CS331 Project: Retrieval Augmented Generation

Assignment 6

Team Members:
Khushi Mandal - 2201108
Arya Sahu - 2201033
Anushka Srivastava - 2201030
Ahlad Pataparla - 2201017

March 20, 2025

Q1. Core Functional Modules (Business Logic Layer)

This section details the core functional modules of the application, focusing on the business logic layer (BLL). It describes the purpose of each module, provides code snippets, and shows how they interact. The interaction diagram is provided at the end of this section.

1. Product Retrieval

Module: `get_item(item_id)`

Purpose: Retrieves a single product from the preloaded Pandas DataFrame (which is loaded from the database during application startup) based on its ID. This function also constructs the image URL for the frontend.

Code:

```python
def get_item(item_id: str) -> Dict[str, Any]:
    """Retrieve an item from the preloaded dataframe."""
    try:
        item = ml_model.df[ml_model.df["id"] == item_id].iloc[0].to_dict()
        item["id"] = int(item["id"])
        item["image_url"] = f"/static/images/{item['id']}.jpg"
        return item
    except IndexError:
        raise HTTPException(status_code=404, detail="Item not found")
```

Interaction:

- Called by: `product_page`, `get_random_products`
- Interacts with: The preloaded `ml_model.df` (Pandas DataFrame). No direct database interaction, as the data is loaded at startup.

2. Random Product Retrieval

Module: `get_random_products(limit=10)`

Purpose: Retrieves a specified number of random products from the preloaded DataFrame. This is used for displaying a selection of products on the homepage or other sections where a diverse set of items is needed.

Code:

```python
@app.get("/api/products", response_model=ProductsResponse)
async def get_random_products(limit: int = 10):
    """Return a random selection of products."""
    try:
        all_product_ids = ml_model.df["id"].tolist()
        selected_ids = sample(all_product_ids, min(limit, len(all_product_ids)))
        products = [Item(**get_item(product_id)) for product_id in selected_ids]
        return ProductsResponse(products=products)
    except Exception as e:
        print(f"Error fetching random products: {e}")
        raise HTTPException(status_code=500, detail="An error occurred while fetching random products")
```

Interaction:

- Called by: Frontend (e.g., homepage).
- Interacts with: `get_item` (to retrieve details of each selected product), `ml_model.df`.

3. Attribute Prediction

Module: `predict_attributes(image)`

Purpose: Takes an image as input and uses the CLIP model to predict various attributes of the item in the image. These attributes include gender, article type, season, usage, master category, subcategory, and base color. The dominant color is extracted from the image, and the closest color name is found using a predefined color map.

Code:

```python
def get_dominant_color(image: Image.Image) -> np.ndarray:
    """Get the dominant color from an image."""
    image = image.resize((100, 100))
    img_array = np.array(image).reshape(-1, 3)
    kmeans = KMeans(n_clusters=3, random_state=0).fit(img_array)
    counts = np.bincount(kmeans.labels_)
    return kmeans.cluster_centers_[np.argmax(counts)]

def find_closest_color(target_color: np.ndarray, color_names: List[str]) -> str:
    """Find the closest color name to the target RGB."""
    color_map = {
        "Navy Blue": (0, 0, 128),
        "Blue": (0, 0, 255),
        "Black": (0, 0, 0),
        "Silver": (192, 192, 192),
        "Grey": (128, 128, 128),
                # ... (rest of the color map)
        "Fluorescent Green": (127, 255, 0),
    }

    target_rgb = target_color.astype(int)
```

```python
    min_dist, closest = float("inf"), "Black"
    for name, rgb in color_map.items():
        dist = np.linalg.norm(np.array(rgb) - target_rgb)
        if dist < min_dist:
            min_dist, closest = dist, name
    return closest

def predict_attributes(image: Image.Image) -> dict:
    """Predict attributes from an image using CLIP."""
    attributes = {}
    for label_type, labels in [
        ("gender", ["Men", "Women", "Boys", "Girls", "Unisex"]),
        ("articleType", ml_model.df["articleType"].unique().tolist()),
        ("season", ["Summer", "Winter", "Spring", "Fall"]),
        ("usage", ["Casual", "Ethnic", "Formal", "Sports", "Smart Casual", "Travel", "Party", "Home"]),
        ("masterCategory", ["Apparel", "Accessories", "Footwear", "Personal Care", "Free Items", "Sporting Goods", "Home"]),
        ("subCategory", ["Topwear", "Bottomwear", "Watches", "Socks", "Shoes", "Belts", "Flip Flops", "Bags", "Innerwear", "Sandal", "Shoe Accessories", "Fragrance", "Jewellery", "Lips", "Saree", "Eyewear", "Nails", "Scarves", "Dress", "Loungewear and Nightwear", "Wallets", "Apparel Set", "Headwear", "Mufflers", "Skin Care", "Makeup", "Free Gifts", "Ties", "Accessories", "Skin", "Beauty Accessories", "Water Bottle", "Eyes", "Bath and Body", "Gloves", "Sports Accessories", "Cufflinks", "Sports Equipment", "Stoles", "Hair", "Perfumes", "Home Furnishing", "Umbrellas", "Wristbands", "Vouchers"])
    ]:
        inputs = ml_model.clip_processor(text=labels, images=image, return_tensors="pt", padding=True)
        outputs = ml_model.clip_model(**inputs)
        attributes[label_type] = labels[outputs.logits_per_image.softmax(dim=1).argmax().item()]

    dominant_color = get_dominant_color(image)
    attributes["baseColour"] = find_closest_color(dominant_color, ml_model.df["baseColour"].unique().tolist())
    return attributes
```

