Topic: Multimedia Information Retrieval

Ngày 4 tháng 11 năm 2024

Môn học: Xử lý Ảnh số và Thị giác Máy tính
Giảng viên hướng dẫn: Võ Thanh Hùng
Sinh viên thực hiện: Nguyễn Trang Sỹ Lâm - 2152715
Nguyễn Hoàng Khôi Nguyên - 2152809

1 Giới thiệu

Báo cáo này trình bày việc phát triển một hệ thống truy xuất đa phương tiện cho tìm kiếm sản phẩm trên Amazon, nhằm nâng cao khả năng tìm kiếm của người dùng bằng cách cho phép sử dụng đồng thời cả văn bản và hình ảnh. Dự án giải quyết những hạn chế của phương pháp tìm kiếm truyền thống, thường chỉ dựa vào văn bản hoặc hình ảnh, bằng cách tích hợp cả hai loại dữ liệu để mang lại trải nghiệm người dùng chính xác và trực quan hơn.

Hệ thống sử dụng mô hình BGE visualized để xử lý và truy xuất các sản phẩm liên quan dựa trên cả embedding văn bản và hình ảnh. Sử dụng tập dữ liệu All_Beauty từ Amazon Reviews 2023, mỗi sản phẩm được biểu diễn bằng các embedding được tạo thông qua quá trình xử lý bằng Python. Các embedding này được lưu trữ trong Milvus, một cơ sở dữ liệu vector hiệu suất cao được lưu trữ trên Zilliz, cho phép khả năng tìm kiếm hiệu quả và mở rộng.

Phần backend, được phát triển bằng Flask, xử lý các truy vấn của người dùng và truy xuất các sản phẩm liên quan bằng cách truy vấn Milvus. Phần frontend, được xây dựng bằng Next.js, cung cấp giao diện thân thiện và đáp ứng, nơi người dùng có thể nhập truy vấn văn bản, tải lên hình ảnh hoặc kết hợp cả hai để tinh chỉnh tìm kiếm của họ.

2 Nền tảng và Công trình Liên quan

Chương này cung cấp tổng quan về các khái niệm và công nghệ cơ bản được sử dụng trong việc phát triển hệ thống truy xuất đa phương tiện, bao gồm truy xuất đa phương tiện, mô hình BAAI General Embedding (BGE) và công nghệ cơ sở dữ liệu vector Milvus.

2.1 Truy xuất Đa phương tiện trong Thương mại Điện tử

Truy xuất đa phương tiện là phương pháp cho phép truy vấn đồng thời bằng văn bản và hình ảnh, mang lại trải nghiệm tìm kiếm tinh tế hơn. Trong thương mại điện tử, các hệ thống tìm kiếm truyền thống thường xử lý độc lập truy vấn văn bản hoặc hình ảnh, dẫn đến độ chính xác tìm kiếm hạn chế. Truy xuất đa phương tiện giải quyết vấn đề này bằng cách kết hợp thông tin từ cả văn bản và hình ảnh, tạo ra một biểu diễn toàn diện phù hợp hơn với ý định của người dùng. Phương pháp này đặc biệt hữu ích trong các lĩnh vực thời trang, làm đẹp và phong cách sống, nơi mà các thuộc tính hình ảnh quan trọng không kém mô tả bằng văn bản.

2.2 Mô hình BAAI General Embedding (BGE)

BAAI General Embedding (BGE) là một mô hình đa năng được phát triển bởi Viện Trí tuệ Nhân tạo Bắc Kinh (BAAI), tạo ra các biểu diễn nhúng dày đặc và chất lượng cao cho dữ liệu văn bản và hình ảnh, cho phép ứng dụng hiệu quả trong các tác vụ tìm kiếm và truy xuất đa phương thức. Mô hình BGE Visualized, có sẵn trên Hugging Face, mở rộng từ BGE bằng cách cung cấp công cụ trực quan hóa, giúp tăng cường khả năng giải thích, cho phép người dùng khám phá các biểu diễn nhúng một cách không gian và trực quan. Với khả năng hỗ trợ truy vấn đồng thời văn bản và hình ảnh, BGE Visualized đặc biệt hữu ích cho các ứng dụng yêu cầu tìm kiếm sâu rộng trên các loại dữ liệu đa dạng, như tìm kiếm sản phẩm hoặc hệ thống đề xuất nội dung.

2.3 Cơ sở Dữ liệu Vector: Milvus

Milvus là một cơ sở dữ liệu vector mã nguồn mở được tối ưu hóa để lưu trữ và truy xuất các embedding có chiều cao như những embedding được tạo bởi mô hình BGE. Khác với các cơ sở dữ liệu truyền thống, vốn bị giới hạn trong dữ liệu có cấu trúc, Milvus được thiết kế đặc biệt cho dữ liệu phi cấu

trúc, như embedding văn bản và hình ảnh. Bằng cách sử dụng các phương pháp lập chỉ mục hiệu quả như HNSW (Hierarchical Navigable Small World), Milvus hỗ trợ truy xuất nhanh dựa trên các số liệu tương đồng, cho phép các ứng dụng tìm kiếm thời gian thực.

Trong dự án này, Milvus đóng vai trò là cơ sở dữ liệu chính, lưu trữ các embedding được tạo bởi mô hình BGE. Cơ sở dữ liệu được lưu trữ trên Zilliz Cloud, cung cấp khả năng mở rộng và quản lý tài nguyên hiệu quả để xử lý khối lượng dữ liệu lớn. Việc tích hợp Milvus với mô hình BGE cho phép truy xuất nhanh các sản phẩm liên quan dựa trên truy vấn đa phương thức của người dùng, đảm bảo rằng kết quả tìm kiếm không chỉ chính xác mà còn phản hồi nhanh với các đầu vào của người dùng trong thời gian thực.

3 Nguồn Tập dữ liệu

Tập dữ liệu được lấy từ bộ sưu tập Amazon Reviews 2023, do McAuley Lab biên soạn. Đây là một tập dữ liệu lớn, bao gồm hơn 571 triệu đánh giá và 48 triệu sản phẩm trên 33 danh mục, cung cấp một nguồn tài nguyên phong phú cho nghiên cứu và phát triển trong các hệ thống gợi ý và phân tích sản phẩm.

3.1 Tổng quan về Tập dữ liệu

Chương này cung cấp cái nhìn tổng quan về tập dữ liệu được sử dụng trong việc phát triển hệ thống truy xuất đa phương tiện, tập trung vào tập con All_Beauty của bộ dữ liệu Amazon Reviews 2023.

3.2 Tập con All_Beauty

Trong dự án này, tập con All_Beauty được chọn, bao gồm:

- Số lượng Người dùng: Khoảng 632.000 người dùng duy nhất.
- Số lượng Sản phẩm: Khoảng 112.600 sản phẩm làm đẹp khác nhau.
- Số lượng Đánh giá: Khoảng 701.500 đánh giá.
- Số lượng Token trong Đánh giá: Khoảng 31,6 triệu token trong các đánh giá của người dùng.
- Số lượng Token trong Metadata: Khoảng 74,1 triệu token trong metadata của sản phẩm.

Tập con này cung cấp một tập dữ liệu đủ lớn để phát triển và thử nghiệm hệ thống truy xuất đa phương tiện.

3.3 Cấu trúc Dữ liệu

Tập dữ liệu bao gồm hai thành phần chính:

Đánh giá của Người dùng, mỗi đánh giá bao gồm:

- Reviewer ID: Định danh duy nhất cho người đánh giá.
- ASIN (Amazon Standard Identification Number): Định danh duy nhất cho sản phẩm.
- Review Text: Nội dung của đánh giá.
- Rating: Điểm số do người đánh giá cung cấp.
- Timestamp: Ngày và giờ khi đánh giá được đăng.

Metadata của Sản phẩm: Metadata của mỗi sản phẩm bao gồm:

- ASIN: Định danh duy nhất của sản phẩm.
- Title: Tên của sản phẩm.
- Description: Thông tin chi tiết về sản phẩm.
- Price: Giá của sản phẩm.
- Brand: Nhà sản xuất hoặc tên thương hiệu.
- Image URLs: Liên kết đến hình ảnh của sản phẩm.
- Category: Thông tin về danh mục sản phẩm.

