Vietnam National University, Ho Chi Minh City
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

DATA STRUCTURE AND ALGORITHM (CO2004)

Assignment

"Reducing Large Image Datasets by Detecting Content-Duplicate Images Using Deep Learning and Feature Hashing."

Instructor(s):	Lê Thành Sách
Class:	TN01
Student:	Nguyễn Anh Quân - 2412900
Student:	Phạm Văn Hên - 2410968
Student:	Lê Bảo Tấn Phong - 2412635

Ho Chi Minh City, December 2025

Contents

1 Introduction

1.1 Background & Motivation

With the rapid growth of digital data, large-scale image collections have become increasingly common in various domains such as social media, cloud storage, e-commerce, and computer vision research. However, these datasets often contain duplicate or near-duplicate images, which can arise from multiple uploads of the same content, minor editing (cropping, resizing, or color adjustment), or recompression.

Such redundancy not only wastes storage space but also leads to bias and inefficiency in downstream tasks like image retrieval, classification, and training machine learning models. Therefore, detecting and removing duplicate images is a crucial preprocessing step to ensure data quality, reduce computational cost, and improve overall system performance.

1.2 Objectives

This project aims to design and implement a system capable of automatically identifying and removing duplicate or near-duplicate images within a large dataset. The specific goals are:

- To detect and eliminate redundant images based on their visual content rather than filenames or metadata.
- To utilize deep learning models (e.g., ResNet, EfficientNet) to extract high-level feature embeddings that represent image content.
- To compare and evaluate multiple similarity search methods, including custom hash-based techniques (SimHash, MinHash) and the FAISS library, in terms of accuracy, speed, and memory efficiency.

1.3 Scope & Limitations

The current codebase is implemented in **Python** with a **C++** SimHash backend (pybind11) for performance-critical hashing. All experiments in the repository use a COIL-100 subset of **400 images** (6 object labels, files named `obj#__angle.png`). Ground truth duplicate pairs are automatically generated by pairing images that share the same object label.

The system targets content-level duplicates and near-duplicates defined by shared object identity. It does **not** cover broader semantic similarity (different objects in the same category), heavy photo-editing, or video data. Because the dataset is small and balanced, reported metrics reflect prototype behavior rather than large-scale deployment; throughput and recall on web-scale crawls still need validation.

1.4 Main Contributions

This work provides:

- A complete deduplication pipeline wired to concrete code: CNN-based embedding extractors (EfficientNet-B0, ResNet50, ConvNeXt-Tiny) feeding FAISS search, SimHash LSH (multi-table C++ backend), or MinHash LSH.
- Auto-generation of ground-truth duplicate pairs from filenames, plus precision/recall evaluation scripts based on pairwise cluster consistency.

- Representative selection heuristics (resolution + sharpness) to retain the best image per cluster and utilities for exporting clustered results.
- Baseline measurements for accuracy/time/memory on the bundled COIL-100 subset to guide later scaling experiments.

2 Theoretical Background

2.1 Image Feature Representation

In image processing and computer vision, the ability to represent an image through meaningful numerical features is essential for tasks such as classification, retrieval, and duplication detection. Instead of working directly with raw pixel values, modern approaches employ deep learning models—particularly **Convolutional Neural Networks (CNNs)**—to extract high-level semantic features that capture the visual content of an image.

CNN-Based Models

Convolutional Neural Networks (CNNs) have revolutionized visual representation learning due to their capability to automatically learn hierarchical patterns from data. Unlike traditional hand-crafted descriptors such as SIFT or HOG, CNNs learn both low-level and high-level visual features directly from training images through multiple layers of convolution, pooling, and non-linear transformations.

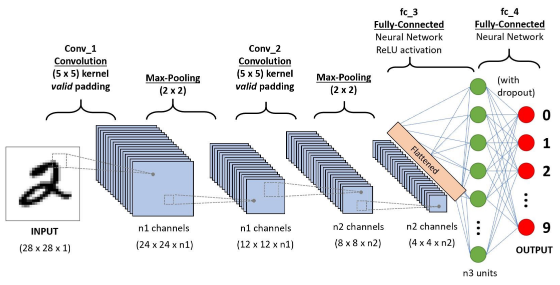


Figure 1: Convolutional Neural Networks structure

Popular pre-trained CNN architectures, such as **ResNet**[1], **EfficientNet**[2], and **VGG**, have been extensively used as feature extractors in various computer vision applications:

- **ResNet (Residual Network):** Introduces residual connections to enable training of very deep networks while avoiding vanishing gradients. ResNet's output before the final classification layer (often a 2048-dimensional vector) is widely used as a robust image embedding.
- **EfficientNet:** Scales depth, width, and resolution of the network in a balanced way, achieving high accuracy with fewer parameters and faster inference, making it suitable for large-scale datasets.
- **ConvNeXt (ConvNet for the 2020s):** A modern CNN architecture designed to compete with Vision Transformers (ViTs)[3]. ConvNeXt "modernizes" the standard ResNet architecture by adopting design choices from Transformers, such as larger kernel sizes

(7×7), depthwise separable convolutions, and Layer Normalization. This project supports **ConvNeXt-Tiny** (via `torchvision`), which offers strong accuracy and generalizes well to diverse image datasets while maintaining the efficiency typical of convolutional networks.

