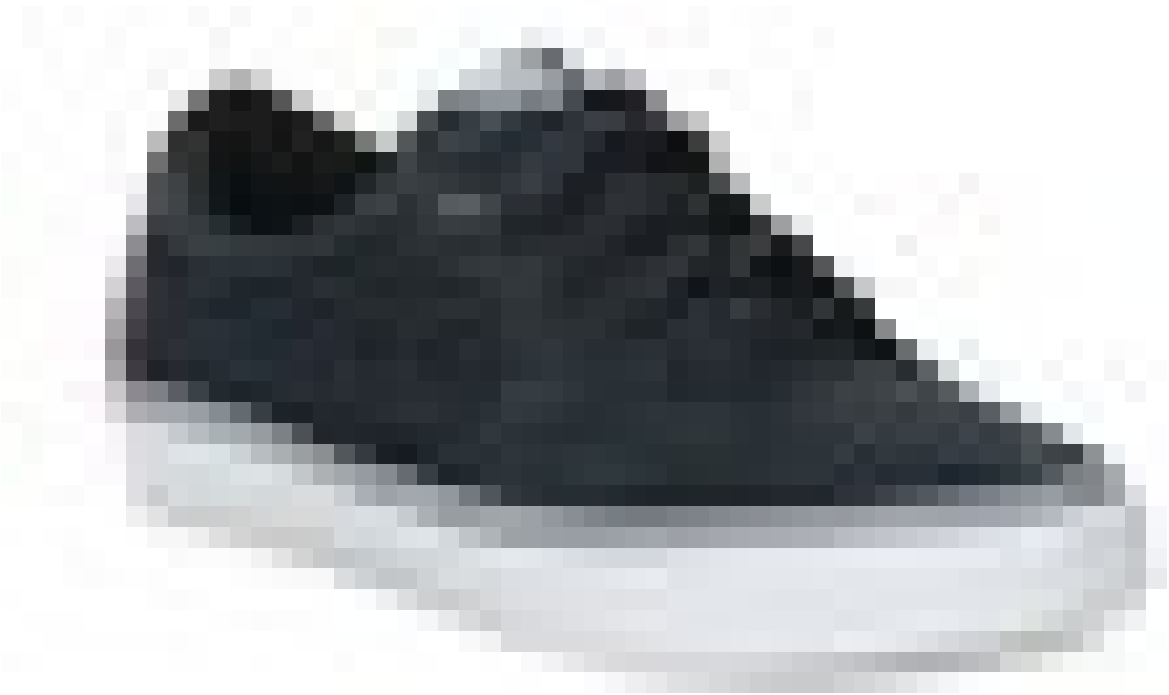
To facilitate this, we write a prompt that asks the LLM for a simple "yes" or "no" answer to the question of whether the suggested items match the original outfit or not. This binary response helps streamline the refinement process and ensures clear and actionable feedback from the model.

```
from langchain_core.messages import HumanMessage

def check_match(reference_image_base64, suggested_image_base64):
    # Prepare the prompt and image inputs for AzureChatOpenAI
    """
    Determines if two clothing items, represented by their base64-
encoded images, would work well together in an outfit using an LLM-
based vision model.

    Guidelines for implementing this function:
    1. **Inputs**:
        - `reference_image_base64` (str): Base64-encoded string of the
reference clothing item image (the item to be matched).
        - `suggested_image_base64` (str): Base64-encoded string of the
suggested clothing item image (the candidate match).
```