3.4 Khả năng Truy cập Dữ liệu

Tập dữ liệu có sẵn công khai trên nền tảng Hugging Face, cung cấp khả năng truy cập dễ dàng cho các mục đích nghiên cứu và phát triển. Nó được phân phối theo giấy phép cho phép sử dụng trong học thuật và phi thương mại, đảm bảo tuân thủ các chính sách sử dụng dữ liệu.

4 Quy trình Xử lý Dữ liệu

Chương này mô tả chi tiết quy trình xử lý dữ liệu, nêu rõ từng giai đoạn và các tệp mã liên quan. Quy trình bao gồm nhiều bước, mỗi bước cần thiết để chuyển đổi dữ liệu thô thành định dạng phù hợp để đưa vào cơ sở dữ liệu vector Milvus. Dưới đây là giải thích từng bước trong quy trình chuẩn bị và chèn dữ liệu.

4.1 Liên kết Sản phẩm với Đánh giá

Bước đầu tiên là liên kết mỗi sản phẩm với các đánh giá liên quan để hợp nhất thông tin. Tệp *link_ data.py* thực hiện việc này bằng cách:

- Truy xuất dữ liệu sản phẩm và đánh giá từ tập dữ liệu thô.
- Liên kết mỗi sản phẩm với các đánh giá tương ứng thông qua ASIN (Amazon Standard Identification Number).
- Tạo ra một tập dữ liệu hợp nhất trong đó mỗi mục sản phẩm chứa metadata và danh sách các đánh giá liên quan.

```
def link_data(metadata_file, reviews_file, output_file):
    # Load data from both files
    metadata = load_jsonl(metadata_file)
    reviews = load_jsonl(reviews_file)

    # Create a lookup dictionary for metadata using
    ↪  'parent_asin'
    metadata_lookup = {item["parent_asin"]: item for item in
    ↪  metadata}

    # Create a defaultdict to store reviews grouped by
    ↪  'parent_asin'
    grouped_reviews = defaultdict(list)
    for review in reviews:
        parent_asin = review.get("parent_asin")
        grouped_reviews[parent_asin].append(review)

    # Link the reviews with their corresponding metadata
    linked_data = []
    for parent_asin, product_metadata in
    ↪  metadata_lookup.items():
        linked_entry = {
```

```
            "product_metadata": product_metadata,
            "reviews": grouped_reviews.get(parent_asin, [])
        }
        linked_data.append(linked_entry)

    # Save the linked data to a new JSONL file
    save_jsonl(linked_data, output_file)
    print(f"Linked data saved to {output_file}")
```

4.2 Tái cấu trúc Dữ liệu cho Tương thích Schema

Sau khi liên kết đánh giá, dữ liệu cần được định dạng lại để phù hợp với schema đã được định nghĩa trong Milvus. Tệp *reconstruct_ data.py* thực hiện các tác vụ sau:

- Trích xuất các trường chính từ dữ liệu đã hợp nhất, bao gồm tiêu đề sản phẩm, mô tả, giá, thương hiệu, URL hình ảnh và các đánh giá liên quan.
- Cấu trúc dữ liệu để phù hợp với schema của Milvus, tổ chức các trường theo yêu cầu của cơ sở dữ liệu.
- Lưu dữ liệu đã tái cấu trúc ở định dạng sẵn sàng cho quá trình xử lý tiếp theo, đảm bảo tính tương thích với cơ sở dữ liệu vector.

```
def reconstruct_data(linked_data):
    """Reconstruct the linked data into the desired format."""
    transformed_data = {
        "collectionName": "AmazonProduct",
        "data": []
    }

    for entry in linked_data:
        product_metadata = entry.get("product_metadata", {})
        reviews = entry.get("reviews", [])

        # Transform and truncate data to fit the constraints
        transformed_entry = {
            "embed_image": None,  # Placeholder for 768-D float
            ↪ vector
            "main_category": safe_truncate(product_metadata.
            get("main_category"), 250),  # VARCHAR(250)
```

```
            "title":
            ↪  safe_truncate(product_metadata.get("title"),
            ↪  1000),  # VARCHAR(1000)
            "average_rating":
            ↪  float(product_metadata.get("average_rating",
            ↪  0)),  # FLOAT
            "rating_number":
            ↪  int(product_metadata.get("rating_number", 0)),
            ↪  # INT32
            "features": safe_list_truncate(product_metadata.
            get("features", []), 150),  #
            ↪  Array<VARCHAR(2000)>[150]
            "description": safe_list_truncate(product_metadata.
            get("description", []), 150),  #
            ↪  Array<VARCHAR(5000)>[150]
            "images_thumb": safe_list_truncate(
                [img.get("thumb") for img in
                ↪  product_metadata.get("images", [])], 150
            ),  # Array<VARCHAR(100)>[150]
            "images_large": safe_list_truncate(
                [img.get("large") for img in
                ↪  product_metadata.get("images", [])], 150
            ),  # Array<VARCHAR(100)>[150]
            "store":
            ↪  safe_truncate(product_metadata.get("store"),
            ↪  500),  # VARCHAR(500)
            "parent_asin": safe_truncate(product_metadata.
            get("parent_asin"), 15),  # VARCHAR(15)
            "reviews_title":
            ↪  safe_list_truncate([safe_truncate(review.
            get("title"), 100) for review in reviews], 100),  #
            ↪  Array<VARCHAR(100)>[100]
            "reviews_text":
            ↪  safe_list_truncate([safe_truncate(review.
            get("text"), 5000) for review in reviews], 100),  #
            ↪  Array<VARCHAR(5000)>[100]
            "reviews_rating":
            ↪  safe_list_truncate([float(review.get("rating",
            ↪  0)) for review in reviews], 100),  #
            ↪  Array<FLOAT>[100]
```

```
            "reviews_timestamp":
            ↪  safe_list_truncate([int(review.get("timestamp",
            ↪  0)) for review in reviews], 100),  #
            ↪  Array<INT64>[100]
            "reviews_verified_purchase":
            ↪  safe_list_truncate([review.
            get("verified_purchase", False) for review in
            ↪  reviews], 100),  # Array<BOOL>[100]
            "reviews_helpful_vote":
            ↪  safe_list_truncate([int(review.
            get("helpful_vote", 0)) for review in reviews],
            ↪  100)  # Array<INT32>[100]
        }

        # Add the transformed entry to the final dataset
        transformed_data["data"].append(transformed_entry)

    return transformed_data
```

4.3 Tải về Hình ảnh

Vì dữ liệu hình ảnh rất quan trọng cho truy xuất đa phương tiện, tệp *download_ image.py* tải xuống các hình ảnh liên quan đến mỗi sản phẩm:

- Truy cập URL trong dữ liệu đã tái cấu trúc và tải xuống các hình ảnh sản phẩm tương ứng.
- Lưu trữ hình ảnh tại địa phương hoặc trong thư mục được chỉ định để xử lý embedding và truy xuất sau này.
- Theo dõi trạng thái tải xuống để phát hiện và xử lý bất kỳ liên kết bị thiếu hoặc hỏng.

```
    def download_image_task(item, output_folder):
    """Download the first large image for a single product
    ↪  item."""
    parent_asin = item.get("parent_asin", "unknown")
    title = item.get("title", "unknown")

    # Get the first large image URL, if available
    images_large = item.get("images_large", [])
    if images_large:
        image_url = images_large[0]
```

```
        # Generate a meaningful filename
        title_snippet = sanitize_filename(title)  # Sanitize title for filename
        file_name = f"{output_folder}/{parent_asin}_{title_snippet}.jpg"

        # Download the image
        save_image(image_url, file_name)
    else:
        print(f"No large image found for product {parent_asin}")
```

4.4 Tạo Embedding cho Hình ảnh

Sau khi tải về hình ảnh, bước tiếp theo là tạo embedding cho mỗi hình ảnh bằng tệp *embed_image.py*, bao gồm:

- Tải từng hình ảnh sản phẩm và xử lý qua một mô hình được huấn luyện trước (như BGE) để tạo embedding.
- Thêm embedding hình ảnh vào mục sản phẩm tương ứng trong tập dữ liệu.
- Lưu tập dữ liệu với embedding hình ảnh dưới dạng JSON để xử lý tiếp theo.

```
class Encoder:
def __init__(self, model_name: str, model_path: str):
    self.model = Visualized_BGE(model_name_bge=model_name, model_weight=model_path)
    self.model.eval()

def encode_image(self, image_path: str) -> list[float]:
    with torch.no_grad():
        query_emb = self.model.encode(image=image_path)
    return query_emb.tolist()[0]

def embed_image_task(encoder, image_path):
"""Generate an embedding for a single image."""
try:
    embedding = encoder.encode_image(image_path)
    return os.path.basename(image_path), embedding
except Exception as e:
```

```
        print(f"Failed to generate embedding for {image_path}. Error: {e}")
        return os.path.basename(image_path), None
```

4.5 Lọc Dữ liệu

Trong quá trình xử lý dữ liệu, một số mục có thể gặp lỗi, chẳng hạn như thiếu hoặc lỗi embedding hình ảnh. Tệp *filter_ data.py* chịu trách nhiệm:

- Quét tập dữ liệu để phát hiện các mục không đầy đủ, đặc biệt là những mục thiếu embedding hình ảnh hợp lệ.
- Loại bỏ các mục bị lỗi trong khi giữ lại tập dữ liệu chất lượng cao.
- Bảo toàn 111.907 trong số 112.600 mục, đảm bảo rằng chỉ dữ liệu đáng tin cậy được chuyển vào cơ sở dữ liệu.

```
def clean_data(data_file, image_folder, output_file):
    """Filter out entries with errors in the embed_image field and delete corresponding images."""
    cleaned_data = {"data": []}

    # Stream the large JSON file with ijson
    with open(data_file, 'r', encoding='utf-8') as file:
        # Parse JSON items in "data" array one by one
        for item in ijson.items(file, "data.item"):
            embed_image = item.get("embed_image", None)
            parent_asin = item.get("parent_asin", "unknown")
            title = item.get("title", "unknown")

            # Check if embed_image is None or an empty list, consider it an error and delete the image
            if not embed_image:
                title_snippet = sanitize_filename(title)
                image_filename = f"{parent_asin}_{title_snippet}.jpg"
                image_path = os.path.join(image_folder, image_filename)

                # Delete the image file if it exists
                if os.path.exists(image_path):
                    os.remove(image_path)
                    print(f"Deleted image: {image_path}")
```

```
                else:
                    print(f"Image not found, skipping delete:
                    ↪ {image_path}")
            else:
                # If embed_image is valid, add entry to cleaned
                ↪ data
                cleaned_data["data"].append(item)

    # Save the cleaned data to a new JSON file using
    ↪ DecimalEncoder
    save_json(cleaned_data, output_file)
    print(f"Cleaned data saved to {output_file}")
```

4.6 Chia nhỏ Dữ liệu

Tập dữ liệu đã làm sạch, với dung lượng trên 3.5 GB, quá lớn để chèn vào cơ sở dữ liệu trong một lần. Tệp *split_ data.py*:

- Chia tập dữ liệu thành 404 phần nhỏ hơn, mỗi phần khoảng 8.8 MB.
- Lưu từng phần riêng biệt để dễ dàng chèn vào cơ sở dữ liệu.
- Chuẩn bị tập dữ liệu để xử lý hiệu quả trong bước chèn cuối cùng.

```
    def chunk_json(data_file, output_folder, chunk_size_mb=5):
    """
    Split JSON data into smaller chunks for easier HTTP
    ↪ transmission.

    :param data_file: Path to the JSON file to be chunked.
    :param output_folder: Directory to save the chunked JSON
    ↪ files.
    :param chunk_size_mb: Maximum size (in MB) for each chunk
    ↪ file.
    """
    # Convert chunk size to bytes
    estimated_chunk_size = chunk_size_mb * 1024 * 1024  #
    ↪ Convert MB to bytes
    chunked_data = []
    chunk_index = 1
    current_size = 0

    # Make sure the output directory exists
```

```
    os.makedirs(output_folder, exist_ok=True)

    # Stream JSON items from the data file
    with open(data_file, 'r', encoding='utf-8') as file:
        # Parse JSON items in the "data" array one by one
        for item in ijson.items(file, "data.item"):
            # Convert Decimal values to float within the item
            item = convert_decimal(item)

            # Estimate size of the current item by encoding it to JSON
            item_size = len(json.dumps(item).encode('utf-8'))

            # If adding the item would exceed the chunk size, save the current chunk
            if current_size + item_size > estimated_chunk_size:
                # Save current chunk
                chunk_filename = os.path.join(output_folder, f"cleaned_data_chunk_{chunk_index}.json")
                save_json({"data": chunked_data}, chunk_filename)
                print(f"Saved chunk {chunk_index} with approximately {current_size / (1024 * 1024):.2f} MB.")

                # Reset for the next chunk
                chunked_data = []
                current_size = 0
                chunk_index += 1

            # Add item to the current chunk
            chunked_data.append(item)
            current_size += item_size

    # Save any remaining data as the last chunk
    if chunked_data:
        chunk_filename = os.path.join(output_folder, f"cleaned_data_chunk_{chunk_index}.json")
        save_json({"data": chunked_data}, chunk_filename)
        print(f"Saved chunk {chunk_index} with approximately {current_size / (1024 * 1024):.2f} MB.")
```

4.7 Chèn Dữ liệu vào Milvus

Bước cuối cùng là chèn dữ liệu đã xử lý vào Milvus bằng tệp *insert_ data.py*, bao gồm:

- Tải từng phần dữ liệu tuần tự và chèn vào Milvus.
- Ánh xạ embedding văn bản và hình ảnh của mỗi mục vào schema vector của Milvus, thiết lập cấu trúc có thể truy vấn.
- Theo dõi quá trình chèn để đảm bảo tính toàn vẹn dữ liệu và việc lập chỉ mục thành công trong cơ sở dữ liệu

```
    headers = {
    'Authorization': f"Bearer {ZILLIZ_API_KEY}",
    'Accept': "application/json",
    'Content-Type': "application/json"
}

def send_chunk_to_db(chunk_data):
    """Send a JSON chunk to the database with retry and error
    ↪ handling."""
    payload = json.dumps({
        "collectionName": ZILLIZ_COLLECTION,
        "data": chunk_data["data"]
    })

    attempts = 0
    max_attempts = 3
    while attempts < max_attempts:
        try:
            # Set up connection for each attempt
            conn = http.client.HTTPSConnection(ZILLIZ_HOST,
            ↪ timeout=30)
            conn.request("POST", ZILLIZ_ENDPOINT, payload,
            ↪ headers)
            res = conn.getresponse()
            data = res.read()

            # Log response status
            print(f"Status: {res.status}, Response:
            ↪ {data.decode('utf-8')}")
            conn.close()
            break  # Exit retry loop on success
```

```
        except (http.client.HTTPException, ssl.SSLEOFError) as
        ↪  e:
            print(f"Error sending chunk: {e}")
            attempts += 1
            time.sleep(2)  # Wait before retrying
            conn.close()
            if attempts == max_attempts:
                print("Failed to send chunk after multiple
                ↪  attempts.")
        except Exception as e:
            print(f"Unexpected error: {e}")
            break  # Exit loop for unexpected errors
```

5 Mô hình BGE Visualized cho Truy xuất Đa phương tiện

Chương này trình bày chi tiết về Mô hình BGE Visualized, là nền tảng cho truy xuất đa phương tiện trong hệ thống. Khả năng của mô hình BGE trong việc xử lý và trực quan hóa cả đầu vào văn bản và hình ảnh mang lại lợi thế đáng kể, giúp tăng độ chính xác và tính phù hợp của kết quả tìm kiếm trong thương mại điện tử. Chương này bao gồm kiến trúc mô hình, xử lý truy vấn đa phương thức.

5.1 Kiến trúc Mô hình

Mô hình BGE Visualized được thiết kế để xử lý các đầu vào đa phương thức bằng cách xử lý cả dữ liệu văn bản và hình ảnh thông qua các lớp mã hóa riêng biệt nhưng song song. Kiến trúc bao gồm:

- Mã hóa Văn bản: Một mã hóa văn bản dựa trên transformer để chuyển đổi thông tin văn bản thành các embedding có độ chiều cao, nắm bắt các mối quan hệ ngữ nghĩa giữa các từ khóa.
- Mã hóa Hình ảnh: Một mạng nơ-ron tích chập (CNN) để xử lý hình ảnh, tạo ra các embedding phản ánh các mẫu hình ảnh, màu sắc và đặc điểm đặc trưng của từng sản phẩm.

