# **Technical Report: Visual Search and Recommendation System**

## **1. System Overview**

This document describes a visual search and recommendation system designed to identify similar images and related text based on a user-provided image URL and a text query. The system combines visual and textual feature extraction techniques to provide a rich and multi-faceted search experience. It consists of:

* **Python gRPC Service**: Contains all the machine learning and recommendation logic.
* **Java gRPC Client**: Interfaces with the Python service to send requests and display results.

The system receives a request containing an image URL and a text query, and it returns a set of similar images as well as similar text IDs related to those images.

## **2. Architecture**

The system is built using a client-server architecture with gRPC as the communication protocol.

### **Python gRPC Service**

#### **Role**

Responsible for:

* Loading data from a CSV file, including image URLs and text.
* Extracting image embeddings using a pre-trained ResNet50 model.
* Generating text embeddings using a Sentence Transformer model.
* Calculating cosine similarities between embeddings.
* Implementing the recommendation logic.
* Exposing the gRPC API for the Java client to make requests.

#### **Libraries**

Key libraries used:

* grpcio, grpcio-tools, protobuf: For defining and handling gRPC communication.
* pandas: For reading and managing CSV files.
* torch, torchvision: For image processing and the pre-trained ResNet50 model.
* Pillow (PIL): For image loading and transformations.
* numpy: For numerical operations, especially for processing embeddings.
* scikit-learn: For calculating cosine similarity.
* requests: To fetch images from URLs.
* sentence-transformers: For generating text embeddings.
* tqdm: For progress bars during processing.

#### **Data Loading and Processing**

* The service loads image URLs and text from a CSV file.
* Extracts image and text embeddings, which are cached to minimize redundant computations.

#### **gRPC Definition**

Defined in visual\_search.proto:

* **Service**: VisualSearchService with a Search method.
* **Messages**:
  + VisualSearchRequest: Contains image\_url and text fields.
  + VisualSearchResponse: Contains repeated fields for similar\_image\_urls and similar\_text\_ids.

### **Java gRPC Client**

#### **Role**

Responsible for:

* Sending image and text query requests to the Python service.
* Receiving and processing responses containing similar images and text IDs.

#### **Libraries**

Key libraries used:

* io.grpc:grpc-netty-shaded, io.grpc:grpc-protobuf, io.grpc:grpc-stub: For implementing the gRPC client.

#### **Communication**

* Uses a blocking stub to make synchronous calls to the Python server.

## **3. Key Components**

### **Image Embedding Extraction**

* Uses a pre-trained ResNet50 model to generate image embeddings.
* Images are resized to 224x224 pixels, converted to tensors, and normalized.
* The embeddings (output before the classification layer) are stored in a dictionary and cached in embeddings\_cache.pkl.

### **Text Embedding Generation**

* Uses the Sentence Transformer model (all-MiniLM-L6-v2) for text embeddings.
* Pre-processes text by concatenating all text columns.
* Stores embeddings in a dictionary and caches them in text\_embeddings\_cache.pkl.

### **Similarity Calculation**

* Computes cosine similarity between query embeddings and stored embeddings using cosine\_similarity from scikit-learn.

### **Recommendation Logic**

Combines image and text similarities:

1. **Image Similarity**: Identifies visually similar images using cosine similarity.
2. **Text Similarity**: Finds related text IDs based on the visual similarity results.
3. **Output**: Returns a list of indices corresponding to the dataset.

### **gRPC Communication**

* **Proto Definitions**: The visual\_search.proto defines the service and message structure.
* **Stubs**: Python generates stubs using grpcio-tools; Java uses protoc with the gRPC plugin.
* **Data Exchange**: Uses protobuf for serialization and deserialization.

## **4. Implementation Details**

### **Programming Languages**

* **Python**: Backend service logic.
* **Java**: Client-side application.

### **Libraries and Dependencies**

* **Python**:
  + gRPC-related: grpcio, grpcio-tools, protobuf.
  + ML-related: torch, torchvision, sentence-transformers.
  + Utility: pandas, numpy, Pillow, tqdm, scikit-learn.
* **Java**:
  + gRPC-related: io.grpc:grpc-netty-shaded, io.grpc:grpc-protobuf, io.grpc:grpc-stub.
  + Annotation API: javax.annotation:javax.annotation-api.