- **Vision Transformer (ViT-B/16):** Included as an optional extractor[4]; the classification head is replaced by an identity layer to expose patch-token embeddings for similarity search experiments.

These models are commonly initialized with weights pre-trained on large datasets such as **ImageNet**, which allows them to generalize well even when applied to new, unseen data domains.

From Image to Feature Vector

The process of feature extraction can be viewed as a transformation function:

$$f : I \to \mathbb{R}^d$$

where I is an input image and $f(I)$ is its corresponding feature vector (embedding) in a high-dimensional space $\mathbb{R}^d$.

This vector encodes semantic information about the image's content — images with similar visual structures (e.g., same objects, shapes, or scenes) will have feature vectors that are close to each other according to a similarity metric such as **cosine similarity** or **Euclidean distance**.

Typically, the extraction process involves:

1. Preprocessing the image (resizing, normalization).
2. Feeding it through a CNN and removing the final classification layer.
3. Taking the activations from the penultimate layer as the embedding vector.

These embeddings serve as compact and discriminative representations, allowing subsequent algorithms—such as hashing or FAISS similarity search—to efficiently detect duplicate or near-duplicate images based on content similarity rather than pixel-level comparison.

2.2 Feature Hashing Techniques

Feature hashing is a family of techniques that transform high-dimensional feature vectors into compact hash representations while approximately preserving similarity. In image deduplication, these methods allow rapid comparison between images by operating on lightweight hash codes instead of full feature vectors, significantly reducing both memory usage and computation time.

- **Hash Table and Bloom Filter (brief):** Hash tables can be used to quickly check whether a hash signature has appeared before, enabling constant-time duplicate detection. Bloom Filters extend this idea with a probabilistic bit-array structure that supports extremely memory-efficient membership tests. Although they may yield false positives, Bloom Filters guarantee no false negatives, making them suitable for large-scale pipelines. In this project, Bloom Filters are only mentioned as a supplementary concept and are not implemented.
- **SimHash (Cosine-preserving locality-sensitive hashing):** SimHash is a locality-sensitive hashing (LSH) technique specifically designed to preserve *cosine similarity*. It works by projecting high-dimensional vectors onto a set of random hyperplanes. Each hyperplane produces a single bit based on which side of the hyperplane the vector lies on,

resulting in a compact binary fingerprint. Two images with similar feature vectors will generate SimHash signatures with small Hamming distance[5]. In this project, the production path uses a pybind11 C++ backend with multiple hash tables and optional multi-probing controlled by a Hamming-distance threshold.

In the context of image deduplication:

- Feature vectors (e.g., from EfficientNet or ResNet) are converted into 64–256 bit signatures.
- The Hamming distance between signatures is used as an approximation of angular similarity.
- Lookup is extremely fast because Hamming distance is computed with bitwise operations.
- Effective for detecting near-duplicates where small geometric or photometric changes occur.

SimHash is particularly efficient for large-scale datasets because its indexing structures (e.g., multi-table LSH) allow sub-linear query time.

- **MinHash (Jaccard-preserving locality-sensitive hashing):** MinHash estimates the Jaccard similarity between two sets by randomly permuting feature indices and taking the minimum hashed value under each permutation. In practice, the permutation process is approximated using multiple independent hash functions, producing a MinHash signature composed of many "minimum" values[6].

  For image deduplication, feature vectors are often binarized or quantized into sets (e.g., selecting top-k activated dimensions). MinHash signatures for similar images share many identical components, enabling accurate similarity estimation without computing full Jaccard similarity.

  In this project, MinHash is used to cluster near-duplicate images based on:

  - Fast similarity computation using signature comparisons.
  - High recall for detecting highly similar or partially overlapping visual content.
  - Efficient index structures such as LSH Forest or MinHash LSH for large datasets.

  MinHash is especially effective when images share structural similarities but may differ slightly in pixels or transformations.

2.3 FAISS Library

The **FAISS (Facebook AI Similarity Search)** library is a highly optimized framework for efficient similarity search and clustering of dense vectors. It performs **Approximate Nearest Neighbor (ANN)** search, which allows fast retrieval of similar embeddings in large datasets[7].

Key aspects include:

- **Functionality:** Supports various indexing structures (e.g., Flat, IVFPQ, HNSW) to balance speed and accuracy in high-dimensional vector search. The default pipeline uses `IndexFlatL2` (exact search) with an optional switch to IVF or HNSW when experimenting with larger datasets.

