Image Recommender

Modul: Big Data Engineering
Dozent: Prof. Dr. Florian Huber
Sommersemester 2025

Thasarathakumar Palanisamy, 947552
Amilin Shahida Binti Saadon, 946804

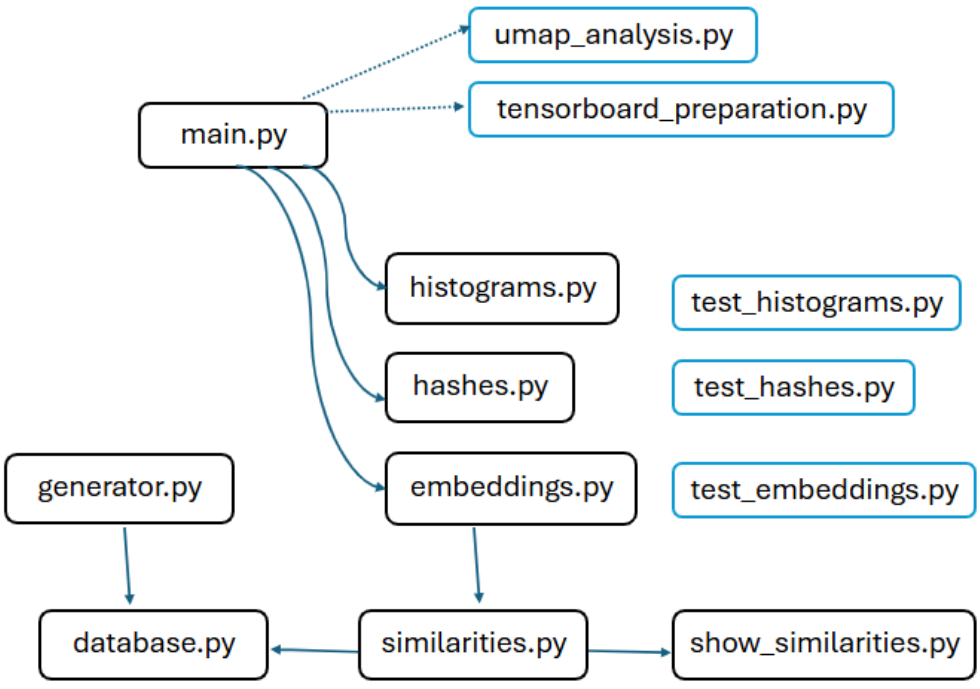
Inhaltsverzeichnis

1. Projektziel

Unser Image Recommender ist eine Python-basierte Anwendung, die auf Grundlage eines eingegebenen Bildes ähnliche Bilder aus einem umfangreichen Datensatz identifiziert und vorschlägt. Ziel des Projekts war es, ein effizientes und erweiterbares System zu entwickeln, das verschiedene Verfahren zur Ähnlichkeitsbestimmung integriert und deren Ergebnisse vergleichbar macht.

Die extrahierten Merkmale der Eingabebilder werden mit den zuvor berechneten Merkmalen der Datenbankbilder verglichen, sodass die ähnlichsten Ergebnisse vorgeschlagen werden können. Das Projekt wurde in Zweiergruppen im Rahmen unseres ersten Big-Data-Projekts umgesetzt. Neben der Implementierung der Verfahren lag der Schwerpunkt auf der Optimierung der Software für schnelle Abfragen sowie auf einer klaren Dokumentation und strukturierten Arbeitsweise.

2. Programmdesign



Die Architektur unseres Projekts ist bewusst modular aufgebaut, um eine klare Trennung der Verantwortlichkeiten zu schaffen und gleichzeitig die Erweiterbarkeit sowie Wartbarkeit des Codes zu gewährleisten. Im Zentrum steht die Datei *main.py*, die als Einstiegspunkt dient und den gesamten Ablauf der Bild-Empfehlungspipeline steuert. Von hier aus werden sämtliche Module angesteuert, die gemeinsam den Prozess vom Laden der Bilder, über die Berechnung von Ähnlichkeiten, bis hin zur Visualisierung der Ergebnisse abdecken.

Der Workflow beginnt mit der Ausführung von *generator.py*, welches als erste Instanz die Bilddaten verarbeitet. Dieses Skript durchsucht rekursiv das angegebene Verzeichnis, erkennt sämtliche Bilder und speichert die wichtigsten Metadaten, darunter UUIDs, Dateipfade und Bildgrößen. Diese Informationen werden in einer SQLite-Datenbank abgelegt und zusätzlich in Pickle-Dateien gespeichert, sodass sowohl eine schnelle Abfrage als auch eine effiziente Wiederverwendung der Daten möglich ist. Die eigentliche Interaktion mit der Datenbank wird dabei durch *database_utils.py* geregelt, das Funktionen zur Erstellung von Tabellen, zum Einfügen von Datensätzen und zum gezielten Abrufen einzelner Bildinformationen bereitstellt. Da in unserem Anwendungsfall lediglich grundlegende Informationen wie Pfad und UUID benötigt werden, bleibt die Datenbank bewusst schlank gehalten und ist damit auch bei größeren Datenmengen performant.

Um sicherzustellen, dass alle Module auf konsistente Strukturen zugreifen, existiert mit *config_paths.py* eine zentrale Konfigurationsdatei. In ihr werden sämtliche Pfade definiert, die innerhalb des Projekts verwendet werden. Auf diese Weise wird verhindert, dass unterschiedliche Module mit voneinander abweichenden Angaben arbeiten. Lediglich der Pfad zu den Bilddaten (*IMAGE_DATA_DIR*) muss von den Nutzerinnen und Nutzern selbst angepasst werden, da er auf das jeweilige Datenset verweist. Sobald dieser Pfad gesetzt ist, laufen alle weiteren Prozesse automatisch und greifen auf die gleichen, konsistenten Verzeichnisse zurück.