Interaction:

- Called by: `recommend_from_image`
- Interacts with: `ml_model.clip_model`, `ml_model.clip_processor`, `get_dominant_color`, `find_closest_color`.

4. Outfit Recommendation

Modules:

- `product_page(item_id)`: Provides recommendations for a specific product page.
- `recommend_from_image(file)`: Provides recommendations based on an uploaded image.
- `get_ml_recommendations(...)`: The core recommendation engine (called by both of the above).

- `get_compatible_types(article_type)`: Retrieves compatible types from ‘constants.py‘.
- `get_accessory_types(usage, season)`: Retrieves accessory types from ‘constants.py‘.
- `check_negative_constraints(target_item, candidate_item)`: Filters out incompatible combinations.
- `maximal_marginal_relevance(...)`: Ensures diversity in recommendations.
- `color_compatibility(color1, color2)`: Calculates color compatibility.

Purpose: These modules work together to generate outfit recommendations. `product_page` and `recommend_from_image` are the API endpoints, while `get_ml_recommendations` performs the core recommendation logic using cosine similarity, MMR, and various filtering rules. The helper functions retrieve compatibility data and enforce constraints.

Code:

```python
def maximal_marginal_relevance(target_features, category_features, top_n: int, lambda_param: float = 0.1) -> np.ndarray:
    """Select highly diverse items using MMR."""
    selected_indices = []
    remaining_indices = list(range(category_features.shape[0]))

    similarities = cosine_similarity(target_features, category_features).flatten()
    first_idx = np.argmax(similarities)
    selected_indices.append(first_idx)
    remaining_indices.remove(first_idx)

    # Dynamic lambda: more diversity for larger pools, min 0.05
    dynamic_lambda = max(0.05, lambda_param - (len(remaining_indices) / 2000))

    for _ in range(min(top_n - 1, len(remaining_indices))):
        mmr_scores = []
        for idx in remaining_indices:
            relevance = similarities[idx]
            diversity = min(cosine_similarity(category_features[idx], category_features[selected_indices]).flatten())
            mmr_score = dynamic_lambda * relevance - (1 - dynamic_lambda) * diversity
            mmr_scores.append(mmr_score)

        next_idx = remaining_indices[np.argmax(mmr_scores)]
        selected_indices.append(next_idx)
        remaining_indices.remove(next_idx)

    return np.array(selected_indices)

def get_ml_recommendations(
    target_features,
    target_article_type: str,
    product_gender: str,
    target_color: str,
    target_id: Optional[str] = None,
    top_n: int = 3
) -> tuple[List[Dict], float, float, float]:
```

```python
    """Generate recommendations with maximum diversity."""
    df = ml_model.df
    # Broaden pool with ARTICLE_TYPE_GROUPS and COMPATIBLE_TYPES
    target_group = next((group for group, types in ARTICLE_TYPE_GROUPS.items() if target_article_type in types), None)
    compatible_types = COMPATIBLE_TYPES.get(target_article_type, [])
    if target_group:
        candidate_types = list(set(ARTICLE_TYPE_GROUPS[target_group] + compatible_types))
    else:
        candidate_types = [target_article_type] + compatible_types

    mask = (df["articleType"].isin(candidate_types)) & (df["gender"].isin([product_gender, "Unisex"]))
    if target_id:
        mask &= (df["id"] != target_id)

    category_df = df[mask]
    logger.info(f"Initial filter: {len(category_df)} items for {target_article_type}, gender: {product_gender}")

    if category_df.empty:
        logger.warning(f"No items found for {target_article_type}")
        return [], 0.0, 0.0, 0.0

    color_scores = category_df["baseColour"].apply(lambda x: color_compatibility(target_color, x))
    min_threshold = 0.2 if len(category_df[color_scores >= 0.2]) >= top_n * 2 else 0.0
    category_df = category_df[color_scores >= min_threshold]
    logger.info(f"After color filter (threshold {min_threshold}): {len(category_df)} items")

    if category_df.empty:
        return [], 0.0, 0.0, 0.0

    category_indices = category_df.index.tolist()
    category_features = ml_model.combined_features[category_indices]
