Style Finder: Computer Vision-Based Fashion Analysis

Estimated time needed: 45 minutes

Overview

This lab walks you through building a complete multimodal Retrieval-Augmented Generation (RAG) pipeline for fashion analysis. You'll learn how to combine computer vision, vector similarity search, and large language models using the Llama 3.2 Vision Instruct model to perform detailed, catalog-style fashion analysis from images.

By the end, you'll deploy your pipeline as an interactive web app using Gradio, allowing users to upload images and receive AI-generated fashion insights in real time.

Learning objectives

After completing this lab, you will be able to:

- Implement a complete multimodal RAG pipeline integrating computer vision with structured fashion datasets
- Use image encoding techniques for vector-based similarity matching and retrieval
- Explain how to enhance LLM responses by augmenting them with retrieved contextual information
- Develop a user-friendly interface with Gradio
- Apply modular programming principles to create maintainable AI applications
- Gain practical experience with state-of-the-art multimodal AI techniques

A quick look at Style Finder

What does the Style Finder do?

The Style Finder is an advanced computer vision application that leverages deep learning to analyze fashion elements in images. Using Meta's Llama 3.2 90B Vision Instruct model, this system identifies clothing items, retrieves relevant metadata, and provides actionable insights based on visual inputs.

Upload any fashion photo, and Style Finder will identify garments, analyze their style elements, and provide detailed information about each item. It can also find similar items at different price points, making high-end fashion more accessible.

With the dataset based on Taylor Swift's iconic outfits, the system demonstrates how to connect visual inputs with structured fashion data for precise identification of garments, accessories, and styling elements with corresponding commercial availability information.

Tips for the best experience! - Read Only

Keep the following tips handy and refer to them at any point of confusion throughout the lab. Do not worry if they seem irrelevant now. You will go through everything step by step later.

Please note the content on this page is for read-only. Do NOT try to run any code here.

- At any point throughout the lab, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.

- Whenever you make changes to a file, be sure to save your work. Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.

- At the end of each section, you will be given the fully updated script for that part. And at the end of the lab, you will be prompted to pull the complete codebase of the lab as well.

- For running the application, always ensure `app.py` is running in the background before opening the Web Application.

- You run a code block by clicking `>_` on bottom right.

  ```
  python app.py
  ```

- Always ensure that your current directory is `/home/project/style-finder`. If you are not in the `style-finder` folder, certain code files may fail to run. Use the `cd` command to navigate to the correct location.

- Make sure you are accessing the application through port `5000`. Clicking the purple Web Application button will run the app through port 5000 automatically.

  Web Application

- If you get an error about not being able to access another port (for example, 8888), just `refresh` the app by clicking the small refresh icon. In case of other errors related to the server, simply refresh the page as well. (The following image is for illustration purpose only.)

- To stop execution of `app.py` in addition to closing the application tab, press `Ctrl+C` in terminal.

- If you encounter an error running the application or after you entered your desired keyword, try refreshing the app using the button on the top of the application's page. You can try inputting a different query too.

- Typically, using the models provided by watsonx.ai would require watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed.

Setting up your development environment

Before you dive into development, set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Gradio application.

Step 1: Create your project directory

Open the terminal in Cloud IDE and run:

```
git clone --no-checkout https://github.com/HaileyTQuach/style-finder.git
cd style-finder
git checkout 1-start
```

Once you are all set up, select File Explorer, and then the `style-finder` folder. You should have all the files structured as below.

Step 2: Set up the virtual environment

Next, you will set up a virtual environment for the project and install all the required libraries.

Note, you can run the snippet directly by clicking the run button `>_`. Approximately, it will take **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

This set of commands installs essential packages for this project:

- `ibm-watsonx-ai`: To interact with Llama Vision Instruct model for generating AI-powered responses.
- `image`: For handling image processing tasks.
- `requests`: For handling HTTP requests, such as downloading images from URLs.
- `pillow`: To handle image manipulation and preprocessing tasks.
- `transformers`: For integrating advanced models like Llama Vision Instruct into the workflow.
- `torch`: For deep learning operations and managing model computations.
- `ipywidgets`: To add interactive elements like image uploaders in Jupyter Notebook.
- `scikit-learn`: For calculating cosine similarity between vectors and other machine learning tasks.

```
python3.11 -m venv venv
source venv/bin/activate # activate venv
pip install -r requirements.txt
```

Step 3: Download the dataset

To demonstrate this lab, a dataset is curated from taylorswiftstyle.com. Each entry represents a fashion item and includes its name, price, purchase link, and an image showing the full outfit as worn by a model. For ease of use, the dataset was preprocessed to retain only the features necessary for model inference and comparison, ensuring smooth integration with our system.

How the dataset was built

Below is an overview of the steps involved in building the dataset:

1. **Web scraping**:

   - Scrape the first 10 pages of the website with BeautifulSoup, extracting key information such as:
     - Images of Taylor Swift's outfits
     - The name of each item
     - The price of each item
   - The data was systematically collected and organized into a structured format.

2. **Data storage**:

   - The scraped data, including item names, prices, and image URLs, was stored in a pandas DataFrame for easy processing and analysis.