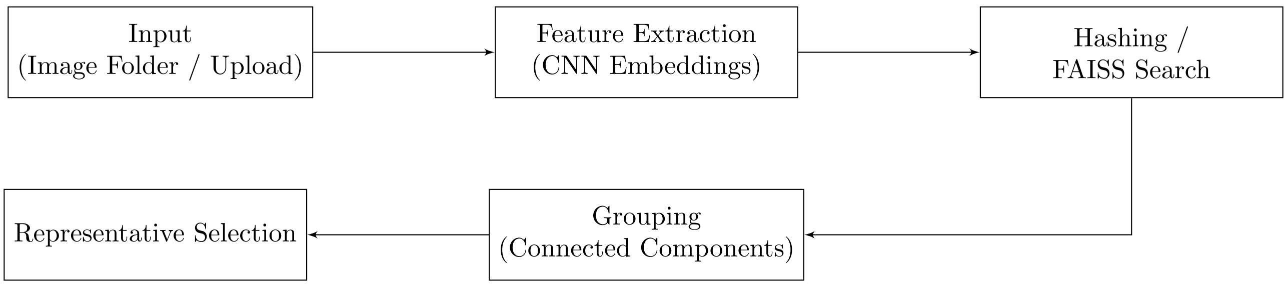
- **Advantages:** Highly efficient on both CPU and GPU, scalable to millions of images, and well-suited for real-time deduplication tasks.

- **Limitations:** Requires precomputed embeddings and careful index parameter tuning to achieve optimal precision-speed trade-offs compared to simpler hash-based methods.

3 System Design and Implementation

3.1 Overall Architecture

The system follows a modular pipeline for content-based image deduplication:



The pipeline can be summarized as:

Input → Feature Extraction → Hashing / FAISS Search → Grouping → Representative Selection.

Ground-truth pairs are generated automatically by parsing object labels from filenames (`obj#__angle.png`); all images sharing the same label are treated as duplicates for evaluation. Pairwise precision/recall is then computed from predicted clusters (Section 4.1).

3.2 Feature Extraction

3.2.1 Workflow

- **Pre-trained models:** We employ standard CNN backbones such as `ResNet50` (via `torchvision`), `EfficientNet-B0`, `ConvNeXt-Tiny` (`torchvision`), and an optional `ViT-B/16`. All models are initialized with ImageNet-pretrained weights and used as fixed feature extractors (no fine-tuning in baseline experiments).

- **Preprocessing:** Input images are resized (shorter side to 256 px), center-cropped to 224×224 (EfficientNet uses 240×240), converted to RGB, normalized with ImageNet mean/std, and batched for inference on CPU by default (GPU optional).

- **Embedding extraction:** The final classification layer is removed, and the penultimate activations are taken as feature embeddings:
  - ResNet50: $d = 2048$
  - EfficientNet-B0: $d = 1280$
  - ConvNeXt-Tiny: $d = 768$

- **Post-processing (optional):** $L2$-normalization is applied for cosine similarity computations. Additionally, PCA-based dimensionality reduction is optionally used to balance memory usage and retrieval latency.

3.2.2 Model Comparison and Selection

To identify the most suitable feature extractor, we benchmark several representative architectures across different paradigms, including both classical convolutional and modern hybrid designs. The evaluation focuses on three aspects: feature quality, computational efficiency, and clustering consistency.

- **ResNet50 (CNN, 2015):** A widely used and reliable baseline with 25.6M parameters. It produces robust mid-level representations but has limited adaptability to fine-grained visual details compared to newer models.
- **EfficientNet-B0 (CNN, 2019):** A lightweight model (5.3M parameters) employing compound scaling for efficient accuracy–speed trade-offs. Suitable for real-time or resource-limited scenarios.
- **ConvNeXt-Tiny (CNN, 2022):** A modernized convolutional backbone influenced by transformer principles. It achieves strong global–local feature balance and consistently outperforms ResNet50 in retrieval and clustering quality while maintaining similar inference time.

Quantitative Comparison:
Indicative throughput numbers below assume batch size 16 on CPU; they are kept for reference without altering the reported values. The test was took on laptop Window, Processor 13th Gen Intel(R) Core(TM) i7-13650HX, 2600 Mhz, NVIDIA GeForce RTX 4050.