Mô hình kết hợp hai loại embedding này vào cùng một không gian embedding, nơi mà các sự tương đồng giữa dữ liệu văn bản và hình ảnh có thể so sánh trực tiếp. Công thức toán học cho việc tạo embedding văn bản và hình ảnh như sau:

1. Gọi x_t là đầu vào văn bản và x_i là đầu vào hình ảnh.

2. Embedding văn bản e_t được tính bằng:

$$e_t = f_{text}(x_t)$$

trong đó f_{text} là mã hóa văn bản dựa trên transformer.

3. Embedding hình ảnh e_i được tính bằng:

$$e_i = f_{image}(x_i)$$

trong đó f_{image} là mã hóa hình ảnh dựa trên CNN.

4. Embedding kết hợp e_c trong không gian chung được mô tả bằng:

$$e_c = W_t e_t + W_i e_i$$

trong đó W_t và W_i là các ma trận trọng số cho embedding văn bản và hình ảnh, tương ứng.

5.2 Xử lý Truy vấn Đa phương thức

Mô hình BGE xử lý các truy vấn đa phương thức bằng cách embedding cả đầu vào văn bản và hình ảnh, cho phép truy xuất chéo phương thức. Điều này đặc biệt quan trọng trong các ứng dụng thương mại điện tử, khi người dùng thường tìm kiếm bằng một hoặc cả hai loại đầu vào để tìm sản phẩm phù hợp với mô tả và ngoại hình.

1. Tạo embedding: Với một truy vấn đa phương thức gồm văn bản x_t và hình ảnh x_i, mô hình tạo ra các embedding e_t và e_i như đã mô tả trong phần kiến trúc.

2. Tính toán độ tương đồng: Độ tương đồng giữa embedding truy vấn e_q và embedding sản phẩm e_p trong cơ sở dữ liệu được tính bằng cosine similarity:

$$Similarity(e_q, e_p) = \frac{e_q \cdot e_p}{||e_q||||e_p||}$$

3. Truy xuất Xếp hạng: Dựa trên các điểm tương đồng, các sản phẩm được xếp hạng, cho phép mô hình truy xuất các mặt hàng phù hợp nhất với truy vấn văn bản và hình ảnh của người dùng.

6 Thiết kế và Thiết lập Cơ sở Dữ liệu

Chương này mô tả thiết kế và thiết lập hệ thống cơ sở dữ liệu, tập trung vào việc sử dụng Milvus làm cơ sở dữ liệu vector được lưu trữ trên Zilliz Cloud. Chương chi tiết các kỹ thuật lập chỉ mục Hierarchical Navigable Small World được sử dụng để truy xuất hiệu quả.

6.1 Giới thiệu về Milvus

Milvus là một cơ sở dữ liệu vector mã nguồn mở được thiết kế để quản lý và truy xuất các vector chiều cao, rất quan trọng trong các ứng dụng liên quan đến dữ liệu phi cấu trúc như hình ảnh, văn bản và âm thanh. Milvus hỗ trợ tìm kiếm tương đồng trên các tập dữ liệu lớn, cho phép truy vấn hiệu quả các vector embedding được tạo bởi các mô hình học máy.

Các tính năng chính của Milvus:

- Hiệu suất cao: Tối ưu hóa cho độ trễ truy vấn ở mức mili giây, ngay cả với các tập dữ liệu vector ở quy mô hàng tỷ.
- Khả năng mở rộng: Hỗ trợ mở rộng ngang để xử lý khối lượng dữ liệu và tải truy vấn ngày càng tăng.
- Tính linh hoạt: Cung cấp các phương pháp lập chỉ mục khác nhau để cân bằng giữa tốc độ tìm kiếm và độ chính xác.
- Tích hợp: Tích hợp liền mạch với các công cụ và khung AI phổ biến, hỗ trợ phát triển các ứng dụng AI.

6.2 Lưu trữ trên Zilliz

Zilliz Cloud là dịch vụ cơ sở dữ liệu vector được quản lý hoàn toàn dựa trên Milvus, cung cấp khả năng mở rộng đám mây và hiệu suất cao.

Lợi ích của việc lưu trữ Milvus trên Zilliz:

- Dịch vụ được quản lý: Loại bỏ các phức tạp trong việc triển khai và bảo trì, cho phép các nhà phát triển tập trung vào phát triển ứng dụng.
- Mở rộng linh hoạt: Tự động mở rộng tài nguyên dựa trên nhu cầu tải công việc, đảm bảo hiệu suất tối ưu.
- Bảo mật: Đảm bảo các biện pháp bảo mật cấp doanh nghiệp để bảo vệ tính toàn vẹn và quyền riêng tư của dữ liệu.

6.3 Hierarchical Navigable Small World

Hierarchical Navigable Small World (HNSW) là một thuật toán tìm kiếm lân cận xấp xỉ dựa trên đồ thị, được tối ưu hóa cho việc tìm kiếm nhanh trong không gian vector có chiều cao. HNSW tạo ra một cấu trúc đồ thị phân cấp mà trong đó các nút kết nối với các nút khác dựa trên độ tương đồng của chúng. Với HNSW, các vector có vị trí gần nhau trong không gian được kết nối chặt chẽ, giúp truy xuất các mục tương tự với hiệu suất cao.

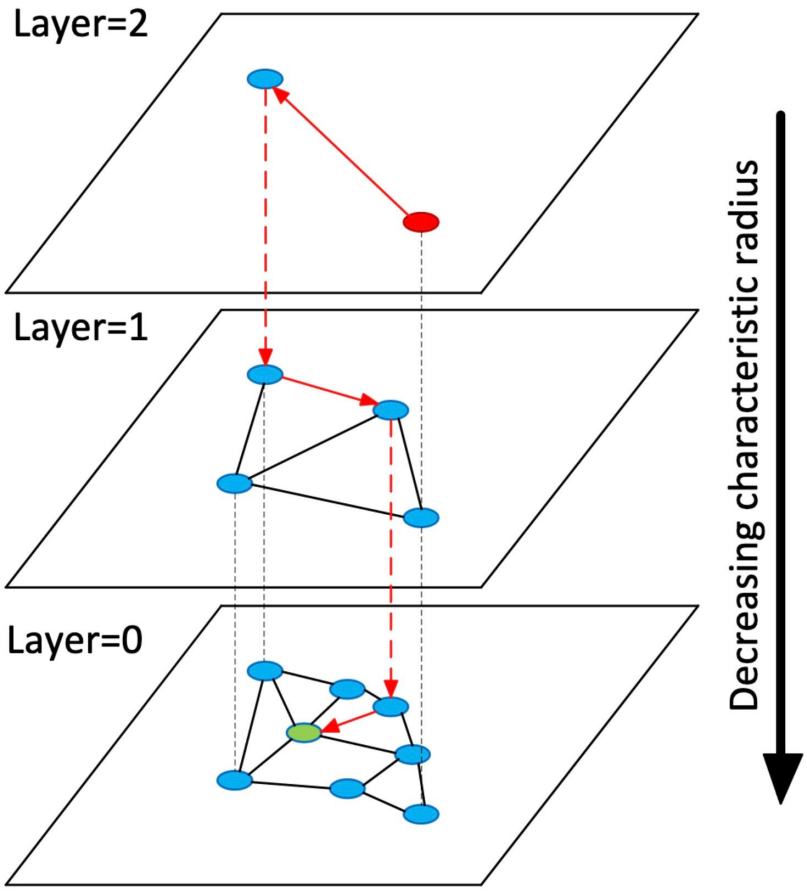
HNSW cải thiện tốc độ truy xuất so với các phương pháp truyền thống bằng cách giảm số lượng phép tính cần thiết để tìm các vector gần nhau, phù hợp cho việc xử lý các truy vấn đa phương tiện với dữ liệu lớn.

HNSW tạo ra một đồ thị lân cận theo cấp bậc với các đặc điểm sau:

- Phân Cấp Cấp Bậc: Các vector được tổ chức thành nhiều tầng (layers), với các nút ở tầng cao có ít kết nối hơn và được sắp xếp để tối ưu hóa tốc độ truy xuất. Các tầng thấp hơn có nhiều kết nối hơn, tập trung vào độ chính xác của tìm kiếm.

