METADATA-AUGMENTED IMAGE INDEXING AND SEARCH USING FAISS

Abstract

The exponential growth of digital media, particularly images, has exposed significant limitations in traditional database systems for similarity search tasks. Conventional databases rely heavily on text-based metadata or manual tagging to retrieve content, which becomes inefficient and impractical as datasets scale to millions or billions of items. These systems struggle with the high-dimensional nature of embeddings vector representations derived from media like images or text lacking the ability to where manual methods fail to keep pace with data volume and complexity.

The Faiss library, developed by Meta's FAIR team, overcomes these challenges by providing a robust framework for Approximate Nearest Neighbour Search (ANNS) tailored to high-dimensional embedding vectors. Unlike traditional databases, Faiss does not manage metadata or offer full database functionalities like concurrent writes; instead, it focuses on optimizing vector indexing and search through techniques like vector compression (e.g., Product Quantization) and non-exhaustive search (e.g., HNSW, IVF). This allows Faiss to trade off between speed, memory usage, and accuracy, delivering rapid similarity searches even on trillion-scale datasets. By separating embedding extraction from search operations and supporting flexible index types, Faiss empowers applications to handle large embedding collections efficiently, making it a cornerstone for modern vector databases like Milvus and Pinecone.

Our "Similar Image Search Tool" leverages Faiss's capabilities to enable intuitive and efficient image similarity search within a user-friendly Streamlit interface. The program integrates a pre-trained ResNet18 model to extract 512-dimensional feature vectors from images, reduces them to 128 dimensions using PCA, and combines them with metadata (e.g., image ratings) before indexing with Faiss's HNSW method. Users can upload a dataset of images, index them, and query with a single image to retrieve the top-k most similar matches, with k adjustable via the interface. By harnessing Faiss's fast, approximate search and our custom pre processing pipeline, the tool delivers accurate results in real-time, demonstrating a practical application of ANNS for tasks like digital archiving or visual content retrieval, bridging advanced algorithms with end-user accessibility.

Keywords: ResNet18, FAISS, PCA, Streamlit, image similarity search, feature extraction, metadata integration

Contents

1. Introduction

The exponential growth of digital imagery in personal and professional domains has intensified the demand for tools that can efficiently search and retrieve similar images based on visual content. Traditional approaches, such as keyword-based searches or manual tagging, fall short when handling large datasets due to their reliance on human effort and subjective interpretation. The "Similar Image Search Tool" introduces an AI-powered solution that automates this process, utilizing advanced computer vision and similarity search techniques to deliver fast, accurate results.

Built on a Streamlit framework, the tool employs a pre-trained ResNet18 model to extract visual features from images, reduces dimensionality with PCA, and uses FAISS's HNSW index for rapid similarity search. It also integrates metadata (e.g., image ratings) to refine results, offering a hybrid approach to image retrieval. Users can upload a dataset of images, index them, and query with a single image to find the top-k most similar matches, all within an intuitive web interface. This project bridges the gap between complex machine learning algorithms and practical usability, making image search accessible and efficient.

1.1 Existing System

Current image search systems often depend on text-based metadata or manual annotations, requiring users to tag images with descriptive keywords. Tools like Google Images or basic database searches exemplify this approach, but they struggle with scalability and accuracy when metadata is incomplete or inconsistent. Some advanced systems use content-based image retrieval (CBIR), but they often lack user-friendly interfaces or require significant computational resources, limiting their accessibility.

Drawbacks:

- Manual tagging is labor-intensive and prone to errors or subjectivity.
- Text-based searches fail when metadata is missing or irrelevant to visual content.
- Existing CBIR tools are often complex, slow, or inaccessible to non-experts.

1.2 Proposed System

The system harnesses the Faiss library, a powerful toolkit for Approximate Nearest Neighbor Search (ANNS), to index and query these embeddings efficiently, as detailed in its design principles by Douze et al. Faiss excels by focusing solely on vector search, employing optimized techniques like Hierarchical Navigable Small World (HNSW) indexing and vector compression to balance speed, memory usage, and accuracy, unlike full-fledged databases that handle broader tasks like transaction management. In our implementation, users upload a dataset of images (PNG, JPG, JPEG) via Streamlit's interface, which the system indexes using Faiss's HNSW method after feature extraction and metadata concatenation. A query image is then processed similarly, and the top-k similar images—where k is user-defined—are retrieved and displayed in a dynamic grid layout, with robust error handling for corrupted inputs. This approach ensures rapid, accurate results, making the tool suitable for applications like e-commerce, digital archiving, and content management, bridging advanced ANNS with practical usability.