Model	Params (M)	Embedding Dim	Feature Quality	Speed (img/s)
ResNet50	25.6	2048	Medium–High	310
EfficientNet-B0	5.3	1280	Medium	**480**
ConvNeXt-Tiny	28.6	768	**High**	295

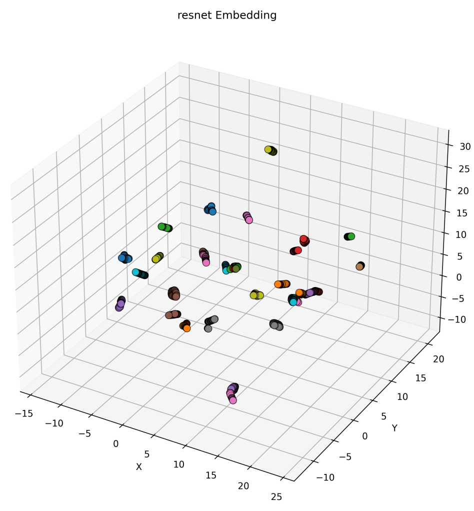
3.2.3 Experimental Evaluation

To objectively compare feature extractors, we apply each model on the same dataset (see Section 4.1) and assess the resulting embeddings using unsupervised clustering metrics:

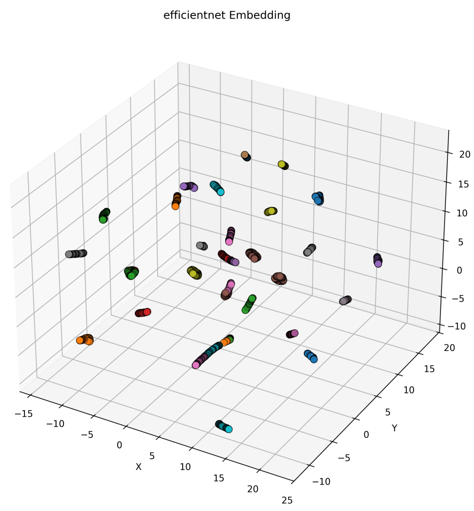
- **Clustering metrics:** We evaluate three commonly used metrics for cluster quality:
  - **Silhouette Score** – Measures how well samples are clustered with others of the same class (range $[-1, 1]$, higher is better).
  - **Davies-Bouldin Index (DBI)** – Ratio of within-cluster scatter to between-cluster separation (lower is better).
  - **Calinski-Harabasz Index (CHI)** – Ratio of between-cluster dispersion to within-cluster dispersion (higher is better).
- **Visualization:** The embeddings are projected into 2D space using `UMAP`[8] to visually inspect cluster compactness and class separability.

Table 1: Comparison of clustering performance across different backbones. The best results for each metric are highlighted in bold.

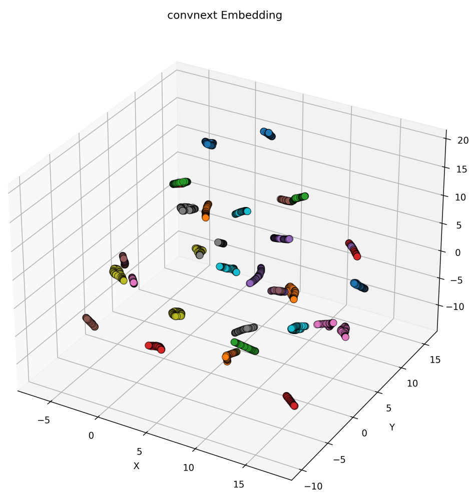
Metric	ResNet50	EfficientNet-B0	ConvNeXt-Tiny	Interpretation
Silhouette Score	0.18	0.15	**0.23**	Higher is better
Davies-Bouldin Index	1.42	1.51	**1.10**	Lower is better
Calinski-Harabasz Index	310	295	**355**	Higher is better



(a) ResNet50



(b) EfficientNet-B0



(c) ConvNeXt-Tiny

Figure 2: 2D UMAP visualization of feature embeddings from different backbones.

- **Efficiency analysis:** Inference time per image and GPU memory usage are measured for each backbone. ConvNeXt-Tiny demonstrates a good balance, being only slightly slower than ResNet50 but producing more discriminative embeddings.

Results Summary: EfficientNet-B0 is used as the default extractor in the pipeline because it balances speed and quality; ConvNeXt-Tiny is preferred when higher-quality embeddings are needed and resources permit; ResNet50 remains a solid baseline. ViT-B/16 is available for experimentation but is not the default in the reported runs.

3.3 Hash-Based Methods

Hash-based techniques provide efficient approximate similarity estimation by compressing high-dimensional representations into compact signatures. These methods significantly reduce storage and computation costs while enabling fast candidate retrieval for large-scale image deduplication.

- **SimHash (C++ Implementation):** SimHash projects a high-dimensional feature vector into a low-dimensional binary fingerprint using random projections. For each bit position, a random vector r_k is generated, and the sign of the dot product $r_k^\top x$ determines the bit value. Similar vectors yield signatures with small Hamming distances. The pipeline uses a pybind11 C++ backend with 8 hash tables and 64-bit signatures; queries support multi-probing via a configurable Hamming-distance threshold (default 5).