- Tìm kiếm và Duyệt Đồ Thị: Để tìm các vector gần nhau, thuật toán bắt đầu từ tầng cao nhất của đồ thị, di chuyển xuống các tầng thấp hơn, và duyệt qua các nút gần nhất với truy vấn để tìm các kết quả chính xác nhất.

- Chèn và Cập nhật Nút: Khi thêm một vector mới, HNSW duyệt qua đồ thị từ tầng cao xuống tầng thấp và tạo kết nối với các nút tương tự, duy trì cấu trúc phân cấp và tối ưu hóa truy xuất.



Hình 1: Minh họa HNSW.

7 Phát triển API Backend

Chương này mô tả API backend được phát triển để xử lý các truy vấn tìm kiếm đa phương tiện, tập trung vào việc triển khai bằng Flask.

7.1 API Framework

Flask được chọn làm framework phát triển API backend nhờ thiết kế nhẹ, linh hoạt và khả năng dễ dàng tích hợp với các dịch vụ bên ngoài như Milvus. Flask cho phép phát triển và triển khai nhanh chóng, phù hợp để xử lý các yêu cầu truy vấn và xử lý cần thiết cho ứng dụng này. Thêm vào đó, flask_cors được sử dụng để quản lý các yêu cầu cross-origin (CORS), đảm bảo sự kết nối mượt mà giữa giao diện frontend và backend.

7.2 Endpoint của API

API bao gồm một endpoint chính:

- Endpoint: /search
- Phương thức: POST
- Các Tham số Bắt buộc:
  - image: Tệp hình ảnh được tải lên bởi người dùng để tìm kiếm dựa trên sự tương đồng hình ảnh.
  - text: Truy vấn văn bản mô tả sản phẩm mong muốn.
  - limit: Một số nguyên chỉ định số lượng kết quả tối đa để trả về.

```
    @app.route('/search', methods=['POST'])
def search():
    """API endpoint for searching with an image and text query."""
    if 'image' not in request.files or 'text' not in request.form:
        return jsonify({"error": "Image file and text query are required."}), 400

    image = request.files['image']
    query_text = request.form['text']

    # Validate and parse the limit parameter
    try:
        limit = int(request.form.get('limit', 10))
        if limit < 1 or limit > 30:
            return jsonify({"error": "Limit must be between 1 and 30."}), 400
    except ValueError:
        return jsonify({"error": "Invalid limit value."}), 400

    # Debug logging to verify limit handling
    app.logger.debug(f"Limit received: {limit}")

    # Save the image temporarily
    image_filename = secure_filename(image.filename)
    temp_image_path = os.path.join("temp", image_filename)
    os.makedirs("temp", exist_ok=True)
```

```
    image.save(temp_image_path)

    # Generate the embedding
    query_vec =
    ↪  encoder.encode_query(image_path=temp_image_path,
    ↪  text=query_text)

    # Clean up the temporary file
    os.remove(temp_image_path)

    # Send the embedding to the database and get results
    response = send_query_to_db(query_vec, limit)

    return jsonify(json.loads(response))
```

7.3 Luồng Xử lý Truy vấn

Luồng xử lý truy vấn trong backend bao gồm các bước chính sau:

1. Nhận Dữ liệu Đầu vào từ Người dùng: API nhận các tham số image, text, và limit thông qua một yêu cầu POST. Cả đầu vào image và text đều cần thiết để tạo ra một embedding truy vấn đa phương thức, và limit xác định số lượng kết quả cần truy xuất.

2. Mã hóa Truy vấn: Sử dụng lớp Encoder, mô hình BGE tạo các embedding dựa trên kết hợp hình ảnh và văn bản đầu vào. Quy trình mã hóa bao gồm:

   - Tải và xử lý tệp hình ảnh và truy vấn văn bản.
   - Chạy cả hai đầu vào qua mô hình để tạo ra một embedding vector có chiều cao.

3. Gửi Truy vấn đến Milvus: Hàm send_query_to_db chịu trách nhiệm gửi embedding được tạo đến Milvus trên Zilliz Cloud. Payload gửi đi bao gồm:

   - collectionName: Chỉ định bộ sưu tập trong Milvus nơi lưu trữ các embedding sản phẩm.
   - data: Chứa embedding truy vấn dưới dạng danh sách số thực.
   - limit: Đặt số lượng kết quả tối đa được trả về, theo tham số limit do người dùng cung cấp.

4. Nhận Kết quả: Milvus xử lý truy vấn, tìm kiếm các mục gần nhất dựa trên sự tương đồng với embedding kết hợp từ hình ảnh và văn bản.

```
    def send_query_to_db(query_vec, limit):
    """Send the embedding query to the database with a
    ↪  specified limit."""
    conn = http.client.HTTPSConnection(ZILLIZ_HOST)
    payload = json.dumps({
        "collectionName": "Product",
        "data": [query_vec],
        "limit": limit,
        "outputFields": ["*"]
    })
    headers = {
        'Authorization': f"Bearer {ZILLIZ_API_KEY}",
        'Accept': "application/json",
        'Content-Type': "application/json"
    }
    conn.request("POST", ZILLIZ_SEARCH_ENDPOINT, payload,
    ↪  headers)
    res = conn.getresponse()
    data = res.read()
    conn.close()
    return data.decode("utf-8")
```

7.4 Định dạng Phản hồi

Dựa trên phản hồi từ cơ sở dữ liệu, API backend định dạng kết quả trước khi trả lại cho frontend dưới dạng cấu trúc JSON. Dưới đây là mô tả chi tiết về cấu trúc phản hồi của API.

```
interface SearchResponse {
    code: number;      // Mã trạng thái phản hồi, cho biết tình
    ↪  trạng xử lý yêu cầu
    cost: number;      // Chi phí tài nguyên của truy vấn
    data: Product[];   // Danh sách các sản phẩm khớp với truy
    ↪  vấn
}

interface Product {
    average_rating: number;                // Điểm đánh giá
    ↪  trung bình của sản phẩm
```

```
    description: Description;                // Mô tả chi tiết
    ↪ sản phẩm
    distance: number;                        // Khoảng cách
    ↪ vector, biểu thị mức độ tương đồng
    embed_image: number[];                   // Embedding hình
    ↪ ảnh của sản phẩm
    features: Features | null;               // Các tính năng
    ↪ đặc biệt (nếu có)
    id: number;                              // ID duy nhất của
    ↪ sản phẩm
    images_large: ImageUrls;                 // URL hình ảnh
    ↪ kích thước lớn của sản phẩm
    images_thumb: ImageUrls;                 // URL hình ảnh thu
    ↪ nhỏ của sản phẩm
    key: number;                             // Khóa chính của
    ↪ sản phẩm
    main_category: string;                   // Danh mục chính
    ↪ của sản phẩm
    parent_asin: string;                     // ASIN của sản
    ↪ phẩm
    rating_number: number;                   // Số lượng đánh
    ↪ giá của sản phẩm
    reviews_helpful_vote: ReviewData<IntData>;    // Số lượt
    ↪ bình chọn hữu ích
    reviews_rating: ReviewData<FloatData>;        // Điểm đánh
    ↪ giá từ các bình luận
    reviews_text: ReviewData<StringData>;         // Nội dung
    ↪ bình luận
    reviews_timestamp: ReviewData<LongData>;      // Dấu thời
    ↪ gian của các bình luận
    reviews_title: ReviewData<StringData>;        // Tiêu đề
    ↪ của các bình luận
    reviews_verified_purchase: ReviewData<BoolData>; // Xác
    ↪ minh mua hàng
    store: string;                            // Tên cửa hàng
    ↪ bán sản phẩm
    title: string;                            // Tên sản phẩm
}
```

8 Phát triển Giao diện Frontend

Chương này trình bày quá trình phát triển giao diện frontend của ứng dụng tìm kiếm đa phương tiện, bao gồm lựa chọn framework, các nguyên tắc thiết kế giao diện người dùng (UI), giao diện tìm kiếm hỗ trợ đầu vào văn bản và hình ảnh, và cách hiển thị kết quả để nâng cao trải nghiệm người dùng.