    color_matches = (category_df["baseColour"] == target_color).astype(float)

    similarities = cosine_similarity(target_features, category_features).flatten()
    similarities += similarities.max() * 0.05 * color_matches

    # Use MMR with maximum diversity emphasis
    top_indices = maximal_marginal_relevance(target_features, category_features, top_n, lambda_param=0.1)

    # Ensure at least two distinct colors
    results = []
    color_set = set()
    selected_positions = []
    for idx in top_indices:
        item = category_df.iloc[idx].to_dict()
        if len(color_set) < 2 or item["baseColour"] in color_set:
            results.append(item)
            color_set.add(item["baseColour"])
```

```python
                selected_positions.append(idx)
            if len(results) == top_n:
                break

    # Fill with randomized remaining indices for variety
    if len(results) < top_n:
        remaining_indices = [i for i in top_indices if i not in selected_positions]
        shuffle(remaining_indices)  # Randomize for diversity
        for idx in remaining_indices:
            item = category_df.iloc[idx].to_dict()
            results.append(item)
            if len(results) == top_n:
                break

    for item in results:
        item["image_url"] = f"/static/images/{item['id']}.jpg"
        item["id"] = int(item["id"])

    logger.info(f"Recommended items: {[item['id'] for item in results]}")
    logger.info(f"Item details: {[f'{item['articleType']} - {item['baseColour']}' for item in results]}")

    novelty_score = inverse_popularity_score(results)
    diversity_score = intra_list_diversity(results)
    serendipity_score = serendipity_measure(results, ml_model.df.iloc[ml_model.id_to_index[target_id]].to_dict()) if target_id else 0.0

    logger.info(f"Recommended {len(results)} items with novelty: {novelty_score}, diversity: {diversity_score}, serendipity: {serendipity_score}")
    return results, novelty_score, diversity_score, serendipity_score

def check_negative_constraints(target_item: dict, candidate_item: dict) -> bool:
    """Check for incompatible combinations based on updated ARTICLE_TYPE_GROUPS."""
    def get_group(article_type: str) -> Optional[str]:
        for group_name, types in ARTICLE_TYPE_GROUPS.items():
            if article_type in types:
                return group_name
        return None

    target_group = get_group(target_item["articleType"])
    candidate_group = get_group(candidate_item["articleType"])

    if target_group is None or candidate_group is None:
        return True  # Assume compatible if not in any group

    # General rule: items from the same group are incompatible, except for accessories
    if target_group == candidate_group and target_group not in ["Accessories", "Jewellery", "Bags", "Makeup", "Skincare", "Bath and Body", "Haircare", "Fragrance", "Tech Accessories", "Home Decor"]:
        return False
```

```python

    # Specific rules
    if target_group == "Trousers":
        if candidate_group in ["Casual Shoes", "Formal Shoes"]:
            if candidate_item["articleType"] in ["Sandals", "Flip Flops"] and target_item["usage"] != "Casual":
                return False
    if target_group in ["Shirts", "Tshirts", "Tops", "Dresses", "Suits"]:
        if candidate_group in ["Shirts", "Tshirts", "Tops", "Dresses", "Suits"]:
            return False
    if candidate_group == "Casual Shoes" and target_item["usage"] == "Formal":
        if candidate_item["articleType"] in ["Flip Flops", "Sports Sandals"]:
            return False
    # Add more specific rules as needed

    return True

def get_compatible_types(article_type: str) -> List[str]:
    """Get compatible article types from COMPATIBLE_TYPES."""
    return COMPATIBLE_TYPES.get(article_type, [])

def get_accessory_types(usage: str, season: str) -> List[str]:
    """Get accessory types based on usage and season."""
    accessory_list = ACCESSORY_COMBINATIONS.get(usage, []) + SEASONAL_ACCESSORIES.get(season, [])
    return list(set(accessory_list))


def color_compatibility(color1: str, color2: str) -> float:
    """Calculate color compatibility score using COLOR_COMPATIBILITY dictionary."""
    if color1 == color2:
        return 1.0
    if color2 in COLOR_COMPATIBILITY.get(color1, []):
        return 0.8
    return 0.0
@app.get("/api/product/{item_id}", response_model=ProductPageResponse)
async def product_page(item_id: str):
    """Get product details and outfit recommendations."""
    product = get_item(item_id)
    target_id = str(product["id"])
    target_gender, target_usage, target_season = product["gender"], product["usage"], product["season"]
    target_color, target_article_type = product["baseColour"], product["articleType"]

    target_features = ml_model.combined_features[ml_model.id_to_index[target_id]]
    compatible_types_list = get_compatible_types(target_article_type)
    accessory_types = get_accessory_types(target_usage, target_season)

    recommendations_dict = {}
    for compatible_type in compatible_types_list:
        recs, _, _, _ = get_ml_recommendations(target_features,
```

```
            compatible_type, target_gender, target_color, target_id)
        filtered_recs = [item for item in recs if check_negative_constraints(product, item)]
        if filtered_recs:
            recommendations_dict[compatible_type] = [Item(**item) for item in filtered_recs[:3]]

    for accessory_type in accessory_types:
        recs, _, _, _ = get_ml_recommendations(target_features, accessory_type, target_gender, target_color, target_id)
        filtered_recs = [item for item in recs if check_negative_constraints(product, item)]
        if filtered_recs:
            recommendations_dict[accessory_type] = [Item(**item) for item in filtered_recs[:3]]

    return ProductPageResponse(product=Item(**product), recommendations=OutfitRecommendation(recommendations=recommendations_dict))

@app.post("/api/recommend-from-image", response_model=OutfitRecommendation)
async def recommend_from_image(file: UploadFile = File(...)):
    """Recommend outfits based on an uploaded image."""
    try:
        contents = await file.read()
        image = Image.open(BytesIO(contents)).convert("RGB")
        attributes = predict_attributes(image)

        synthetic_name = f"{attributes.get('gender', 'Unisex')}'s {attributes.get('baseColour', '')} {attributes.get('articleType', 'Fashion Item')}"
        categorical_data = [attributes.get(col, "Unknown") for col in ["gender", "masterCategory", "subCategory", "articleType", "baseColour", "season", "usage"]]
        onehot = ml_model.onehot_encoder.transform([categorical_data])
        tfidf = ml_model.tfidf_vectorizer.transform([synthetic_name])
        target_features = hstack([onehot, tfidf])

        target_article_type = attributes.get("articleType", "Shirts")
        target_gender, target_usage = attributes.get("gender", "Unisex"), attributes.get("usage", "Casual")
        target_season, target_color = attributes.get("season", "Summer"), attributes.get("baseColour", "Black")

        compatible_types_list = get_compatible_types(target_article_type)
        accessory_types = get_accessory_types(target_usage, target_season)

        recommendations_dict = {}
        for compatible_type in compatible_types_list:
            recs,_,_,_ = get_ml_recommendations(target_features, compatible_type, target_gender, target_color)
            filtered_recs = [item for item in recs[0] if check_negative_constraints(attributes, item)]

            if filtered_recs:
                recommendations_dict[compatible_type] = [Item(**item)
```

```
                    for item in filtered_recs]

        for accessory_type in accessory_types:
            recs,_,_,_ = get_ml_recommendations(target_features, accessory_type, target_gender, target_color)
            filtered_recs = [item for item in recs[0] if check_negative_constraints(attributes, item)]

            if filtered_recs:
                recommendations_dict[accessory_type] = [Item(**item) for item in filtered_recs]

        return OutfitRecommendation(recommendations=recommendations_dict)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error processing image: {str(e)}")
```