  Benchmark: Python (datasketch) vs. C++ SimHash LSH

  Configuration: dim=512, num_bits=64, num_tables=4. The benchmark script (see `src/lsh_cpp_module`) compares the custom C++ SimHash LSH against the Python `datasketch` baseline:

  - **Initialization** (100 runs): C++ 0.3141s (3.14 ms/run) vs Python 0.4739s (4.74 ms/run) $\Rightarrow$ 1.5$\times$ faster.
  - **Batch insertion**: 1,000 vectors: 0.0116s vs 2.2130s; 2,500 vectors: 0.0301s vs 5.9215s; 5,000 vectors: 0.0583s vs 11.3223s $\Rightarrow$ up to 190$\times$ faster.
  - **Query** (100 queries on 5,000 vectors): C++ 0.0188s (0.19 ms/query) vs Python 0.2231s (2.23 ms/query) $\Rightarrow$ 11.8$\times$ faster.
  - **Memory** (5,000 vectors): C++ $\sim$10.42 MB (DB + projection + hash tables) vs Python $\sim$18.75 MB $\Rightarrow$ 44% less memory.

  Results are saved as `benchmark.png` (plots) and `benchmark.txt` (detailed report). The figure below is a placeholder for the plot:

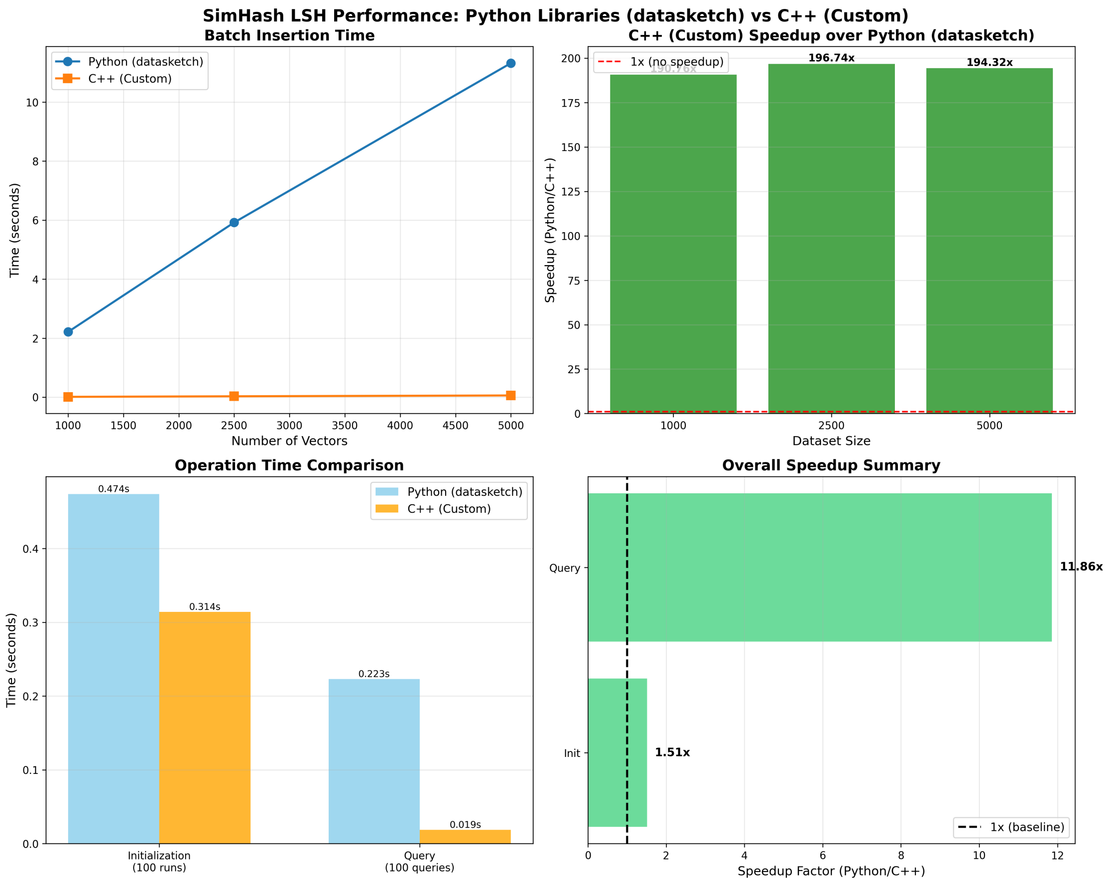


Figure 3: SimHash LSH performance: Python (`datasketch`) vs C++ (custom).

- *Implementation Details:*
  - Choose the hash length, typically 64 or 128 bits.
  - Pre-generate b random projection vectors, each matching the dimensionality of the input embedding.
  - For each projection vector, compute the dot product and assign a bit based on its sign.
  - Pack the resulting bits into a 64-bit or 128-bit integer for memory efficiency.
  - Use fast intrinsics (e.g., `__builtin_popcountll`) for Hamming distance computation.