1.3 Literature Survey

The Faiss library, detailed in a comprehensive paper by Johnson et al. (2019) and expanded upon by Douze et al., emerges as a pivotal tool for Approximate Nearest Neighbour Search (ANNS), offering a suite of indexing methods like HNSW and IVF to tackle these challenges. Faiss's design, as described, optimizes vector search by employing vector compression (e.g., PQ) and non-exhaustive search strategies (e.g., Hierarchical Navigable Small World graphs by Malkov and Yashunin, 2018), achieving high performance on trillion-scale datasets. The library's flexibility is evident in its support for various codecs—scalar quantizers, additive quantizers, and binary indexes allowing trade-offs between memory usage, search speed, and accuracy, as benchmarked on datasets like Deep1B (Babenko and Lempitsky, 2016). Studies like Guo et al. (2020) with SCANN and Subramanya et al. (2019) with DiskANN further refine these techniques, but Faiss stands out for its broad applicability and integration into vector databases like Milvus (Wang et al., 2021). Our project leverages Faiss's HNSW implementation, enhancing it with ResNet18 feature extraction and PCA, as inspired by these works, to ensure efficient and accurate image similarity search. Streamlit's role, per its documentation, facilitates rapid prototyping, though it demands careful state management—addressed in our code via session state—to delive

2. System Analysis

2.1 Functional Requirements

- **Image Upload:** Users can upload multiple images (PNG, JPG, JPEG) to create a dataset and a single query image for searching.
- **Indexing:** The system extracts features from uploaded images, reduces dimensionality, and builds a FAISS HNSW index.
- **Similarity Search:** Users specify k (number of results) and retrieve the top-k similar images based on a query image.
- **Metadata Integration:** Incorporates image metadata (e.g., Rating) into the search process.
- **Result Display:** Shows similar images with filenames and metadata in a grid layout.

2.2 Non-Functional Requirements

- **Performance:** Processes and indexes images quickly, with search results returned in under a few seconds.
- **Usability:** Offers an intuitive Streamlit interface accessible to non-technical users.
- **Reliability:** Handles corrupted or truncated images gracefully with error messages.
- **Scalability:** Supports datasets of varying sizes, limited only by memory and FAISS optimization.
- **Security:** Ensures uploaded files are processed locally without external storage risks.

2.3 Hardware Requirements

Processor: Intel Core i3 or AMD Ryzen 3 or above for efficient feature extraction.

Memory (RAM): Minimum 4GB, recommended 8GB for larger datasets.

Storage: At least 10GB free space for temporary image storage and indexing.

Internet Connection: Optional, required only for initial library downloads.

2.4 Software Requirements

Operating System: Windows, macOS, or Linux (compatible with Python).

Programming Language: Python 3.8+.

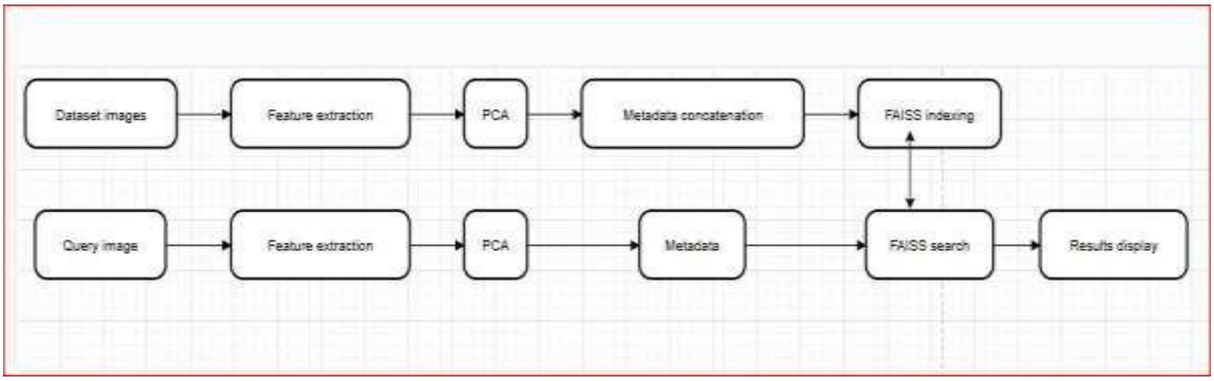
Libraries: Streamlit, PyTorch, torchvision, FAISS, scikit-learn, PIL, numpy.