8.1 Framework Frontend: Next.js

Next.js được chọn làm framework frontend cho dự án này nhờ vào các tính năng mạnh mẽ và tính linh hoạt trong việc xây dựng các ứng dụng web phản hồi nhanh. Next.js là một framework dựa trên React hỗ trợ render phía server (SSR) và tạo trang tĩnh (SSG), giúp cải thiện tốc độ tải trang và trải nghiệm người dùng.

Lợi ích của Next.js cho Ứng dụng Này:

- Tối ưu hóa Hiệu suất: SSR và SSG giúp ứng dụng render trang nhanh hơn, cải thiện hiệu suất và trải nghiệm người dùng, đặc biệt đối với nội dung đa phương tiện.
- Quản lý Routing Dễ dàng: Next.js hỗ trợ routing dựa trên cấu trúc thư mục, giúp quản lý các URL và phân cấp trang một cách hiệu quả mà không cần cấu hình phức tạp.
- Tích hợp API: Với sự hỗ trợ tích hợp sẵn cho các route API, Next.js giúp việc kết nối với API backend trở nên đơn giản, tối ưu hóa quá trình lấy và quản lý dữ liệu cho kết quả tìm kiếm đa phương tiện.
- Trải nghiệm Phát triển Tốt: Các tính năng như hot reloading, báo lỗi tức thời và dễ dàng quản lý các thành phần với React làm cho Next.js trở thành lựa chọn lý tưởng cho phát triển frontend.

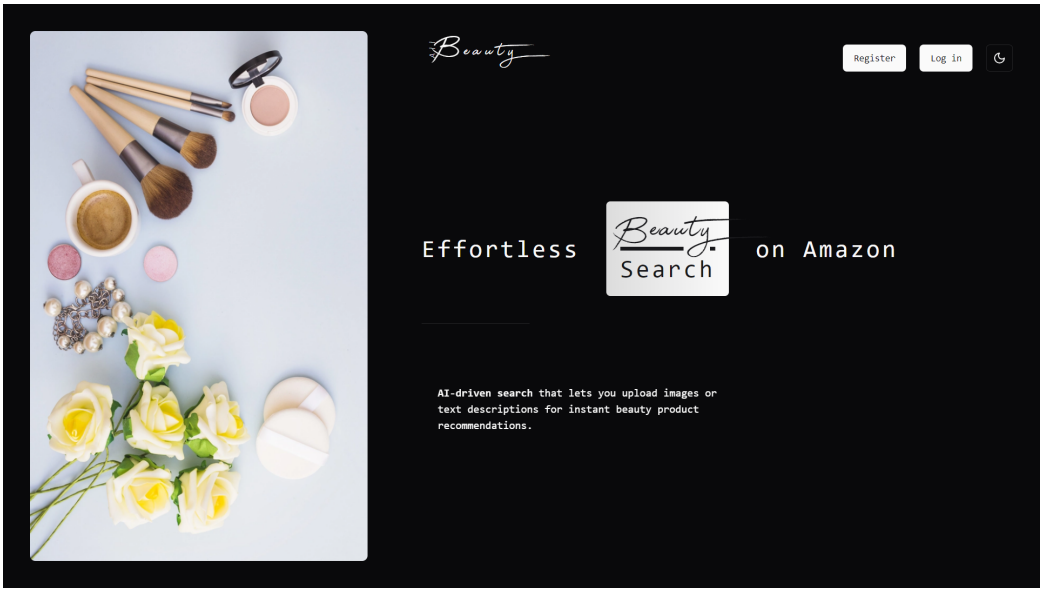
8.2 Thiết kế Giao diện Người dùng (UI)

Giao diện người dùng (UI) được thiết kế nhằm tạo ra một trải nghiệm trực quan và thu hút. Các nguyên tắc thiết kế bao gồm tính đơn giản, nhất quán và dễ sử dụng, đảm bảo rằng người dùng có thể dễ dàng điều hướng và tương tác với ứng dụng.

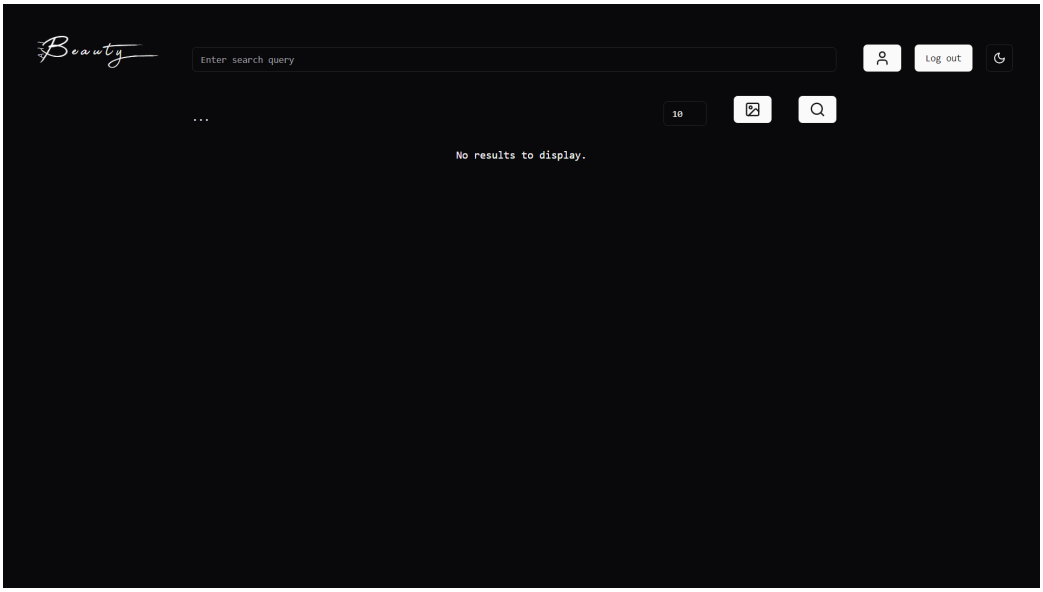
Nguyên tắc Thiết kế UI Chính:

- Bố cục Tối giản: Bố cục được giữ gọn gàng, tập trung vào tính năng tìm kiếm chính mà không làm người dùng rối mắt với quá nhiều lựa chọn.
- Thứ tự Thị giác Rõ ràng: Sử dụng tiêu đề rõ ràng, nhãn ngắn gọn và sắp xếp các phần một cách hợp lý giúp hướng dẫn người dùng đến các chức năng tìm kiếm và hiển thị kết quả một cách tự nhiên.
- Tính Năng Tiếp Cận: Tuân thủ các tiêu chuẩn về tiếp cận (ví dụ: tỷ lệ tương phản, văn bản thay thế cho hình ảnh) giúp ứng dụng thân thiện với nhiều người dùng, bao gồm cả những người có khuyết tật.