3. **Image encoding**:

   - Each image was downloaded and processed further:
     - The images were encoded into Base64 strings, a data format that the model can digest.
     - The images were also converted into vectors to enable processing by the model, making it possible to analyze and compare visual features.

To access the dataset, run the following command to download the `.pkl` file using `wget`, and then save it as "swift-style-embeddings.pkl" in your current directory.

```
wget -O swift-style-embeddings.pkl https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/95eJ0YJVtqTZhEd7RaUlew/processed-swift-style-with-embed
```

Now that your environment is set up, you're ready to start building your Style Finder application!

Understanding multimodal RAG

1. What is multimodal RAG?

In this lab, you're building a multimodal Retrieval-Augmented Generation (RAG) system. Let's break down what this means:

- **Multimodal:** Our system works with multiple types of data - both images (fashion photos) and text (descriptions, prices, etc.).
- **Retrieval-augmented:** We enhance AI responses by first retrieving relevant information from a database.
- **Generation:** We use this retrieved information to generate detailed, accurate responses.

While Generative AI models, like Llama 3.2 90B Vision Instruct, are built using extensive datasets, they don't naturally include your specific data. [Retrieval-Augmented Generation (RAG)](#) solves this issue by merging your data with the existing knowledge of the models. Throughout this lab, you'll frequently see mentions of RAG, as it's a critical technique used in query and chat engines, as well as agents, to boost their performance.

[Source](#)

2. The multimodal RAG pipeline

Our Style Finder implements a complete multimodal RAG pipeline with these components:

1. Multimodal input processing:

- User-uploaded images are processed and converted to vector representations.
- These vectors capture the essential visual features of the fashion items.

2. Vector-based retrieval:

- The system compares the input image vector against a database of pre-encoded fashion images.
- Cosine similarity is used to find the closest matching items in the vector space.
- Relevant structured data (item details, prices, links) is retrieved from the matched items.

3. Context-enhanced generation:

- The retrieved fashion data is formatted into prompts for the Llama 3.2 Vision Instruct model.
- The LLM generates comprehensive responses based on both the visual input and the retrieved context.
- This augmentation improves response accuracy and detail beyond what would be possible with the image alone.

Part 1: Setting up your configuration

In this step, you'll set up the configuration file for your Style Finder. The `config.py` file centralizes all the settings for your Style Finder application, making it easier to modify parameters without changing the core code.

Step 1: Open the config.py file

First, click the purple button below to open the `config.py` file. It already contains some boilerplate code and placeholders:

```
Open config.py in IDE
```

This configuration file defines:

- **Model and API settings:** Specifies which Llama model to use and connection details
- **Image processing parameters:** Sets the standard size and normalization values for image preprocessing
- **Matching thresholds:** Determines when an image match is considered "good enough"
- **Search settings:** Controls how many alternative products to retrieve

It is recommended to keep these default settings until you are more familiar with the process to experiment with different settings.

Next, you'll implement the image processing module that handles the visual part of your multimodal system.

Part 2: Implementing image processing module

Now create the image processing module. This is a crucial component of your multimodal RAG system as it handles:

- Converting images to a format that your LLM can understand
- Creating vector embeddings for similarity matching
- Finding the closest matches in your fashion database

Why vector embeddings matter

Vector embeddings are crucial for multimodal RAG systems because they:

- Convert visual information into a format that can be mathematically compared
- Allow you to find "similar" images, not just exact matches
- Enable fast retrieval through vector similarity operations
- Form the bridge between visual and textual information

This concept can be visualized using two-dimensional vectors and the table, where closer vectors represent more similar items. For example, in an image retrieval system for outfits, each outfit image can be encoded as a vector based on its features (for example, color, style, or texture). Similar outfits will have feature vectors that are closer together.

For instance:

- The vector `[1, 2]` would be very close to `[1, 2]` (identical) and `[2, 1]` (similar), indicating that these outfits share many visual features.
- Conversely, `[1, 2]` is farther from `[-1, -2]` (opposite features) and `[0, 0]` (no meaningful features), suggesting these outfits are less similar or completely dissimilar.

In the context of image retrieval, when you query with an image of a specific outfit, the system computes the dot product or similarity scores between the query vector and the dataset vectors. The system then retrieves outfits with feature vectors closest to the query vector, effectively surfacing similar outfits.

Vector 1	Vector 2	Dot Product	Explanation
[1, 2]	[1, 2]	5	Vectors are identical, so the dot product is high, indicating high similarity.
[1, 2]	[2, 1]	4	Vectors are somewhat similar, leading to a moderate dot product value.
[1, 2]	[-1, -2]	-5	Vectors are opposites, resulting in a negative dot product, showing dissimilarity.
[1, 2]	[0, 0]	0	One vector is zero, indicating no similarity or contribution to the dot product.

Step 1: Examining the image_processor.py Starter File

First look at the starter file structure. Click the purple button below to open `image_processor.py`.