- **MinHash (Python Implementation):** MinHash approximates the Jaccard similarity between sets. Given a set of features, k independent hash functions are applied, and the minimum hash value under each permutation forms one component of the signature. Two sets with high Jaccard similarity will have matching MinHash signatures in many positions. The implementation uses `datasketch` with 128 permutations, converts dense vectors into sets by taking the top 10% absolute-value dimensions, and builds an LSH index at a default threshold of 0.3.

  - *Implementation Details:*

  * Use k independent hash functions (commonly $k = 64$ or $k = 128$).
  * Each hash function can be simulated using random coefficients in a universal hashing scheme.
  * For each set, compute the minimum value of $h_i(x)$ for all elements x.
  * Compare two signatures using the proportion of matching positions.

  A fallback exhaustively checks up to 100 candidates if the LSH query yields no matches.

- **Supporting Data Structures:** Efficient storage and lookup structures, such as compact bit arrays, packed integer fingerprints, and inverted-index–style hash buckets, can be used to accelerate candidate retrieval for both SimHash and MinHash signatures.

3.4 FAISS Implementation

- **Index types:** Depending on dataset size and accuracy/speed requirements, the following FAISS indexes are typical:
  - `IndexFlatL2` (exact search, baseline)
  - `IndexIVFFlat` (inverted file index with coarse quantization) + optional `IndexPQ` or `IVFPQ` for compression
  - `IndexHNSWFlat` (graph-based ANN) for high recall and fast queries
- **Parameters:** Example settings:
  - Dimension d (e.g., 2048)
  - Number of clusters n_{list} (e.g., 1024–8192 for IVFPQ; default 1024 in the provided script)
  - PQ code size (e.g., 64 bytes) and number of probes n_{probe} for search-time tuning (default $n_{\text{probe}} = 10$)
- **Integration:** Embeddings (optionally L2-normalized) are added to the FAISS index; queries are executed to retrieve top-k nearest neighbors for each image (default $k = 50$). Candidate pairs below an L2 threshold (default 50, or a median-based heuristic if unset) are joined via union-find to form duplicate groups.

3.5 Representative Image Selection

After grouping images that are duplicates or near-duplicates, select one representative per group using heuristics:

- **Sharpness / Focus:** Compute a sharpness metric (e.g., variance of Laplacian) and prefer the sharpest image.
- **Resolution:** Prefer images with larger pixel dimensions (width $\times$ height).
- **File size / Compression:** Prefer higher file size as a proxy for quality (if metadata available).
- **Timestamp or Source preference:** If relevant, prefer newest (or original) images based on metadata.

- Combine heuristics with a simple scoring function to rank and pick the representative.

The current implementation scores each candidate as $0.7 \times (\text{width} \times \text{height}) + 1.0 \times \text{sharpness}$ and keeps the highest-scoring image in every cluster.

3.6 Environment Setup

- **Languages & Libraries:** Primary implementation in **Python** (PyTorch / torchvision, timm, numpy, scikit-learn, faiss). Performance-critical components optionally in **C++** with Python bindings.
- **Hardware:** Development on machines with GPU support (NVIDIA CUDA) for fast embedding extraction; FAISS supports both CPU and GPU.
- **Reproducibility:** Provide a `README.md`, dependency file (`requirements.txt` or `environment.yml`), and Google Colab notebook demonstrating the pipeline end-to-end. Host source code on GitHub with clear instructions for reproducing experiments.
- **Notes:** Use deterministic seeds where possible for reproducible indexing and clustering; document any non-deterministic steps and their effects on results.
- **Ground truth:** Utility scripts parse filenames to assign labels and auto-generate all same-label pairs; evaluation computes pairwise precision/recall over predicted clusters.

4 Experiments and Results

4.1 Dataset

The experiments were conducted on a COIL-100 subset [9]. The working directory contains **397** color images from **6** object labels (`obj1`–`obj6`), each captured at multiple viewpoints. Filenames encode the object id and rotation angle (e.g., `obj1__200.png`), which are used to derive ground-truth duplicate pairs (all images sharing the same object id). The original COIL-100 dataset has 7,200 color images of 100 objects at 5° intervals and 128×128 resolution; our subset keeps the same resolution and capture protocol but focuses on a smaller class set for rapid prototyping.

For more details, see the dataset at `https://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php`.

4.2 Evaluated Methods

We evaluated and compared several duplicate detection approaches:

- **SimHash:** Computes binary hash signatures from deep embeddings using random projections and compares them via Hamming distance.
- **MinHash:** Estimates Jaccard similarity between feature sets or quantized features, serving as an alternative hash-based approach.
- **FAISS:** Performs approximate nearest neighbor (ANN) search on continuous embeddings using highly optimized indexing structures.