Development Environment: Visual Studio Code or any Python IDE, Google coolab

3. System Design

3.1 System Architecture

The system processes images through a pipeline of feature extraction, indexing, and similarity search, integrated into a Streamlit frontend.



The given architecture diagram visually represents the workflow of a **similar image search system** using **FAISS indexing and feature extraction**. The process begins with dataset images under going feature extraction using a **pre-trained ResNet18 model**, followed by **Principal Component Analysis (PCA)** to reduce feature dimensions. Metadata from images is then concatenated with these reduced features to form a **combined feature vector**, which is subsequently indexed using **FAISS (Facebook AI Similarity Search)** for efficient nearest neighbour searches.

In the query phase, the user uploads a query image, which follows the same pipeline of **feature extraction, PCA transformation, and metadata extraction**. The processed query feature is then compared against the FAISS index to retrieve the **most similar images based on their feature similarity**. The final search results are displayed to the user, showing the retrieved images that closely match the query image. This structured pipeline ensures a **fast and scalable** image retrieval system suitable for various real-world applications.

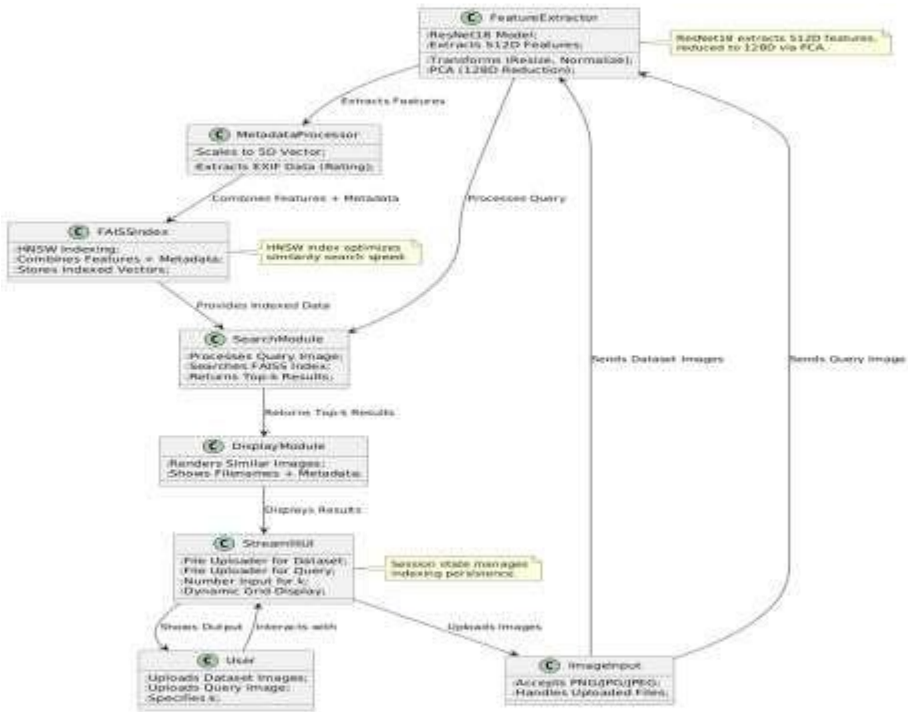
Key Components:

1. **Image Input:** Users upload dataset images and a query image via Streamlit's file uploader.
2. **Feature Extraction:** ResNet18 extracts 512D features, reduced to 128D with PCA.
3. **Metadata Processing:** Extracts and scales metadata (e.g., Rating) into a 5D vector.

4. **Indexing:** Combines features and metadata into a FAISS HNSW index for fast search.
5. **Search Module:** Queries the index with a processed query image, returning top-k results.
6. **Display:** Renders results in a dynamic grid with Streamlit's column layout.

4. UML Diagrams

4.1 Class diagram :-



1. **User Interaction with System**
   - The `User` uploads dataset images and a query image using the `ImageInput` class.
   - The `StreamlitUI` class provides a frontend interface for file uploads and result display.