Interaction: A complex interaction between multiple modules. Key interactions include:

- `product_page` and `recommend_from_image` call `get_ml_recommendations` to get recommendations.
- `get_ml_recommendations` uses `ml_model.combined_features` (precomputed feature vectors), `cosine_similarity`, and `maximal_marginal_relevance` for ranking.
- `get_compatible_types` and `get_accessory_types` (from `constants.py`) provide rule-based filtering.
- `check_negative_constraints` enforces additional compatibility rules.

5. Search

Module: `search(query)`

Purpose: Performs a semantic search using ChromaDB and OpenCLIP embeddings. This allows users to search for products using natural language queries. The results are returned as a list of image URLs and their corresponding distances (similarity scores).

Code:

```
@app.post("/api/search", response_model=SearchResult)
async def search(query: str = Form(...)):
    """Search for images based on a text query."""
    try:
        fashion_collection = ml_model.chroma_client.get_collection("fashion", embedding_function=OpenCLIPEmbeddingFunction(), data_loader=ImageLoader())
        results = fashion_collection.query(query_texts=[query], n_results=5, include=["uris", "distances"])

        results["uris"] = [[uri.replace("/kaggle/input/fashion-product-images-dataset/fashion-dataset/", "") for uri in results["uris"][0]]]
        image_data = [
            {"id": results["ids"][0][i], "distance": results["distances"][0][i], "image_url": f"/static/images/{os.path.basename(results['uris'][0][i])}"}
```

```
            for i in range(len(results["ids"][0]))
            if os.path.exists(os.path.join(STATIC_DIR, "images", os.path.basename(results["uris"][0][i])))
        ]
        return SearchResult(images=image_data)
    except Exception as e:
        print(f"Error during search: {e}")
        raise HTTPException(status_code=500, detail="An error occurred during search")
```

Interaction:

- Called by: Frontend (search bar).
- Interacts with: `ml_model.chroma_client` (ChromaDB).

6. Evaluation

Module:`evaluate_recommendations()`

Purpose: This module provides an endpoint to evaluate the performance of the recommendation system. It compares the ML-based recommendations with popularity-based and random baselines using novelty, diversity, and serendipity metrics.

Code:

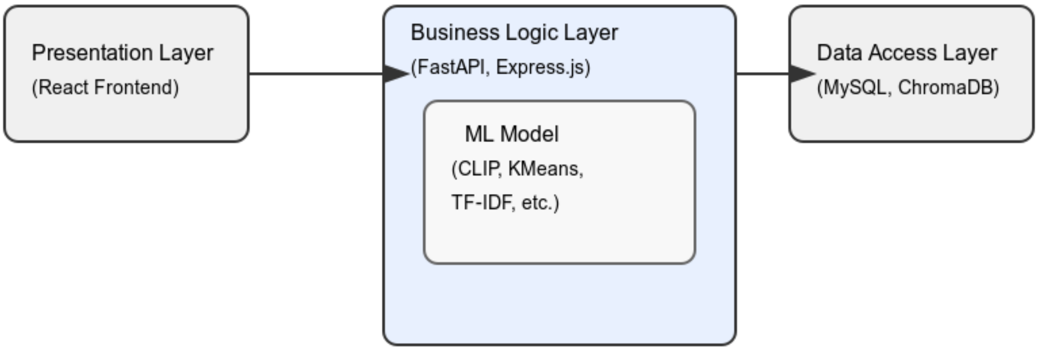
```
def popularity_based_recommender(top_n=5):
    popularity = ml_model.df['articleType'].value_counts().index[:top_n]
    return ml_model.df[ml_model.df['articleType'].isin(popularity)].sample(top_n).to_dict('records')

def random_recommender(top_n=5):
    return ml_model.df.sample(top_n).to_dict('records')

def inverse_popularity_score(recommendations):
    total_purchases = len(ml_model.df)
    scores = [1 - (ml_model.df['articleType'].value_counts()[item['articleType']] / total_purchases) for item in recommendations]
    return np.mean(scores)

def intra_list_diversity(recommendations):
    features = [ml_model.combined_features[ml_model.id_to_index[str(item['id'])]] for item in recommendations]
    stacked_features = vstack(features).toarray()
    similarities = cosine_similarity(stacked_features)
    return 1 - np.mean(similarities)

def serendipity_measure(recommendations, user_history):
    user_features = ml_model.combined_features[ml_model.id_to_index[str(user_history['id'])]]
    rec_features = [ml_model.combined_features[ml_model.id_to_index[str(item['id'])]] for item in recommendations]
    distances = [cosine_similarity(user_features, rec.reshape(1, -1))[0][0] for rec in rec_features]
    return np.mean(distances)

@app.get("/api/evaluate")
async def evaluate_recommendations():
```

```python
    # Select a random product as the target
    target_product = ml_model.df.sample(1).iloc[0]
    target_id = str(target_product['id'])
    target_features = ml_model.combined_features[ml_model.id_to_index[target_id]]

    # Get recommendations from different methods
    ml_recs, ml_novelty, ml_diversity, ml_serendipity = get_ml_recommendations(
        target_features, target_product['articleType'], target_product['gender'],
        target_product['baseColour'], target_id
    )

    popularity_recs = popularity_based_recommender()
    random_recs = random_recommender()

    # Calculate metrics for baseline methods
    pop_novelty = inverse_popularity_score(popularity_recs)
    pop_diversity = intra_list_diversity(popularity_recs)
    pop_serendipity = serendipity_measure(popularity_recs, target_product)

    rand_novelty = inverse_popularity_score(random_recs)
    rand_diversity = intra_list_diversity(random_recs)
    rand_serendipity = serendipity_measure(random_recs, target_product)

    return {
        "ML Model": {
            "Novelty": ml_novelty,
            "Diversity": ml_diversity,
            "Serendipity": ml_serendipity
        },
        "Popularity Baseline": {
            "Novelty": pop_novelty,
            "Diversity": pop_diversity,
            "Serendipity": pop_serendipity
        },
        "Random Baseline": {
            "Novelty": rand_novelty,
            "Diversity": rand_diversity,
            "Serendipity": rand_serendipity
        }
    }
```

Interaction:

- Called by: An external script or tool for evaluation.
- Interacts with: `get_ml_recommendations`, helper functions for calculating metrics.

Interaction Diagram (Simplified)



Flow: A simplified representation of the core flow. Note that the diagram does not show every single function call, but rather the high-level interaction between layers and major components.

Q2. Business Rules, Validation, and Data Transformation

A. Business Rules

The application implements several business rules to ensure the quality and relevance of recommendations and to maintain data integrity. These rules are primarily enforced within the `get_ml_recommendations` function and related helper functions.

- **Compatibility Rules:** Ensures that recommended outfit components are compatible with each other. This is primarily handled by the `COMPATIBLE_TYPES` dictionary in `constants.py`, which defines allowed combinations of article types (e.g., "Shirts" are compatible with "Trousers", "Jeans", etc.). The `get_compatible_types` function retrieves these rules.
- **Gender, Usage, and Seasonal Filtering:** Recommendations are filtered based on the gender, usage, and season of the target item. This ensures that, for example, a formal shirt for men will not be recommended with casual shorts. This is implemented in `get_ml_recommendations` using a filtering mask and in the `get_accessory_types` function.
- **Accessory Rules:** The `get_accessory_types` function uses the `ACCESSORY_COMBINATIONS` and `SEASONAL_ACCESSORIES` dictionaries to filter the accessories based on the target item's usage and season.
- **Color Compatibility:** The `color_compatibility` function calculates a score based on the `COLOR_COMPATIBILITY` dictionary, preventing clashing colors in recommendations. Items with low color compatibility scores are filtered out.
- **Negative Constraints:** The `check_negative_constraints` function implements specific rules to prevent incompatible item pairings that might not be captured by the broader compatibility rules (e.g., preventing sandals from being recommended with