Open **image_processor.py** in IDE

This module uses a pre-trained neural network (ResNet50) to turn images into vectors (embeddings). These vectors capture the essential features of images, making it possible to compare them mathematically. In this file, you need to implement the following:

1. In the `__init__` method:

- Initialize self.device: Determine if CUDA is available and set the appropriate device for PyTorch (CPU or GPU)
- Initialize self.model: Load a pre-trained ResNet50 model, set it to evaluation mode, and move it to the specified device
- Initialize self.preprocess: Create an image transformation pipeline using torchvision's transforms that resizes images, converts them to tensors, and normalizes them

2. In the `encode_image` method:

- **Function purpose:**
  Convert an image (from a URL or local file) into two key components: a base64-encoded string for LLM input and a numerical feature vector for similarity comparisons

- **Implementation details:**

  - Accepts an image input and a Boolean flag indicating whether the image is from a URL or a local path
  - If the input is a URL, it downloads and loads the image; otherwise, it reads the image from the local filesystem. The image is converted to RGB to ensure consistency
  - Converts the image to a base64-encoded JPEG string using an in-memory buffer (BytesIO)—useful for transmitting the image to vision-language models
  - Preprocesses the image using a predefined transformation pipeline (self.preprocess) and prepares it as a tensor batch on the designated device (self.device)
  - Passes the tensor through a pretrained ResNet50 model to extract high-level image features
  - Flattens the resulting tensor to a one-dimensional NumPy array representing the image's feature vector (embedding)
  - Wraps and returns both the base64 string and the feature vector in a dictionary

- **Expected output:**
  A dictionary with two keys:

  - 'base64': A string representing the JPEG-encoded image in base64 format
  - 'vector': A flattened NumPy array containing the feature embedding of the image, useful for downstream tasks like similarity computation or clustering
    If an error occurs during processing, both values are returned as None

3. In the `find_closest_match` method:

- **Function purpose:**
  Identify the most visually similar item in the dataset to a user-uploaded image by comparing feature vectors using cosine similarity

- **Implementation details:**

- Takes the feature vector of the user's image (user_vector) and a DataFrame (dataset) that contains precomputed image embeddings under the Embedding column
- Stacks all non-null embeddings from the dataset into a single NumPy array for batch similarity computation
- Computes the cosine similarity between the user's vector and each dataset vector using cosine_similarity from sklearn
- Finds the index of the highest similarity score, indicating the closest match
- Retrieves the corresponding row from the dataset based on this index
- Returns both the matched row and the similarity score as a tuple

- **Expected output:**
  A tuple containing:

  - closest_row: The DataFrame row that has the highest cosine similarity to the user's image
  - similarity_score: A float value between -1 and 1 indicating how similar the matched image is to the user's image
    If an error occurs during processing, both outputs are returned as None

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the init function

Now, implement the function to initialize the image processor with a pre-trained ResNet50 model. Copy-paste the following code into `__init__` function and save your files.

```
def __init__(self, image_size=(224, 224),
             norm_mean=[0.485, 0.456, 0.406],
             norm_std=[0.229, 0.224, 0.225]):
    """
    Initialize the image processor with a pre-trained ResNet50 model.

    Args:
        image_size (tuple): Target size for input images
        norm_mean (list): Normalization mean values for RGB channels
        norm_std (list): Normalization standard deviation values for RGB channels
    """
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.model = resnet50(pretrained=True).to(self.device)
    self.model.eval()  # Set model to evaluation mode

    # Image preprocessing pipeline
    self.preprocess = transforms.Compose([
        transforms.Resize(image_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=norm_mean, std=norm_std),
    ])
```

Step 3: Implement the encode_image function

Now, implement the function to encode an image and extract its feature vector. Copy-paste the following code into `encode_image` function and save your files.

```
def encode_image(self, image_input, is_url=True):
    """
    Encode an image and extract its feature vector.

    Args:
        image_input: URL or local path to the image
        is_url: Whether the input is a URL (True) or a local file path (False)

    Returns:
        dict: Contains 'base64' string and 'vector' (feature embedding)
    """
    try:
        if is_url:
            # Fetch the image from URL
            response = requests.get(image_input)
            response.raise_for_status()
            image = Image.open(BytesIO(response.content)).convert("RGB")
        else:
            # Load the image from a local file
            image = Image.open(image_input).convert("RGB")
        # Convert image to Base64
        buffered = BytesIO()
        image.save(buffered, format="JPEG")
        base64_string = base64.b64encode(buffered.getvalue()).decode("utf-8")
        # Preprocess the image for ResNet50
        input_tensor = self.preprocess(image).unsqueeze(0).to(self.device)
        # Extract features using ResNet50
        with torch.no_grad():
            features = self.model(input_tensor)

        # Convert features to a NumPy array
        feature_vector = features.cpu().numpy().flatten()
        return {"base64": base64_string, "vector": feature_vector}
    except Exception as e:
```

```
        print(f"Error encoding image: {e}")
        return {"base64": None, "vector": None}
```

Step 4: Implement the find_closest_match function

Now, implement the function to find the closest match in the dataset based on cosine similarity. Copy-paste the following code into `find_closest_match` function and save your files.

```
def find_closest_match(self, user_vector, dataset):
    """
    Find the closest match in the dataset based on cosine similarity.

    Args:
        user_vector: Feature vector of the user-uploaded image
        dataset: DataFrame containing precomputed feature vectors

    Returns:
        tuple: (Closest matching row, similarity score)
    """
    try:
        dataset_vectors = np.vstack(dataset['Embedding'].dropna().values)
        similarities = cosine_similarity(user_vector.reshape(1, -1), dataset_vectors)

        # Find the index of the most similar vector
        closest_index = np.argmax(similarities)
        similarity_score = similarities[0][closest_index]

        # Retrieve the closest matching row
        closest_row = dataset.iloc[closest_index]
        return closest_row, similarity_score
    except Exception as e:
        print(f"Error finding closest match: {e}")
        return None, None
```

Click the below button to see the fully updated `image_processor.py`.