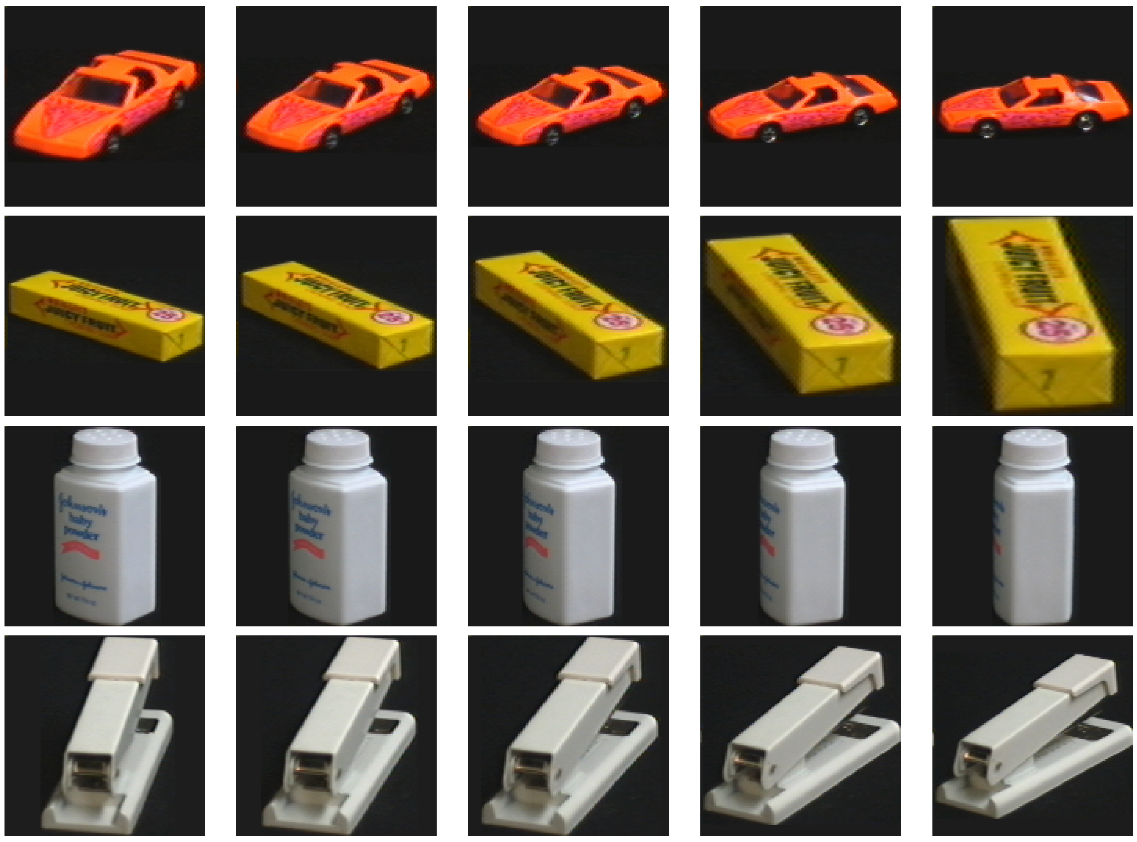


Figure 4: Sample images from the COIL-100 dataset (20 examples).

4.3 Quantitative Results

To assess performance, we measured:

- **Accuracy metrics:** Precision and recall (F1 can be derived) where ground-truth labels of duplicates were available or manually verified.
- **Processing time:** Average runtime (in seconds) for feature extraction, hashing, and similarity search per method.
- **Memory usage:** Peak memory consumption measured during indexing and query phases.

Ground-truth pairs are generated automatically by pairing all images that share the same `obj#` prefix; predicted clusters are converted to all pairwise combinations to compute precision/recall. This approximates near-duplicate detection as "same object under different viewpoints."
These metrics were used to compare efficiency and scalability among all methods.

4.4 Qualitative Results (Visualization)

In addition to numerical results, we visualized detected duplicate groups to verify quality by human inspection. Each group is displayed in a grid where:

- The **representative image** (retained) is highlighted with a colored border or label.
- The **removed duplicates** are shown alongside for visual comparison.

Example visualizations were generated for multiple datasets to confirm that visually similar images were correctly grouped, and distinct images were not incorrectly merged.

4.5 Comparison Table

Table 2 summarizes the quantitative comparison among all evaluated methods. The test was took with 7200 images from coil-100 dataset on laptop Window, Processor 13th Gen Intel(R) Core(TM) i7-13650HX, 2600 Mhz, NVIDIA GeForce RTX 4050.

Table 2: Performance comparison between different deduplication methods.