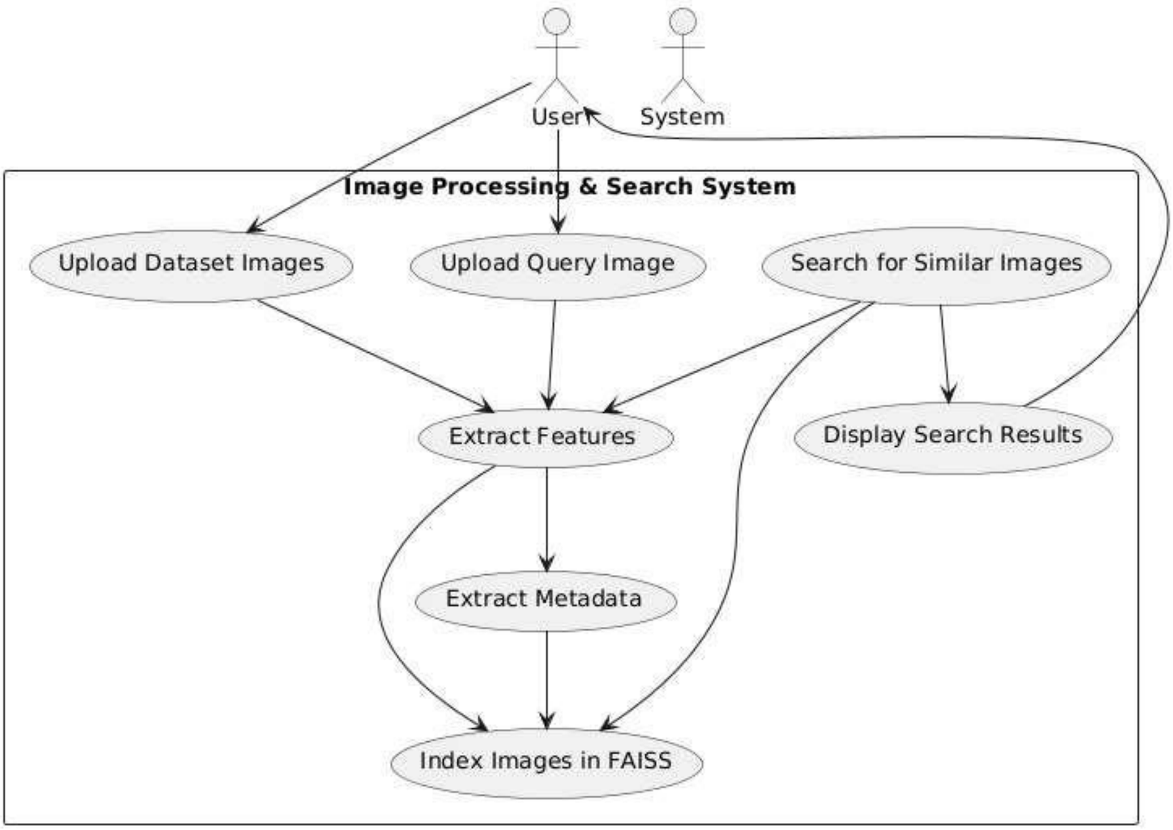
2. **Image Processing & Feature Extraction**

   - The `FeatureExtractor` class processes images using a **ResNet18 model** to extract **512D feature vectors**.
   - The extracted features are transformed (resized, normalized) and reduced to **128D using PCA**.
   - The output features are sent to both **indexing (for dataset images) and querying (for search images).**
3. **Metadata Processing**
   - The `Metadata Processor` extracts metadata such as **EXIF data (image rating)** and scales it to a **5D vector**.
   - The processed metadata is later **concatenated with the extracted image features**.
4. **Indexing & Search (FAISS Index & Search Module)**
   - The `FAISS Index` class receives combined **image features + metadata** and stores them using **HNSW indexing** for fast retrieval.
   - When a query image is uploaded, the `Search Module` extracts features, retrieves **top-k similar images** from the FAISS index, and returns results.
5. **Result Display & Output Handling**
   - The `Display Module` takes the retrieved images and **renders filenames and metadata**.
   - The `Streamlit UI` dynamically updates the results grid to show the most similar images.

Function & Object Interaction Summary:

- **`FeatureExtractor.extract_features(image)`** → Returns a 128D feature vector.
- **`MetadataProcessor.extract_metadata(image)`** → Returns a 5D metadata vector.
- **`FAISSIndex.add_to_index(feature_vector, metadata_vector)`** → Stores processed vectors.
- **`SearchModule.search(query_vector)`** → Retrieves **top-k most similar images**.