▶ Click to see the solution

Now that you have your image processing module ready, let's move on to implementing the LLM service that will generate responses based on these image matches.

Part 3: Implementing the interface with the Llama Vision Instruct model

In this step, you'll implement the service that interfaces with the Llama 3.2 Vision Instruct model. This component will handle communication with the AI model and generate fashion-specific responses.

Step 1: Examining the llm_service.py Starter File

First examine the starter file structure. Click the following purple button to open `llm_service.py`.

Open **llm_service.py** in IDE

This module interfaces with a large language model that can understand both text and images. It's specifically configured to analyze fashion images and generate helpful responses about outfits. In this file, you need to implement the following:

1. In the `__init__` method:

1. Initialize credentials: Set up authentication credentials using the IBM watsonx AI Credentials class with the appropriate region URL and API key.
   Note that the API key is optional in this function because the lab environment allows you to use watson.ai services without requiring one. However, outside of the lab environment, you will need an API key to use watsonx.ai services.
2. Initialize self.client: Create an API client using the APIClient class with the credentials.
3. Define params: Configure text generation parameters using TextChatParameters, setting temperature and top_p values.
4. Initialize self.model: Create a ModelInference object with the model ID, credentials, project ID, and parameters.

2. In the `generate_response` method:

- **Function purpose:**
Generate a textual response from a vision-language model using a base64-encoded image and a user-provided prompt.

- **Implementation details:**

  - Constructs a messages payload combining the user's prompt as text and the encoded image in the appropriate format (data:image/jpeg;base64,···) to comply with the model's input expectations.

  - Sends this payload to the vision-language model's chat endpoint for processing.

  - Extracts the model's textual response from the returned object.

  - Logs the length of both the prompt and the received response to monitor behavior.

  - Includes a safeguard that checks whether the response is unusually long, flagging potential truncation if the content length nears typical model output limits (for example, ~7900 characters).

  - Implements error handling to catch and log exceptions during model interaction, returning a fallback error message if an issue occurs.

- **Expected output:**
Returns the model-generated text based on the given image and prompt. If an error occurs during the request, it returns a descriptive error message instead.

3. In the `generate_fashion_response` method:

- **Function purpose:**
Generate a detailed, fashion-specific response tailored for a professional retail catalog, using a base64-encoded image and dataset-derived information about visually or semantically matched items.

- **Implementation details:**

  - Converts the DataFrame all_items into a markdown-formatted list of item names, prices, and product links.

  - Based on the similarity score, it selects one of two structured prompts:

    - If the score exceeds the threshold, it uses a prompt designed for exact matches, directing the model to perform a professional fashion analysis and append an ITEM DETAILS section.

    - If below the threshold, it uses a prompt tailored for visually similar items, directing the model to clarify that the items are not exact and include a SIMILAR ITEMS section instead.

  - Sends the structured prompt and image to the underlying model via generate_response.

  - Implements two safety checks post-response:

    - If the model's reply is too short (likely incomplete), it creates a basic fallback response using the item descriptions.

    - If the response lacks the required item section (ITEM DETAILS: or SIMILAR ITEMS:), it appends it manually to ensure the information is always included.

- **Expected output:**
Returns a markdown-formatted, catalog-style fashion response including a formal analysis of the clothing and a detailed section listing either matched or similar items. If the model fails to deliver a full response, the function gracefully falls back to basic, readable output to maintain UX consistency.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the `__init__` function

First, implement the function to provides methods to interact with the Llama 3.2 Vision Instruct model. Copy-paste the following code into `__init__` function and save your files.

```
def __init__(self, model_id, project_id, region="us-south",
             temperature=0.2, top_p=0.6, api_key=None, max_tokens=2000):
    """
    Initialize the service with the specified model and parameters.

    Args:
        model_id (str): Unique identifier for the model
        project_id (str): Project ID to associate the task
        region (str): Region for the watsonx AI service
        temperature (float): Controls randomness in generation
        top_p (float): Nucleus sampling parameter
        api_key (str, optional): API key for authentication
        max_tokens (int): Maximum tokens in the response
    """
    # Set up authentication credentials
    credentials = Credentials(
        url=f"https://{region}.ml.cloud.ibm.com",
        api_key=api_key
    )
    self.client = APIClient(credentials)
```

```
# Define parameters for the model's behavior
params = TextChatParameters(
    temperature=temperature,
    top_p=top_p,
    max_tokens=max_tokens
)

# Initialize the model inference object
self.model = ModelInference(
    model_id=model_id,
    credentials=credentials,
    project_id=project_id,
    params=params
)
```