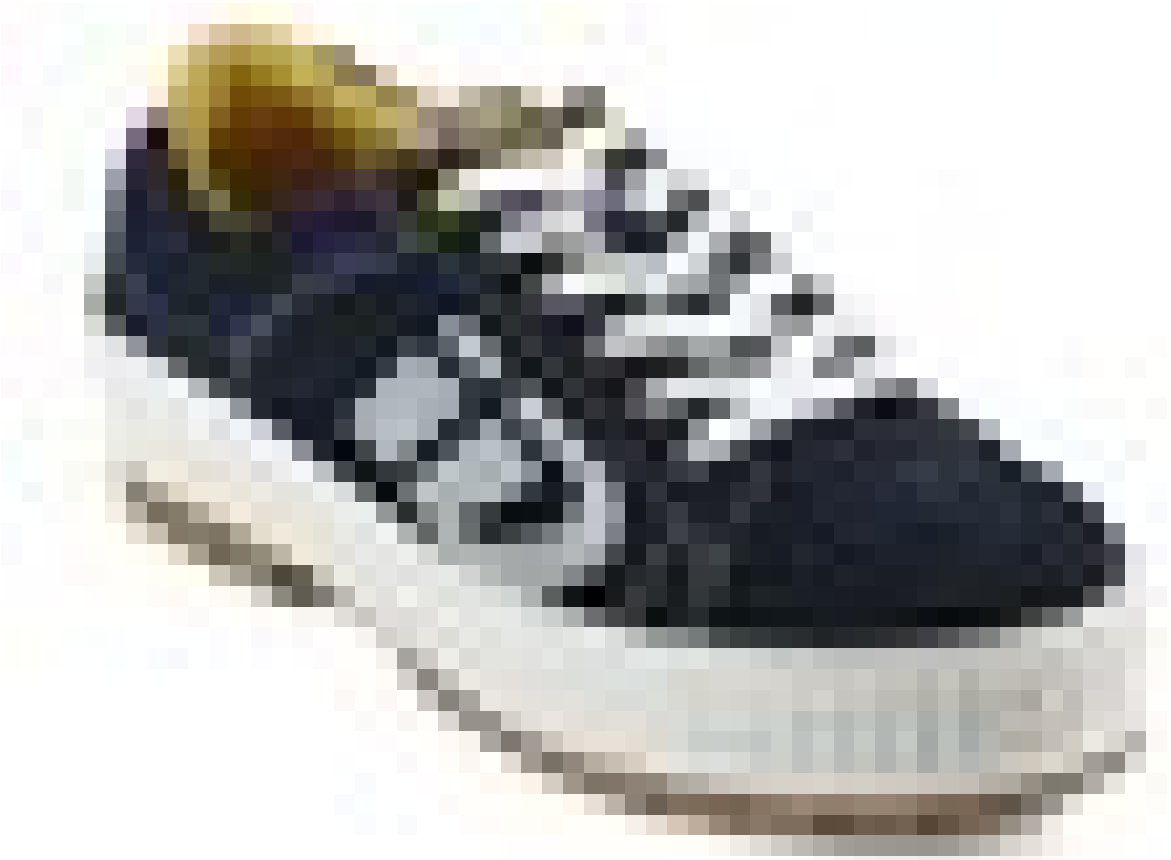
```
    2. **Prompt Construction**:
        - Construct a prompt that clearly instructs the model to
compare the two clothing items and decide if they would work well
together in an outfit.
        - The prompt should specify:
            - The first image is the reference item.
            - The second image is the suggested item.
            - The model must output a JSON object with two fields:
"answer" (either "yes" or "no") and "reason" (a brief explanation).
            - The explanation should not include descriptions of the
images themselves.
            - The output must not include the ```json``` tag.

    3. **Message Formatting**:
        - Use the `HumanMessage` class to format the prompt and attach
both images as base64-encoded image URLs.
        - The message content should be a list containing:
            - The prompt text (as described above).
            - The reference image (as a dict with type "image_url" and
the base64 string).
            - The suggested image (as a dict with type "image_url" and
the base64 string).

    4. **Model Invocation**:
        - Use the AzureChatOpenAI client (assumed to be initialized as
`client`) to send the message.
        - Set an appropriate `max_tokens` limit (e.g., 300) to ensure
concise output.

    5. **Output Handling**:
        - Extract the model's response content.
        - Return the response as a string (expected to be a JSON
object with "answer" and "reason").

    6. **Error Handling**:
        - Optionally, add error handling to manage cases where the
model's response is not valid JSON or does not contain the expected
fields.

    Example output:
        {
            "answer": "yes",
            "reason": "The suggested item complements the reference
item in both color and style, making them suitable for an outfit
together."
        }
    """
    # implement this function

    raise NotImplementedError("This function is not yet implemented")
```

Finally, let's determine which of the items identified above truly complement the outfit.

```
# Select the unique paths for the generated images
paths = list(set(paths))

for path in paths:
    # Encode the test image to base64
    suggested_image = encode_image_to_base64(path)

    # Check if the items match
    match = json.loads(check_match(encoded_image, suggested_image))

    # Display the image and the analysis results
    if match["answer"] == 'yes':
        display(Image(filename=path))
        print("The items match!")
        print(match["reason"])
```



```
The items match!
The black shirt and black shoes share a similar color, creating a cohesive look. Additionally, the casual style of both items makes them compatible for a relaxed outfit.
```



```
The items match!
The casual style of the black button-down shirt pairs well with the sporty look of the navy sneakers. Both items share a relaxed, everyday aesthetic that would complement each other in an informal outfit.
```

```
The items match!
The dark, solid color of the shirt complements the light wash and casual style of the jeans, making them a suitable match for a relaxed yet cohesive outfit.
```

```
The items match!
The casual style and dark color of the shirt complement the relaxed fit and design of the jeans, making them suitable to be paired together in an outfit.
```

We can observe that the initial list of potential items has been further refined, resulting in a more curated selection that aligns well with the outfit. Additionally, the model provides explanations for why each item is considered a good match, offering valuable insights into the decision-making process.

Conclusion

In this Jupyter Notebook, we explored the application of GPT-4 with Vision and other machine learning techniques to the domain of fashion. We demonstrated how to analyze images of clothing items, extract relevant features, and use this information to find matching items that complement an original outfit. Through the implementation of guardrails and self-correction mechanisms, we refined the model's suggestions to ensure they are accurate and contextually relevant.

This approach has several practical uses in the real world, including:

1. **Personalized Shopping Assistants**: Retailers can use this technology to offer personalized outfit recommendations to customers, enhancing the shopping experience and increasing customer satisfaction.

2. **Virtual Wardrobe Applications**: Users can upload images of their own clothing items to create a virtual wardrobe and receive suggestions for new items that match their existing pieces.
3. **Fashion Design and Styling**: Fashion designers and stylists can use this tool to experiment with different combinations and styles, streamlining the creative process.

However, one of the considerations to keep in mind is **cost**. The use of LLMs and image analysis models can incur costs, especially if used extensively. It's important to consider the cost-effectiveness of implementing these technologies. `GPT4o-Mini` is priced at `$0.01` per 1000 tokens. This adds up to `$0.00255` for one 256px x 256px image.

Overall, this notebook serves as a foundation for further exploration and development in the intersection of fashion and AI, opening doors to more personalized and intelligent fashion recommendation systems.