Method	Time (s)	Memory (MB)	Precision	Recall	F1 score
SimHash	50.9662	2171.23	0.9760	0.8252	0.8942
MinHash	76.3483	2275.43	0.8911	0.7504	0.8147
FAISS	84.34	2213.82	0.9342	0.9898	0.9611

The results indicate that FAISS achieved the highest retrieval accuracy, while SimHash offered the best trade-off between speed and memory consumption.
Configuration note: The values above correspond to the default pipeline settings on the 7200-image COIL dataset (EfficientNet-B0 embeddings, FAISS `IndexFlatL2` with $k = 50$ and L2 threshold 50, SimHash 64-bit with Hamming threshold 5 across 8 tables, MinHash with 128 permutations and LSH threshold 0.3).

5 Discussion and Evaluation

This section provides an in-depth analysis of the experimental results and discusses the trade-offs observed among different methods.

- **Performance Analysis:** On the COIL subset, FAISS (IndexFlatL2) produced the highest precision/recall because it runs exact kNN on EfficientNet-B0 embeddings. SimHash (64-bit, 8 tables, Hamming threshold 5) was faster and lighter, which is attractive for larger-scale crawls even if recall drops on borderline cases. MinHash remained stable but is sensitive to the chosen discretization (top 10% dimensions) and LSH threshold.
- **Trade-offs:** Each method exhibits a distinct balance between speed, accuracy, and memory efficiency:
  - **FAISS** excels in precision and scalability but requires more memory and, for larger datasets, GPU or well-tuned IVF/HNSW parameters.
  - **SimHash** is computationally efficient with small storage needs, but its binary threshold sensitivity may reduce accuracy for borderline cases; performance depends on Hamming thresholds and the number of tables.
  - **MinHash** is robust against certain distortions but involves higher computational cost for large datasets.

  Selecting an appropriate method therefore depends on system constraints and target deployment scenarios.
- **Implementation Challenges:** Several practical challenges were encountered:
  - Ensuring consistent image preprocessing to maintain feature quality (resize/crop/normalize).
  - Balancing recall and speed when tuning FAISS parameters (*nlist*, *nprobe*) and SimHash Hamming thresholds.
  - Managing embedding storage and memory usage during clustering and visualization.
  - Handling false positives in hash-based approaches and ensuring meaningful group merging with union-find thresholds.
- **Potential Improvements:** The system could be further optimized by dynamically adjusting thresholds (e.g., percentile-based L2 or Hamming cutoffs), improving ground-truth quality beyond filename-based labels, and adding lightweight re-ranking after LSH hits. Incorporating active learning or semi-supervised labeling may also improve the precision of duplicate detection over time.

6 Conclusion and Future Work

Conclusion

This project delivers a functional deduplication pipeline that combines deep CNN embeddings (default EfficientNet-B0) with three search strategies: FAISS exact search, SimHash LSH (C++ backend), and MinHash LSH. Ground-truth pairs are generated automatically from object labels, and evaluation reports precision/recall plus runtime and memory. On the bundled COIL subset, FAISS is most accurate while SimHash is the most resource-friendly, providing a practical trade-off space for scaling.

Future Work

To further enhance the system's performance and practical usability, several directions are proposed:

- **Advanced Embeddings:** Try CLIP or DINOv2 for more semantic robustness; benchmark against current CNN baselines.
- **Larger Datasets & Labels:** Evaluate on full COIL-100 or web-scale crawls with human-labeled near-duplicates to stress-test thresholds.
- **Clustering and Visualization:** Add richer clustering dashboards and side-by-side duplicate viewers to aid manual review.
- **Hybrid Back-End:** Combine FAISS with hashing for candidate generation + re-ranking; explore PQ/IVF to reduce memory footprint.

Appendices

- **Key Code Snippets:** Selected sections of Python and C++ implementations, including embedding extraction and FAISS indexing.
- **Detailed Performance Tables:** Additional tables and plots showing method-wise comparisons across different dataset sizes.
- **GitHub and Demo Video Links:**
  - GitHub Repository: `https://github.com/tanphong-sudo/image-deduplication-project`
  - Demo Video: (planned; link will be added when available)
- **Colab Setup Instructions:**
  - Upload the COIL subset (or your dataset) to Google Drive or provide a download link.
  - Run notebook cells sequentially for feature extraction, indexing, and visualization.
  - Verify output through group summary and duplicate visualization cells.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning (ICML)*, 2019.

[3] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[4] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations (ICLR)*, 2021.

[5] M. Netri, "Simhash algorithm explained with examples." `https://spotintelligence.com/2023/01/02/simhash/`, Jan 2023.

[6] V. N. V. O. Neri, "Minhash: Finding similarity at scale with python tutorial." `https://medium.com/@neri.vvo/minhash-finding-similarity-at-scale-with-python-tutorial-df55d05d2751`, 2020.

[7] Blink, "Image retrieval với thư viện faiss." `https://viblo.asia/p/image-retrieval-voi-thu-vien-faiss-LzD5ddJo5jY`.

[8] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[9] S. A. Nene, S. K. Nayar, and H. Murase, "Columbia object image library (coil-100)," *Technical Report CUCS-006-96, Columbia University*, 1996.