Step 3: Implement the generate_response function

Next, implement the function to generate a response from the model based on an image and prompt. Copy-paste the following code into `generate_response` function and save your files.

```
def generate_response(self, encoded_image, prompt):
    """
    Generate a response from the model based on an image and prompt.

    Args:
        encoded_image (str): Base64-encoded image string
        prompt (str): Text prompt to guide the model's response

    Returns:
        str: Model's response
    """
    try:
        logger.info("Sending request to LLM with prompt length: %d", len(prompt))

        # Create the messages object
        messages = [
            {
                "role": "user",
                "content": [
                    {
                        "type": "text",
                        "text": prompt
                    },
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": "data:image/jpeg;base64," + encoded_image,
                        }
                    }
                ]
            }
        ]

        # Send the request to the model
        response = self.model.chat(messages=messages)

        # Extract and validate the response
        content = response['choices'][0]['message']['content']

        logger.info("Received response with length: %d", len(content))

        # Check if response appears to be truncated
        if len(content) >= 7900:  # Close to common model limits
            logger.warning("Response may be truncated (length: %d)", len(content))

        return content

    except Exception as e:
        logger.error("Error generating response: %s", str(e))
        return f"Error generating response: {e}"
```

Step 4: Implement the generate_fashion_response function

Finally, implement the function to generate a fashion-specific response using role-based prompts. Copy-paste the following code into `generate_fashion_response` function and save your files.

```
def generate_fashion_response(self, user_image_base64, matched_row, all_items,
                              similarity_score, threshold=0.8):
    """
    Generate a fashion-specific response using role-based prompts.

    Args:
```

```
            user_image_base64: Base64-encoded user-uploaded image
            matched_row: The closest match row from the dataset
            all_items: DataFrame with all items related to the matched image
            similarity_score: Similarity score between user and matched images
            threshold: Minimum similarity for considering an exact match

        Returns:
            str: Detailed fashion response
        """
        # Generate a simpler list of items with prices and links
        items_list = []
        for _, row in all_items.iterrows():
            item_str = f"{row['Item Name']} (${row['Price']}): {row['Link']}"
            items_list.append(item_str)

        # Join items with clear separators
        items_description = "\n".join([f"- {item}" for item in items_list])
        if similarity_score >= threshold:
            # Simplified prompt focused on professional fashion analysis
            assistant_prompt = (
                f"You're conducting a professional retail catalog analysis. "
                f"This image shows standard clothing items available in department stores. "
                f"Focus exclusively on professional fashion analysis for a clothing retailer. "
                f"ITEM DETAILS (always include this section in your response):\n{items_description}\n\n"
                "Please:\n"
                "1. Identify and describe the clothing items objectively (colors, patterns, materials)\n"
                "2. Categorize the overall style (business, casual, etc.)\n"
                "3. Include the ITEM DETAILS section at the end\n\n"
                "This is for a professional retail catalog. Use formal, clinical language."
            )
        else:
            # Similar approach for non-exact matches
            assistant_prompt = (
                f"You're conducting a professional retail catalog analysis. "
                f"This image shows standard clothing items available in department stores. "
                f"Focus exclusively on professional fashion analysis for a clothing retailer. "
                f"SIMILAR ITEMS (always include this section in your response):\n{items_description}\n\n"
                "Please:\n"
                "1. Note these are similar but not exact items\n"
                "2. Identify clothing elements objectively (colors, patterns, materials)\n"
                "3. Include the SIMILAR ITEMS section at the end\n\n"
                "This is for a professional retail catalog. Use formal, clinical language."
            )

        # Send the prompt to the model
        response = self.generate_response(user_image_base64, assistant_prompt)

        # Check if response is incomplete
        if len(response) < 100:
            logger.info("Response appears incomplete, creating basic response")
            # Create a basic response with the item details
            section_header = "ITEM DETAILS:" if similarity_score >= threshold else "SIMILAR ITEMS:"
            response = f"# Fashion Analysis\n\nThis outfit features a collection of carefully coordinated pieces.\n\n{section_header}\n{items_description}"

        # Ensure the items list is included - this is crucial
        elif "ITEM DETAILS:" not in response and "SIMILAR ITEMS:" not in response:
            logger.info("Item details section missing from response")
            # Append to existing response
            section_header = "ITEM DETAILS:" if similarity_score >= threshold else "SIMILAR ITEMS:"
            response += f"\n\n{section_header}\n{items_description}"

        return response
```

Click the below button to see the fully updated `llm_service.py`.

▶ Click to see the solution

The importance of prompt engineering

Prompt engineering is the art and science of crafting effective instructions for AI language models to produce desired outputs. As demonstrated in the fashion response function, well-designed prompts are fundamental to achieving reliable, consistent, and useful AI-generated content.

Strategic communication with AI

The fashion response function illustrates how carefully constructed prompts can:

1. Define the AI's role and context - Notice how the prompt begins with "You're conducting a professional retail catalog analysis," establishing a clear professional identity and purpose.
2. Set boundaries and focus - The prompt explicitly states to "Focus exclusively on professional fashion analysis," preventing the AI from veering into unrelated topics.
3. Structure the output - By including numbered instructions and requiring specific sections like "ITEM DETAILS," the prompt ensures consistent formatting and content organization.
4. Control tone and style - The instruction to "Use formal, clinical language" guides the linguistic characteristics of the response.