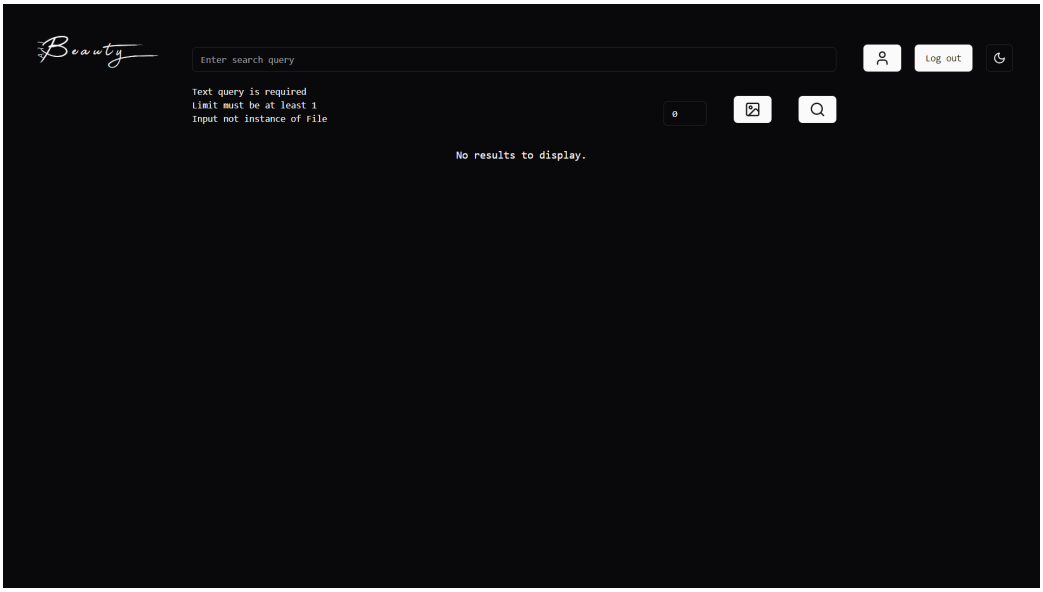
8.3 Minh họa kết quả



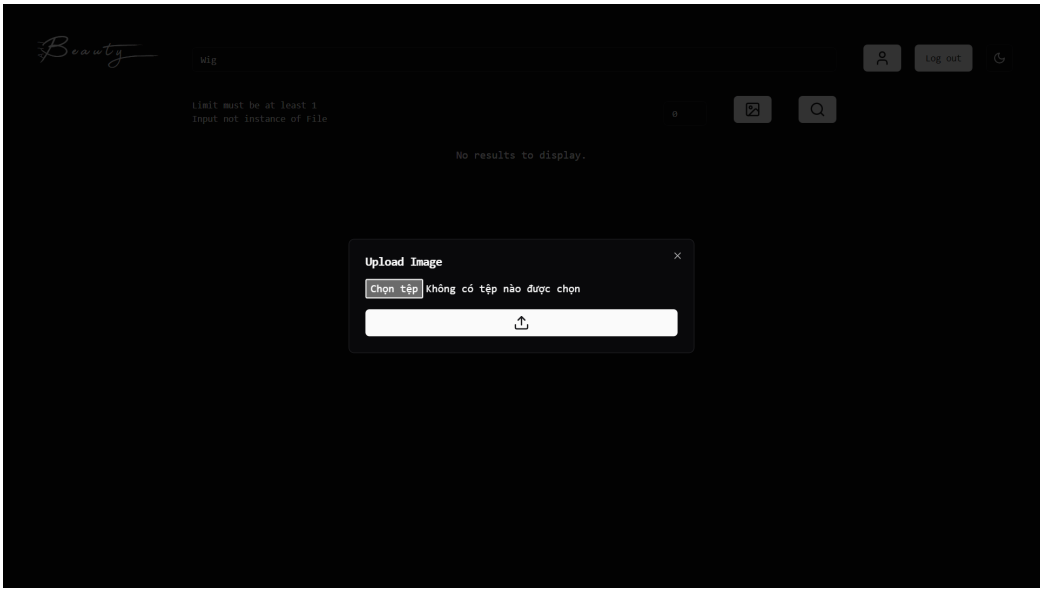
Hình 2: Trang chủ.



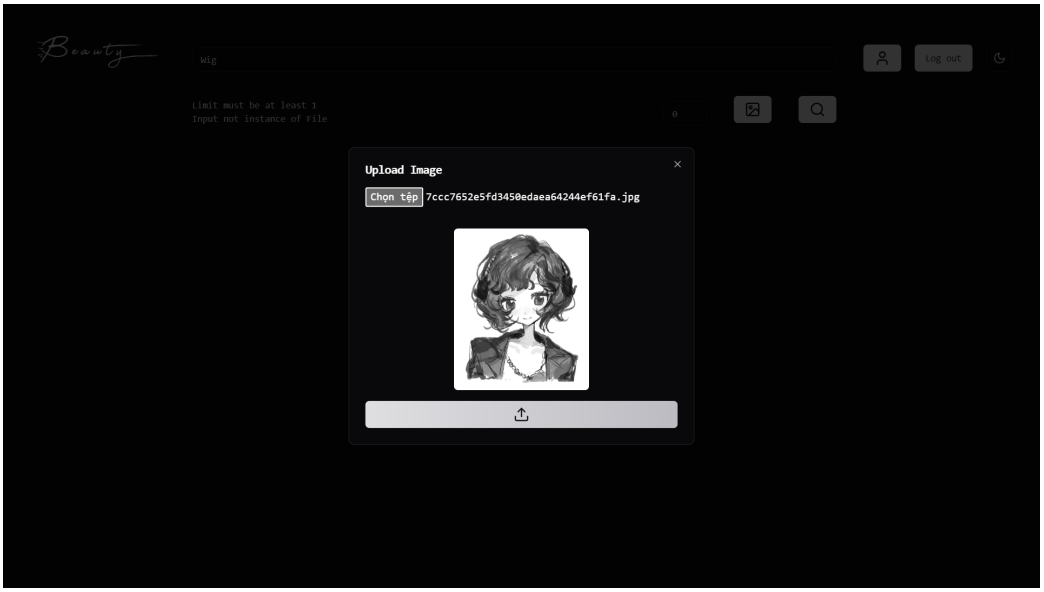
Hình 3: Trang tìm kiếm.



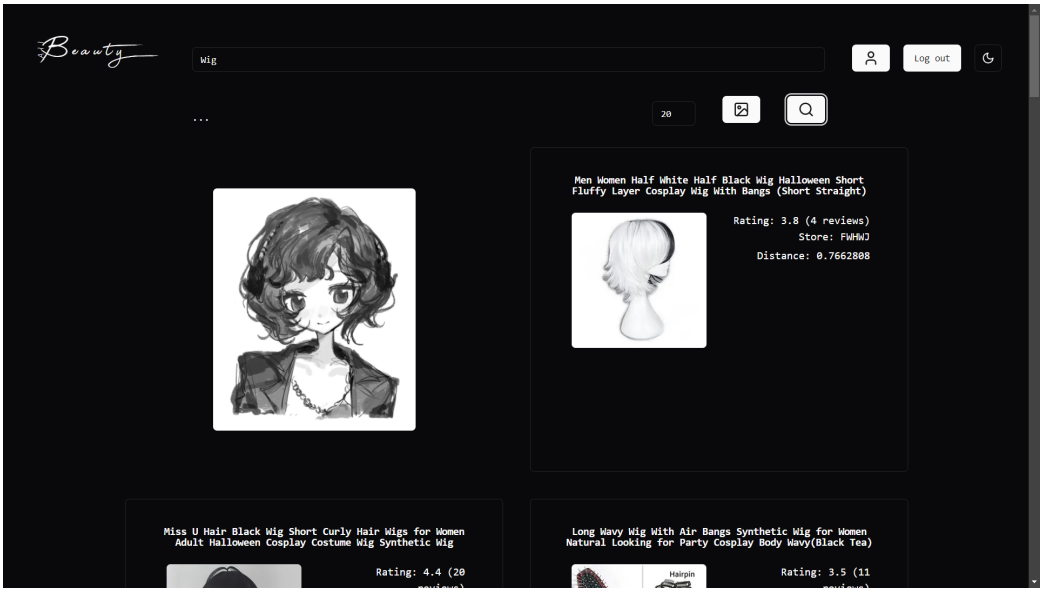
Hình 4: Yêu cầu về tham số truy vấn.



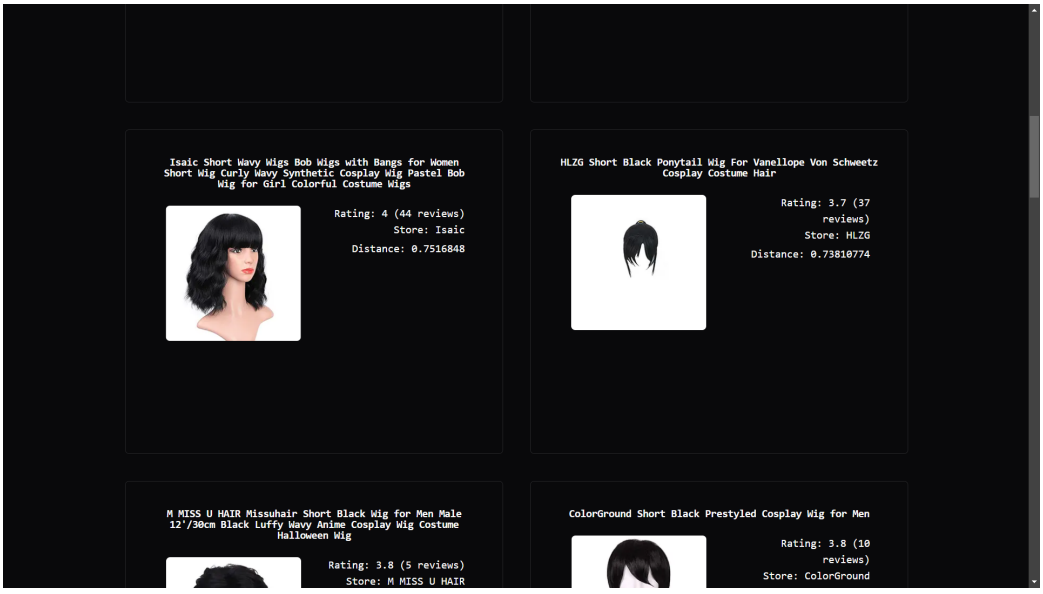
Hình 5: Tải hình ảnh lên.