4.3 UseCase Diagram



- **Actors:**

`User`: Represents the person interacting with the system by uploading images and receiving search results.

- `System`: Represents the backend processing of images.

- **Use Cases and Interactions:**

- **(Upload Dataset Images) (UC1):** The user uploads dataset images to the system.
- **(Upload Query Image) (UC2):** The user uploads a query image for searching similar images.
- **(Extract Features) (UC3):** The system processes uploaded images using a feature extraction model (e.g., ResNet18).
- **(Extract Metadata) (UC4):** The system extracts additional metadata like EXIF data.
- **(Index Images in FAISS) (UC5):** The system indexes dataset images using FAISS with extracted features.

- **(Search for Similar Images) (UC6):** The system compares the query image features with indexed dataset images.
- **(Display Search Results) (UC7):** The system retrieves and shows the most similar images to the user.

- **Workflow:**

  - When the **user uploads images**, the system **extracts features** and **metadata**.
  - The **extracted data is indexed** using FAISS for fast similarity search.
  - When a **query image is uploaded**, the system **searches for top-k similar images**.
  - Finally, the **retrieved results are displayed** back to the user.

5. Implementation

5.1 Project Tree

text
CollapseWrapCopy

```
Similar_Image_Search/
├── app.py                # Main Streamlit application file
├── data/                 # Directory for temporary files
│   └── uploaded_images/ # Subdirectory for uploaded images
├── requirements.txt      # List of Python dependencies
└── README.md             # Project documentation
```

5.2 Structure of Program

- **app.py:** Main module handling UI, indexing, and search logic.
- **FeatureExtractor Class:** Encapsulates ResNet18, PCA, and feature processing.
- **Helper Functions:** Metadata extraction, scaling, and search utilities.