Failsafe mechanisms

The function also demonstrates how prompt engineering includes building in safeguards:

- The code checks if the response meets minimum quality standards (length check)
- It ensures critical information is always included (verification of item details section)
- It provides fallback options when responses are inadequate

Next, create the helper functions that support our Style Finder application.

Part 4: Implementing the utility functions that support your application

Now implement the utility functions that support our application. These helpers may seem simple, but they're essential glue components that connect your larger modules.

Step 1: Examining the helpers.py starter file

First examine the starter file structure. Click the purple button below to open `helpers.py`.

`Open helpers.py in IDE`

The `process_response` function is already written for you. It is a critical component of the Style Finder application that handles the post-processing of AI-generated fashion analysis responses. This function ensures that even when the AI model behaves unexpectedly, users still receive properly formatted, useful output.

At its core, this function serves as a safety net and formatter that:

- Handles empty responses by providing a fallback message
- Detects and manages AI refusals or rejections
- Extracts valuable information even when the model partially rejects a request
- Standardizes formatting for consistent user experience
- Applies proper Markdown syntax for optimal display

With that, in this file, you need to implement 2 functions:

1. In the `get_all_items_for_image` function:

- **Function purpose:** Retrieve all fashion items associated with a specific image URL from the dataset.
- **Implementation needed:** Create a filtering mechanism that searches the dataset for all rows containing the specified image URL in the 'Image URL' column.
- **Expected output:** Return a DataFrame containing only the rows that match the image URL, which would include all items that are part of the same outfit or collection.

2. In the `format_alternatives_response` function:

- **Function purpose:** Enhance the model's original response by appending a formatted list of visually or semantically similar fashion item alternatives based on similarity score thresholds.
- **Implementation details:**

  - The function first checks if the model's original response contains refusal phrases. If so, it replaces it with a generic header to begin the analysis.

  - It then determines the appropriate section heading ("Similar Items Found") based on whether the similarity score exceeds the provided threshold.

  - For each detected item, the function adds a subheader and lists up to three alternative products pulled from the alternatives dictionary.

  - Each alternative includes the title, price, source, and a purchase link. The total number of items added is capped at 10 to keep the response concise.

  - If no alternatives are found for an item, it appends a placeholder message under that item.

- **Expected output:**
  Returns a fully formatted Markdown string that combines the model's initial response (or a fallback response) with a structured section showcasing up to ten visually or semantically similar fashion item recommendations.

These utility functions will help make your Style Finder application more user-friendly by providing relevant item groupings and presenting alternative options in a helpful, educational format.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the get_all_items_for_image function

Next, implement the function to get all items related to a specific image from the dataset. Copy-paste the following code into `get_all_items_for_image` function and save your files.

```
def get_all_items_for_image(image_url, dataset):
    """
    Get all items related to a specific image from the dataset.

    Args:
        image_url (str): The URL of the matched image
        dataset (DataFrame): Dataset containing outfit information

    Returns:
        DataFrame: All items related to the image
    """
    related_items = dataset[dataset['Image URL'] == image_url]
    logger.info(f"Found {len(related_items)} items related to image URL: {image_url}")
    return related_items
```

Step 3: Implement the format_alternatives_response function

Now, implement the function to append alternatives to the user response in a formatted way. Copy-paste the following code into `format_alternatives_response` function and save your files.

```
def format_alternatives_response(user_response, alternatives, similarity_score, threshold=0.8):
    """
    Append alternatives to the user response in a formatted way.

    Args:
        user_response (str): Original response from the model
        alternatives (dict): Dictionary of alternatives for each item
        similarity_score (float): Similarity score of the match
        threshold (float): Threshold for determining match quality

    Returns:
        str: Enhanced response with alternatives
    """
    # Check if user_response is problematic
    if not user_response or any(phrase in user_response for phrase in [
            "I'm not able to provide",
            "I cannot",
            "I apologize, but",
            "I don't feel comfortable"]):
        # Create a basic response if the model refused
        user_response = "## Fashion Analysis Results\n\nHere are the items detected in your image:"

    if similarity_score >= threshold:
        enhanced_response = user_response + "\n\n## Similar Items Found\n\nHere are some similar items we found:\n"
    else:
        enhanced_response = user_response + "\n\n## Similar Items Found\n\nHere are some visually similar items:\n"

    # Count items added to ensure we're not exceeding reasonable limits
    items_added = 0
    max_items = 10

    for item, alts in alternatives.items():
        enhanced_response += f"\n### {item}:\n"
        if alts:
            for alt in alts[:3]:  # Limit to 3 alternatives per item
                if items_added < max_items:
                    enhanced_response += f"- {alt['title']} for {alt['price']} from {alt['source']} ([Buy it here]({alt['link']}))\n"
                    items_added += 1
        else:
            enhanced_response += "- No alternatives found.\n"

    return enhanced_response
```

Click the below button to see the fully updated `helpers.py`.

▶ Click to see the solution

Now that you've implemented all the individual components, create your application to test and verify that your basic RAG pipeline works correctly.

Part 5: Building the application

Implement a streamlined version of your app that processes images, finds matches in your dataset, and generates fashion responses using the Llama model.

Step 1: Examining the app.py Starter File

First, examine the starter file structure. Click the purple button below to open `app.py`.