### **Data Format**

* CSV files for storing data and metadata.
* Requests: Strings for image URLs and text queries.
* Responses: Encoded as gRPC protobuf messages.

## **5. Performance Optimization**

### **Caching**

* Uses pickle to cache embeddings, reducing redundant computations.

### **Multithreading**

* Parallelizes embedding extraction using ThreadPoolExecutor.

### **Asynchronous Programming**

* Implements the gRPC server using asyncio for concurrent request handling.

### **Monitoring**

* Tracks CPU usage, response time, memory usage, and network throughput.

## **6. Error Handling and Data Validation**

### **Python Service**

* Implements try...except blocks for exceptions like requests.RequestException.
* Validates input types and sizes for critical functions.
* Includes logging for parameters and return values for debugging.

### **Java Client**

* Catches StatusRuntimeException for gRPC-related errors.
* Includes robust error handling for production readiness.

### **gRPC**

* Uses built-in mechanisms to handle transmission errors gracefully.

## **7. Deployment**

### **Current Deployment**

* Both Python service and Java client are deployed on the same machine for development.

### **Potential Configurations**

* **Docker Containers**: Ensures consistency and isolation.
* **Separate Machines**: Improves scalability and resource management.
* **Cloud Infrastructure**: Enables scalability with platforms like AWS, GCP, or Azure.

### **Security**

* Use TLS/SSL for secure communication.
* Address potential vulnerabilities in third-party libraries.

## **8. Challenges and Solutions**

* **Dataset Errors**: Ensured correct file paths and runtime directories.
* **Java Dependency Issues**: Fixed Maven configuration and plugin paths.
* **Indexing Errors**: Added validation for list indices.
* **Cache Usage**: Made embedding extraction conditional on cache presence.
* **gRPC Errors**: Implemented validations to prevent NoneType serialization issues.

## **9. Future Improvements**

### **Functionality**

* Advanced filtering options for searches.
* Real-time data updates.

### **Performance**

* Replace pickle with a database for embeddings.
* Design for scalability with tools like Kubernetes.

### **Robustness**

* Implement better logging frameworks and monitoring systems.
* Enhance security for public APIs.

### **Testing**

* Add unit tests for Python and Java components.

## **10. How to Use the System**

This section provides a comprehensive guide on using the visual search and recommendation system.

### **1. Prerequisites**

#### **1.1 Python Setup**

* Ensure Python 3.7+ is installed.

Install all required libraries specified in the requirements.txt file using the following command:  
 pip install -r requirements.txt

#### **1.2 Java Setup**

* Ensure Java 11+ and Maven are installed.
* Configure your development environment for Java projects.

#### **1.3 gRPC Configuration**

* Ensure your environment is correctly set up for running gRPC applications.

#### **1.4 Dataset Setup**

* Place your dataset file (dataset.csv) in the src/main/python\_service/ directory.
* Ensure the dataset adheres to the required format with the following columns:
  + image\_links
  + category\_1
  + category\_2
  + category\_3
  + title

### **2. Running the Python gRPC Service**

#### **Steps:**

1. Open a terminal or command prompt.
2. Navigate to the src/main/python\_service/ directory.

Start the Python gRPC server by executing:  
 python visual\_search\_server.py

Confirm that the server is running by verifying the following output message:  
 Server started, listening on port 50051

1. Ensure the server is active and listening correctly before proceeding.

### **3. Running the Java gRPC Client**

#### **Steps:**

1. Open a terminal or command prompt.
2. Navigate to the root directory of your Java project (containing the pom.xml file).

Execute the Java client by running:  
 mvn compile exec:java -Dexec.mainClass="visual\_search.VisualSearchClient"

1. The client will establish a connection with the Python server and display results in the console.

### **4. Testing the System**

#### **Validation:**

1. Review the results printed in the console after the Java client has finished executing.
2. Verify that:
   * The similar images displayed match the requested image type.
   * The list of similar text IDs:
     + Is not empty.
     + Contains the expected number of elements.

This structured approach ensures that the system operates smoothly and provides accurate outputs.