5.3 Coding: app.py Main Module & Test Cases

```
import streamlit as st
import os
import time
import faiss
import torch
import numpy as np
from torchvision import models, transforms
from sklearn.decomposition import PCA
from PIL import Image, ImageFile, ExifTags

# Allow truncated images to load
ImageFile.LOAD_TRUNCATED_IMAGES = True

# Feature Extractor Class
class FeatureExtractor:
    def __init__(self, pca_components=128):
        self.model = models.resnet18(pretrained=True)
        self.model = torch.nn.Sequential(*(list(self.model.children())[:-
1]))
```

```
        self.model.eval()
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
        ])
        self.pca = PCA(n_components=pca_components)
        self.pca_trained = False

    def extract(self, image):
        try:
            image = image.convert('RGB')
            image = self.transform(image).unsqueeze(0)
            with torch.no_grad():
                feature = self.model(image).squeeze().numpy()
            return feature / np.linalg.norm(feature)
        except Exception as e:
            st.error(f"Error processing image: {e}")
            return None

    def apply_pca(self, features):
        if not self.pca_trained:
            self.pca.fit(features)
            self.pca_trained = True
        return self.pca.transform(features)

# Metadata functions
def extract_metadata(image):
    exif_data = image.getexif()
    metadata = {}
    if exif_data:
        for tag, value in exif_data.items():
            tag_name = ExifTags.TAGS.get(tag, tag)
            if tag_name in ['Rating']:
                try:
                    metadata[tag_name] = float(value)
                except ValueError:
```

```
                pass
        return metadata

def scale_metadata(metadata, size=5):
    rating = metadata.get('Rating', 0.0)
    return np.full(size, rating / 5.0, dtype=np.float32)

# Index dataset function for uploaded files
@st.cache_resource
def index_dataset(uploaded_files):
    extractor = FeatureExtractor(pca_components=128)
    file_paths = []
    features = []
    metadata_vectors = []

    with st.spinner("Indexing uploaded dataset images..."):
        for uploaded_file in uploaded_files:
            if uploaded_file.name.endswith(('.png', '.jpg', '.jpeg')):
                image = Image.open(uploaded_file)
                feature = extractor.extract(image)
                metadata = extract_metadata(image)
                if feature is not None:
                    file_paths.append(uploaded_file.name)
                    metadata_vector = scale_metadata(metadata)
                    features.append(feature)
                    metadata_vectors.append(metadata_vector)

    if not features:
        st.error("No valid images found in uploaded files!")
        return None, None, None

    features = np.array(features, dtype='float32')
    metadata_vectors = np.array(metadata_vectors, dtype='float32')
    reduced_features = extractor.apply_pca(features)
    combined_vectors = np.hstack((reduced_features, metadata_vectors))
    d = combined_vectors.shape[1]

    hnsw_index = faiss.IndexHNSWFlat(d, 32)
```

```
    hnsw_index.add(combined_vectors)

    return hnsw_index, file_paths, extractor

# Search function (modified to accept k as a parameter)
def search_similar_images(query_image, index, file_paths, extractor,
stored_metadata, k):
    query_feature = extractor.extract(query_image)
    if query_feature is None:
        return [], {}

    query_feature = extractor.apply_pca(query_feature.reshape(1, -1))[0]
    metadata_vector = scale_metadata(stored_metadata)
    query_vector = np.hstack((query_feature, metadata_vector)).reshape(1, -
1)

    distances, indices = index.search(query_vector, k)
    results = [file_paths[idx] for idx in indices[0]]
    return results, stored_metadata

# Streamlit UI
def main():
    st.title("Similar Image Search")
    st.write("Upload multiple index images and a query image to find
similar images")

    # Upload multiple index files
    st.subheader("Upload Index Images (Dataset) press Ctrl+a to select all
the images")
    uploaded_index_files = st.file_uploader(
        "Choose multiple images for indexing...",
        type=['png', 'jpg', 'jpeg'],
        accept_multiple_files=True
    )

    # Initialize index when files are uploaded
    if uploaded_index_files:
```

```
        if 'index' not in st.session_state or
st.session_state.get('index_files') != [f.name for f in
uploaded_index_files]:
            st.session_state.index, st.session_state.file_paths,
st.session_state.extractor = index_dataset(uploaded_index_files)
            st.session_state.index_files = [f.name for f in
uploaded_index_files]
            if st.session_state.index is not None:
                st.success(f"Dataset indexed successfully with
{len(uploaded_index_files)} images!")
            else:
                st.error("Failed to index dataset. Please check your
uploaded files.")

    # Upload query image
    st.subheader("Upload Query Image")
    uploaded_query_file = st.file_uploader(
        "Choose a query image...",
        type=['png', 'jpg', 'jpeg'],
        accept_multiple_files=False
    )

    if uploaded_query_file is not None and 'index' in st.session_state and
st.session_state.index is not None:
        # Display uploaded query image
        query_image = Image.open(uploaded_query_file)
        st.image(query_image, caption='Uploaded Query Image',
use_container_width=True)

        # User input for number of similar images (top-k)
        k = st.number_input(
            "Number of similar images to retrieve",
            min_value=1,
            max_value=len(st.session_state.file_paths),  # Limit to dataset
size
            value=3,  # Default value
            step=1
        )
```

```
        # Use the first index image's metadata as stored metadata (for
simplicity)
        if uploaded_index_files:
            sample_metadata_image = Image.open(uploaded_index_files[0])
            stored_metadata = extract_metadata(sample_metadata_image)

        # Search button
        if st.button("Search Similar Images"):
            with st.spinner("Searching for similar images..."):
                similar_images, metadata_used = search_similar_images(
                    query_image,
                    st.session_state.index,
                    st.session_state.file_paths,
                    st.session_state.extractor,
                    stored_metadata,
                    k  # Pass user-defined k
                )

            # Display results
            st.subheader("Similar Images Found:")
            if similar_images:
                # Dynamically adjust columns based on k, max 3 per row
                num_cols = min(3, k)
                cols = st.columns(num_cols)
                for i, img_name in enumerate(similar_images):
                    matching_file = next(f for f in uploaded_index_files if
f.name == img_name)
                    img = Image.open(matching_file)
                    cols[i % num_cols].image(img, caption=img_name,
use_container_width=True)

                st.write("Metadata used for search:", metadata_used)
            else:
                st.warning("No similar images found.")

if __name__ == "__main__":
    main()
```