Open **app.py** in IDE

In the starter template for app.py, you need to implement several key functions to create a working application that can analyze fashion images. You'll focus on:

1. The `__init__` method in StyleFinderApp

This initialization method needs to:

1. Load the dataset from the provided path
2. Handle potential errors (file not found, empty dataset)
3. Initialize the core components of our RAG pipeline
4. Set up the image processor with the configuration parameters
5. Set up the LLM service for generating fashion descriptions

The implementation should properly handle the dataset loading while maintaining a clean error handling approach.

2. The `process_image` method in StyleFinderApp

- **Function purpose:**
  Handle a full image analysis pipeline—taking a user-uploaded image and returning a fashion-oriented response with analysis and recommendations, using a series of subcomponents like encoding, matching, and language modeling.

- **Implementation details:**

  - Image input handling:

    - Accepts a PIL image (from Gradio or other UI).
    - If the image isn't already a file path, it temporarily saves it to disk to enable downstream processing.

  - Step 1 – Encode image:

    - Calls encode_image to generate a base64 string and feature vector for the uploaded image.
    - If encoding fails (that is, vector is None), an error message is returned.

  - Step 2 – Find closest match:

    - Uses find_closest_match to compare the user's image vector to precomputed vectors in self.data and retrieve the most similar item.
    - If no match is found, an error is returned.

  - Step 3 – Retrieve related items:

    - Calls get_all_items_for_image using the matched image's URL to gather all related fashion items (for example, parts of the same outfit).

    - If no related items are found, it returns an appropriate error message.

  - Step 4 – Generate fashion analysis:

    - Passes the base64 image, matched row, all related items, and similarity score into generate_fashion_response, which builds a structured, catalog-style analysis response via LLM.

  - Cleanup:

    If a temporary file was created for the image, it is deleted after processing to free up space.

  - Final output:

    Passes the raw response from the LLM into process_response for optional post-processing (for example, markdown formatting, link validation), and returns the final formatted response.

- **Expected output:**
  A well-formatted, markdown-compatible string that includes an LLM-generated fashion analysis of the user-uploaded image, along with related product recommendations. If any part of the pipeline fails, the function returns a graceful, user-friendly error message.

3. The `create_gradio_interface` function

This function creates the user interface for your application:

1. Set up the Gradio Blocks interface with an appropriate theme and title
2. Add an introduction section explaining the application's purpose
3. Create a section for example images to help users understand what kinds of photos work well
4. Implement buttons to load example images for quick testing
5. Add the main upload component for user images

6. Create a primary analysis button with appropriate styling
7. Add a status indicator to provide feedback during processing
8. Set up the output component to display formatted analysis results
9. Configure event handlers to connect user actions to application functions
10. Add informational sections explaining how the technology works

By implementing these core functions, you'll create a complete RAG system that can analyze fashion images, find similar items in our dataset, and provide detailed fashion descriptions - all without requiring external API integration.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the init function

First, implement the function that initializes the Style Finder application. Copy-paste the following code into `__init__` function and save your files.

```python
def __init__(self, dataset_path):
    """
    Initialize the Style Finder application.

    Args:
        dataset_path (str): Path to the dataset file

    Raises:
        FileNotFoundError: If the dataset file is not found
        ValueError: If the dataset is empty or invalid
    """
    # Load the dataset
    if not os.path.exists(dataset_path):
        raise FileNotFoundError(f"Dataset file not found: {dataset_path}")

    self.data = pd.read_pickle(dataset_path)
    if self.data.empty:
        raise ValueError("The loaded dataset is empty")

    # Initialize components
    self.image_processor = ImageProcessor(
        image_size=config.IMAGE_SIZE,
        norm_mean=config.NORMALIZATION_MEAN,
        norm_std=config.NORMALIZATION_STD
    )

    self.llm_service = LlamaVisionService(
        model_id=config.LLAMA_MODEL_ID,
        project_id=config.PROJECT_ID,
        region=config.REGION
    )
```

Step 3: Implement the process_image function

Now, implement the function that processes a user-uploaded image and generate a basic fashion response. Copy-paste the following code below `process_image` function and save your files.

```python
def process_image(self, image):
    """
    Process a user-uploaded image and generate a fashion response.

    Args:
        image: PIL image uploaded through Gradio

    Returns:
        str: Formatted response with fashion analysis
    """
    # Save the image temporarily if it's not already a file path
    if not isinstance(image, str):
        temp_file = NamedTemporaryFile(delete=False, suffix=".jpg")
        image_path = temp_file.name
        image.save(image_path)
    else:
        image_path = image

    # Step 1: Encode the image
    user_encoding = self.image_processor.encode_image(image_path, is_url=False)
    if user_encoding['vector'] is None:
        return "Error: Unable to process the image. Please try another image."

    # Step 2: Find the closest match
    closest_row, similarity_score = self.image_processor.find_closest_match(user_encoding['vector'], self.data)
    if closest_row is None:
        return "Error: Unable to find a match. Please try another image."

    print(f"Closest match: {closest_row['Item Name']} with similarity score {similarity_score:.2f}")

    # Step 3: Get all related items
    all_items = get_all_items_for_image(closest_row['Image URL'], self.data)
    if all_items.empty:
        return "Error: No items found for the matched image."
```

```
# Step 4: Generate fashion response
bot_response = self.llm_service.generate_fashion_response(
    user_image_base64=user_encoding['base64'],
    matched_row=closest_row,
    all_items=all_items,
    similarity_score=similarity_score,
    threshold=config.SIMILARITY_THRESHOLD
)

# Clean up temporary file
if not isinstance(image, str):
    try:
        os.unlink(image_path)
    except:
        pass

return process_response(bot_response)
```