formal trousers, even if ”Casual Shoes” are generally compatible with ”Trousers”). This uses the `ARTICLE_TYPE_GROUPS` dictionary.

- **Diversity Enforcement (MMR):** The `maximal_marginal_relevance` function is used to ensure diversity among the top recommendations. This prevents the system from recommending very similar items (e.g., multiple shirts of the same color and style).

B. Validation Logic

Validation is performed at multiple levels to ensure data integrity and prevent errors.

- **Input Checks:** Basic checks for required fields and data types are performed using Pydantic models in the API endpoints (e.g., `Item`, `OutfitRecommendation`, `ProductPageResponse`, etc.).
- **Dataframe-Level Validation:** The `get_item` function includes error handling (raising an `HTTPException`) if an item with the requested ID is not found in the preloaded DataFrame.
- **Empty Result Checks:** The `get_ml_recommendations` function includes checks to ensure that the filtered DataFrame is not empty *after* applying filters (e.g., compatibility, color, negative constraints). This prevents errors later in the recommendation process and returns an empty list if no suitable recommendations are found.
- **Logging:** The `get_ml_recommendations` function utilizes the `logger` object (from Python's `logging` module) extensively. This provides valuable insights into the filtering process, including:
  - The number of items remaining after each filtering step (initial filter, color filter).
  - The recommended item IDs and details (article type and color).
  - The calculated novelty, diversity, and serendipity scores.
  - Warnings if no items are found for a particular article type.

  This detailed logging is crucial for debugging, understanding the recommendation process, and identifying potential issues (e.g., overly restrictive filters).

C. Data Transformation

Data transformation is a crucial part of the application, converting raw data into formats suitable for the ML model and the frontend.

- **SQL Result (Database) → Pandas DataFrame → Python Dictionary:** The `load_data` function retrieves data from the MySQL database using SQLAlchemy and converts it into a Pandas DataFrame. The `get_item` function then converts a single row of this DataFrame into a Python dictionary, making it easier to work with. This transformation also includes adding the `image_url` field.
- **One-Hot Encoding and TF-IDF Vectorization:** The `preprocess_data` function transforms categorical features (gender, masterCategory, etc.) into numerical vectors using one-hot encoding. It also transforms the product display name into a TF-IDF

vector. These numerical representations are essential for the cosine similarity calculations used in the recommendation engine. The combined features are stored in `ml_model.combined_features`.

- **CLIP Output → Predicted Attributes:** The `predict_attributes` function uses the CLIP model to predict attributes from an image. The raw output of the CLIP model (logits) is converted into predicted labels (e.g., "Men", "Summer", "Casual") using softmax and argmax. The dominant color is extracted and mapped to the closest color name using a predefined mapping.
- **Image Path → URL:** The `get_item` function and other recommendation functions construct the image URL (`/static/images/{item_id}.jpg`) from the item ID, which is used by the frontend to display the product images.
- **Implicit Transformations within Similarity Calculations:** The `cosine_similarity` function itself performs a transformation. It takes the pre-processed, numerical feature vectors (one-hot encoded and TF-IDF) and calculates the cosine similarity between them. This similarity score is then used to rank the recommendations. The `maximal_marginal_relevance` function builds upon this, adding a diversity component to the ranking.
- **ChromaDB Query Results:** In the `search` function, the results from ChromaDB (which include IDs, distances, and URIs) are transformed into a list of dictionaries, each containing the ID, distance, and a constructed `image_url`. This makes the data suitable for the `SearchResult` Pydantic model and, consequently, for sending as a JSON response to the frontend.