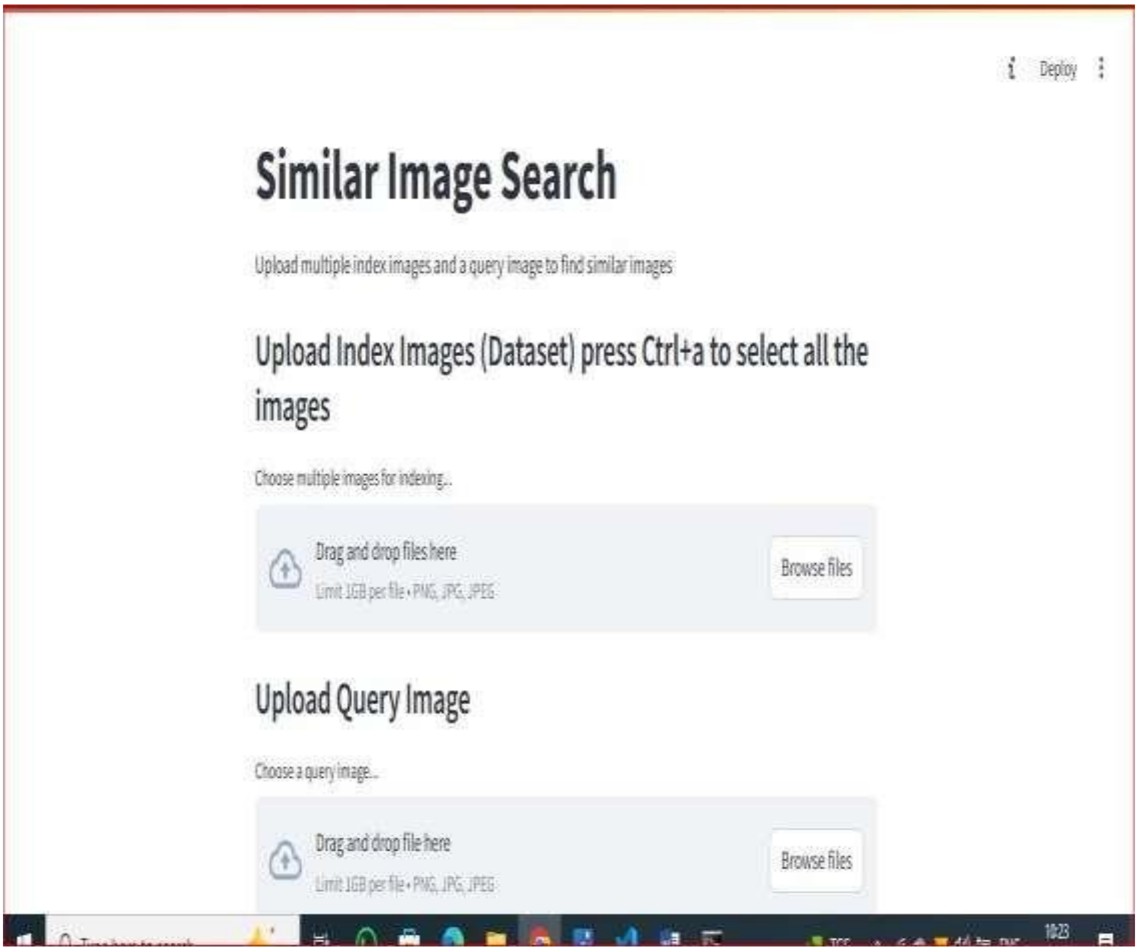
Test Cases:

1. **Test Case 1 (White Box):**
   - **Scenario:** Upload a corrupted image to the dataset.
   - **Expected Outcome:** System displays an error message ("Error processing image...") and skips the image.
2. **Test Case 2:**
   - **Scenario:** Search with k >dataset size.
   - **Expected Outcome:** Limits k to dataset size and returns all indexed images.