Step 4: Implement the create_gradio_interface function

Now, implement the function that creates and configures the Gradio interface. Copy-paste the code below into `create_gradio_interface` function and save your files.

```python
def create_gradio_interface(app):
    """
    Create and configure the Gradio interface.

    Args:
        app (StyleFinderApp): Instance of the StyleFinderApp

    Returns:
        gr.Blocks: Configured Gradio interface
    """
    with gr.Blocks(theme=gr.themes.Soft(), title="Fashion Style Analyzer") as demo:
        # Introduction
        gr.Markdown(
            """
            # Fashion Style Analyzer

            Upload an image to analyze fashion elements and get detailed information about the items.
            This application combines computer vision, vector similarity, and large language models
            to provide detailed fashion analysis.
            """
        )

        # Example images section - moved higher up
        gr.Markdown("### Example Images")
        with gr.Row():
            # Display the images directly
            gr.Image(value="examples/test-1.png", label="Example 1", show_label=True, scale=1)
            gr.Image(value="examples/test-2.png", label="Example 2", show_label=True, scale=1)
            gr.Image(value="examples/test-3.png", label="Example 3", show_label=True, scale=1)

        # Example image buttons
        with gr.Row():
            example1_btn = gr.Button("Use Example 1")
            example2_btn = gr.Button("Use Example 2")
            example3_btn = gr.Button("Use Example 3")

        with gr.Row():
            with gr.Column(scale=1):
                # Image input component
                image_input = gr.Image(
                    type="pil",
                    label="Upload Fashion Image"
                )

                # Submit button
                submit_btn = gr.Button("Analyze Style", variant="primary")

                # Status indicator
                status = gr.Markdown("Ready to analyze.")

            with gr.Column(scale=2):
                # Output markdown component for displaying analysis results
                output = gr.Markdown(
                    label="Style Analysis Results",
                    height=700
                )

        # Event handlers
        # 1. Submit button click with processing indicator
        submit_btn.click(
            fn=lambda: "Analyzing image... This may take a few moments.",
            inputs=None,
            outputs=status
        ).then(
            fn=app.process_image,
            inputs=[image_input],
            outputs=output
        ).then(
            fn=lambda: "Analysis complete!",
            inputs=None,
            outputs=status
        )

        # 2. Example image buttons
```

```
example1_btn.click(
    fn=lambda: "examples/test-1.png",
    inputs=None,
    outputs=image_input
).then(
    fn=lambda: "Example 1 loaded. Click 'Analyze Style' to process.",
    inputs=None,
    outputs=status
)

example2_btn.click(
    fn=lambda: "examples/test-2.png",
    inputs=None,
    outputs=image_input
).then(
    fn=lambda: "Example 2 loaded. Click 'Analyze Style' to process.",
    inputs=None,
    outputs=status
)

example3_btn.click(
    fn=lambda: "examples/test-3.png",
    inputs=None,
    outputs=image_input
).then(
    fn=lambda: "Example 3 loaded. Click 'Analyze Style' to process.",
    inputs=None,
    outputs=status
)

# Information about the application
gr.Markdown(
    """
    ### About This Application

    This system analyzes fashion images using:

    - **Image Encoding**: Converting fashion images into numerical vectors
    - **Similarity Matching**: Finding visually similar items in a database
    - **Advanced AI**: Generating detailed descriptions of fashion elements

    The analyzer identifies garments, fabrics, colors, and styling details from images.
    The database includes information on outfits with brand and pricing details.
    """
)

return demo
```

Click the below button to see the fully updated `app.py`.

▶ Click to see the solution

Launching the application

Return to the terminal and verify that the virtual environment `venv` label appears at the start of the line. This means that you are in the `venv` environment that you just created. Then, you can run the Gradio app by running the below command in the terminal.

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

Since the web application is hosted locally on port 5000, click the following button to view the application you've developed.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.

Once you launch your application. A window opens and you should be able to see the application view similar to the following example:

To stop execution of `app.py` in addition to closing the application tab, press `Ctrl+C` in terminal.

Conclusion and next steps

Conclusion

Congratulations on completing the Style Finder lab! You've successfully built a powerful application that demonstrates the power of combining multimodal AI, vector similarity, and LLMs to create an educational and practical fashion application. The system successfully identifies clothing items, analyzes their style characteristics, and provides users with detailed information including brand details and pricing.

As a final step, feel free to:

- Test the analyzer with different fashion images to see how it adapts to various styles and clothing combinations
- Experiment with different vision models and parameter settings to compare their performance on fashion analysis
- Refine the prompts to generate even more detailed and professional fashion descriptions
- Expand your database with additional fashion items to increase the range of matches, or extend to real-time searches using APIs
- Deploy your application to a cloud platform to make it accessible to fashion enthusiasts in your network

Author(s)

[Hailey Quach](#)