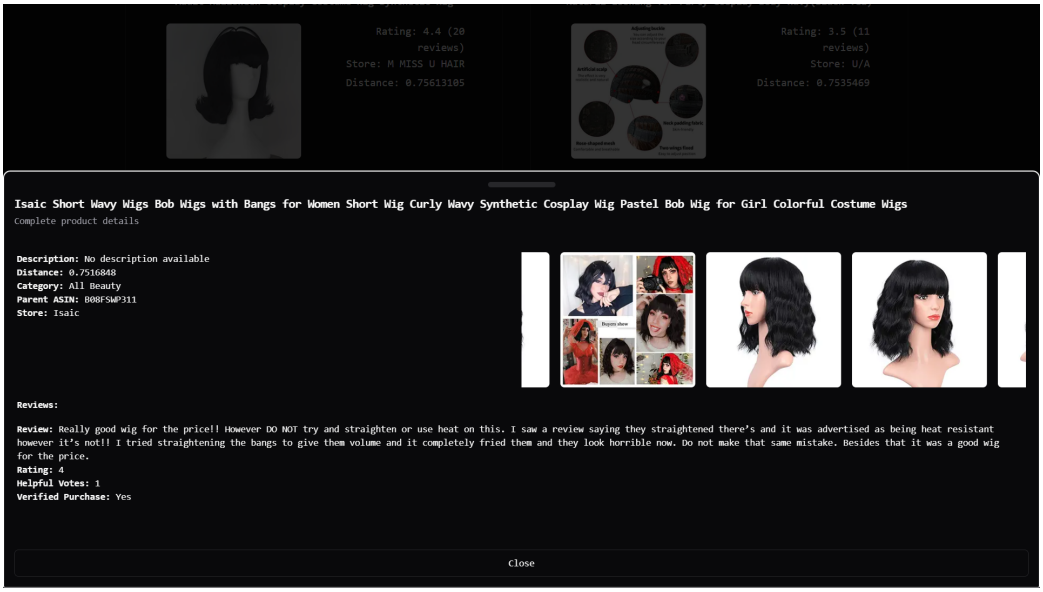
Hình 6: Xem trước hình ảnh đã lên.



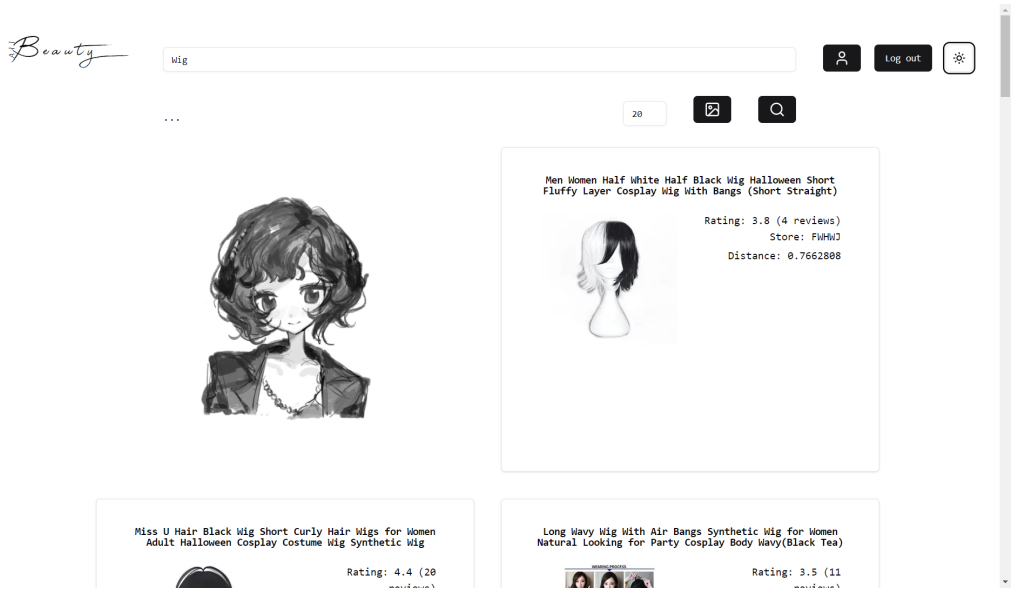
Hình 7: Kết quả tìm kiếm.



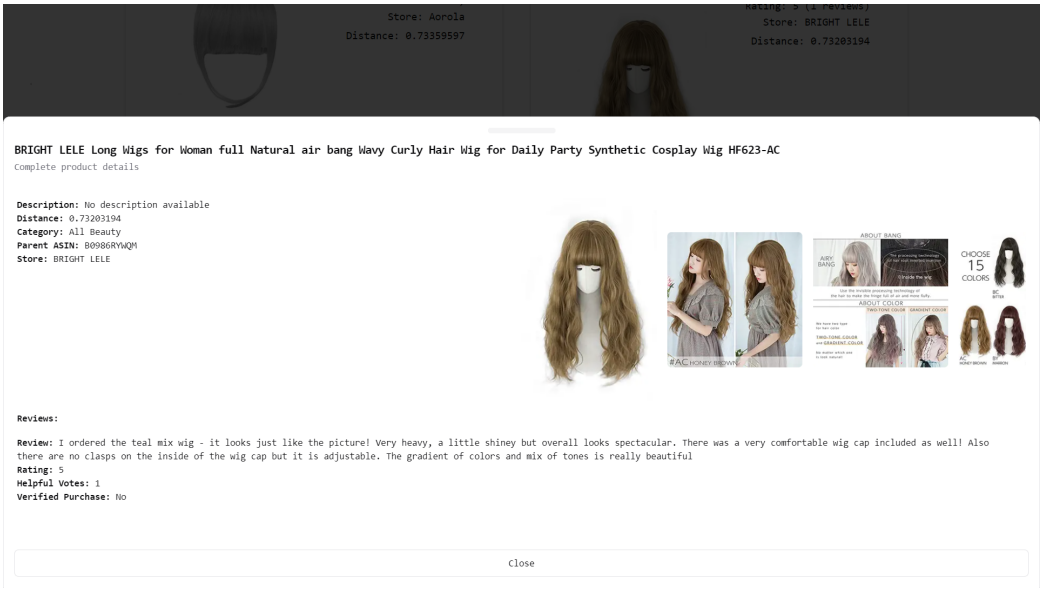
Hình 8: Kết quả tìm kiếm (tiếp theo).



Hình 9: Chi tiết sản phẩm.



Hình 10: Kết quả tìm kiếm ở chế độ sáng.



Hình 11: Chi tiết sản phẩm ở chế độ sáng.

9 Github của dự án

Mã nguồn đầy đủ có thể truy cập tại GitHub: Multimodal-RAG.

10 Tài liệu tham khảo

Tài liệu

[1] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu (2024). *BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation*. arXiv preprint arXiv:2402.03216.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). *Attention Is All You Need*. arXiv preprint arXiv:1706.03762.

[3] Keiron O'Shea and Ryan Nash (2015). *An Introduction to Convolutional Neural Networks*. arXiv preprint arXiv:1511.08458.

[4] Beijing Academy of Artificial Intelligence (BAAI). *BAAI/bge-visualized*. Hugging Face. Available at: `https://huggingface.co/BAAI/bge-visualized`.

[5] McAuley Lab (2023). *Amazon Reviews 2023 Dataset*. Hugging Face. Available at: `https://huggingface.co/datasets/McAuley-Lab/Amazon-Reviews-2023`.

[6] Zilliz (2024). *Understanding Hierarchical Navigable Small Worlds (HNSW)*. Available at: `https://zilliz.com/learn/hierarchical-navigable-small-worlds-HNSW`.