---

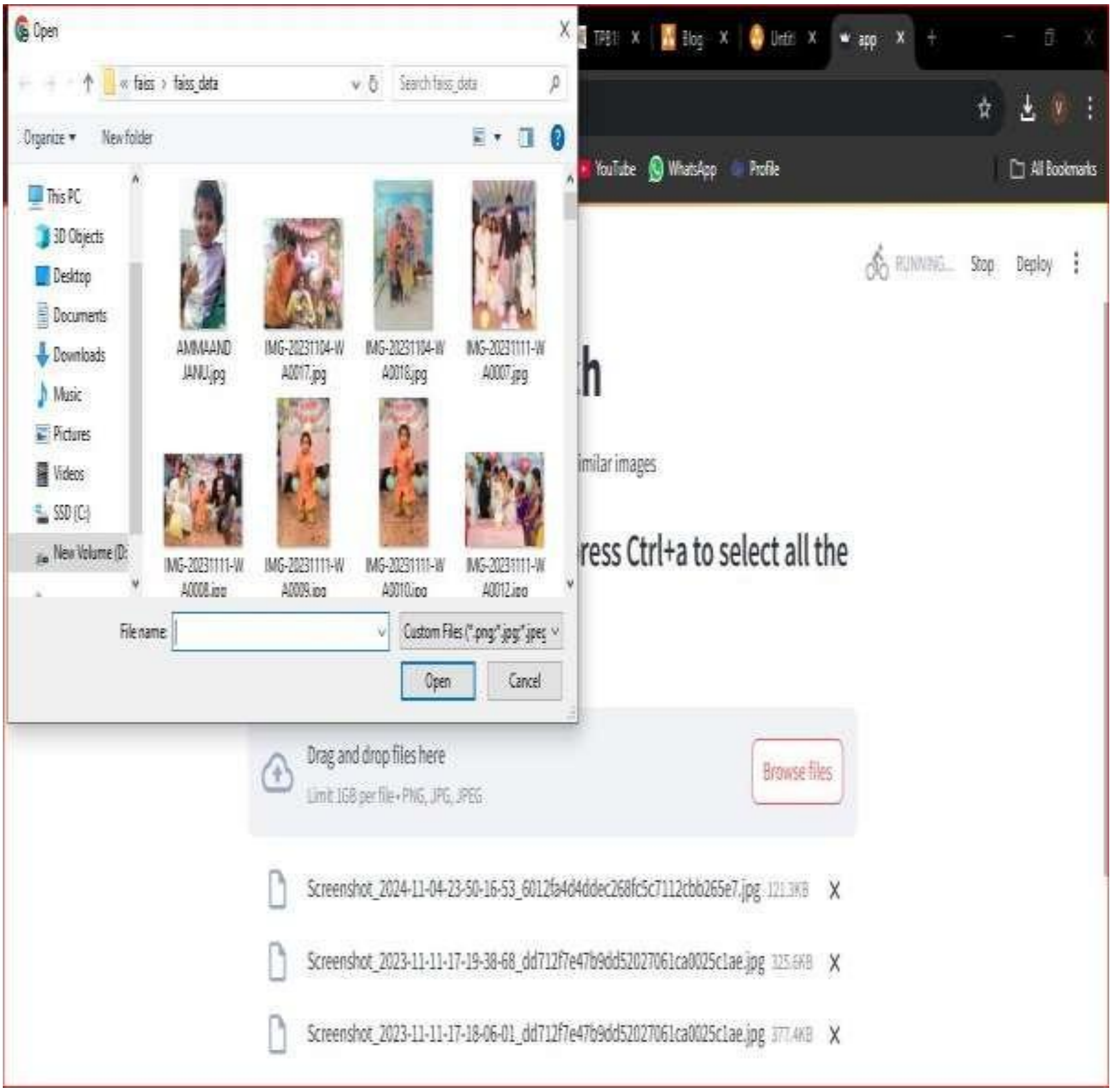
6. Output Screens

6.1 main output Screens

- 

.

6.2 Index Images Upload Screen: Ctrl+A to select all the images



6.3 Query Image Upload Screen:

6.4 Similar Images Results Screen: Grid of top-k similar images with filenames.

7. Conclusion

The "Similar Image Search Tool" successfully demonstrates the integration of computer vision and similarity search to address the challenges of image retrieval. By automating feature extraction with ResNet18, optimizing with PCA, and leveraging FAISS for fast search, it offers an efficient and scalable solution. The Streamlit interface enhances accessibility, making it valuable for applications like e-commerce, digital libraries, and personal photo management. Future enhancements could include multi-modal search (text + image) or real-time indexing, further expanding its utility. This project marks a significant step in applying AI to practical, user-centric problems.

References

1. He, K., et al. (2016). "Deep Residual Learning for Image Recognition." *IEEE CVPR*.
2. Johnson, J., et al. (2017). "FAISS: A Library for Efficient Similarity Search." *arXiv:1702.08734*.
3. Streamlit Inc. (n.d.). "Streamlit Documentation." Retrieved from https://streamlit.io/.

4 Pedregosa, F., et al. (2011). **FAISS (Facebook AI Similarity Search)**
FAISS GitHub Repository
FAISS Documentation

5 ◈ **HNSW (Hierarchical Navigable Small World Graphs)**
HNSW Paper by Malkov & Yashunin (2018)
HNSWLib GitHub Repository

6 "Scikit-learn: Machine Learning in Python." *JMLR*.

Github link:

https://github.com/Vishnu8087/faiss-image-search