Nachdem die Datenbank erstellt wurde, übernimmt *main.py* die Steuerung der eigentlichen Berechnungen. Dieses Skript lädt die Bildpfade aus der Datenbank und iteriert anschließend über sämtliche Einträge. Für jedes Bild werden verschiedene Methoden der Feature-Berechnung angewandt, die in jeweils eigenen Modulen implementiert sind. Dabei kommen drei Ansätze zum Einsatz: Deep Learning mittels ResNet18, Farb-Histogramm-Analysen sowie Hashing. Das Modul *embeddings.py* berechnet Merkmalsvektoren mithilfe eines vortrainierten ResNet18-Modells, das aus der Bibliothek *torchvision* eingebunden ist. Die Bilder werden vorverarbeitet, in RGB konvertiert und normalisiert, bevor das ResNet-Netzwerk sie durchläuft und einen hochdimensionalen Vektor extrahiert, der die charakteristischen Eigenschaften des Bildes abbildet. Parallel dazu liefert *histograms.py* eine Analyse der Farbverteilungen, indem es RGB-Histogramme berechnet und normalisiert. Dieses Verfahren ist besonders nützlich, wenn Bilder anhand ihrer dominanten Farbmuster miteinander verglichen werden sollen. Ergänzt wird dies durch *hashes.py*, das mit den Verfahren A-Hash, D-Hash und P-Hash drei gängige Algorithmen implementiert, die für die Bestimmung visueller Ähnlichkeiten eingesetzt werden. Während der Durchschnitts-Hash (A-Hash) und der Differenz-Hash (D-Hash) auf einfachen Helligkeits- und Pixelmustervergleichen beruhen und dadurch sehr schnell berechnet werden können, geht der

Perceptual Hash (P-Hash) einen Schritt weiter. P-Hash basiert auf der diskreten Kosinustransformation (DCT) und analysiert damit die Frequenzanteile eines Bildes, anstatt nur dessen Pixelwerte zu betrachten. Dadurch ist er robuster gegenüber Bildtransformationen wie Skalierung, Kompression oder leichten Farbänderungen.

Alle diese Methoden werden schließlich in *similarities.py* zusammengeführt, das eine gemeinsame Schnittstelle bereitstellt, welche die Logik der Berechnungen kapselt und im Hintergrund ausführt. Die Auswahl des gewünschten Berechnungsmodus, also Embeddings, Hashes oder Histogramme, sowie der Distanzmetrik erfolgt durch die Nutzerinnen und Nutzer im Notebook *show_similarities.ipynb*, wo sie beim Aufruf von Funktionen wie `calculate_similarities` Parameter flexibel setzen können. Zur Verfügung stehen dabei Verfahren wie Cosine Similarity, euklidische Distanz, Manhattan-Distanz, Hamming-Distanz sowie die Bhattacharyya-Distanz. Zusätzlich lässt sich optional das FAISS-Framework (Facebook AI Similarity Search) aktivieren, um Approximate-Nearest-Neighbor-Suchen auf großen Datensätzen erheblich zu beschleunigen.

Ein weiteres wichtiges Element der Architektur ist die Visualisierung. Für diesen Zweck existieren mehrere Module, die verschiedene Anforderungen abdecken. *show_similarities.ipynb* dient als interaktive Benutzeroberfläche, über die Eingabebilder geladen und die Top-N ähnlichsten Bilder berechnet sowie nebeneinander dargestellt werden können. Für eine tiefergehende Analyse der Verteilung großer Datensätze wird *umap_analysis.ipynb* eingesetzt. Mithilfe des UMAP-Algorithmus lassen sich hochdimensionale Embeddings auf zwei oder drei Dimensionen reduzieren und grafisch darstellen, was es ermöglicht, Cluster und Muster im Datensatz zu erkennen. Zusätzlich ermöglicht *tensorboard_preparation.py* eine Integration der Ergebnisse in TensorBoard. Hierfür werden Sprite-Bilder und Projektionsdaten erzeugt, die anschließend in einer interaktiven Umgebung betrachtet werden können.

Zur Sicherstellung der Qualität sind separate Unit-Test-Dateien wie *test_histograms.py*, *test_hashes.py* und *test_database_utils.py* vorhanden. Jede dieser Dateien überprüft gezielt die Funktionalität der entsprechenden Module. So wird beispielsweise sichergestellt, dass Hashes korrekt berechnet, Histogramme normiert und Datenbankoperationen fehlerfrei ausgeführt werden. Durch diese systematische Qualitätssicherung lassen sich Fehler frühzeitig erkennen und beheben, ohne den gesamten Workflow zu gefährden.

Das System kombiniert Deep Learning (ResNet18), klassische Bildverarbeitung (Histogramme, Hashes) und eine relationale SQLite-Datenbank zu einem flexiblen Image-Recommender. Die modulare Struktur sorgt dafür, dass neue Methoden oder Visualisierungen jederzeit ergänzt werden können, während Checkpoints und Pickle-Dateien einen robusten, wiederaufnahmefähigen Workflow ermöglichen.

3. Bild Ähnlichkeitsmaße

Um die Ähnlichkeit zwischen Eingabe- und Datenbankbildern zu bestimmen, wurden im Projekt drei Ansätze implementiert und miteinander verglichen: **Farbhistogramme** (in RGB), **CNN-Embeddings** (ResNet18) sowie **Hashing** (pHash, dHash, aHash). Jede Methode fokussiert sich auf unterschiedliche visuelle Merkmale der Bilder und liefert dadurch komplementäre Ergebnisse bei der Ähnlichkeitsbestimmung. Die extrahierten Merkmale wurden mithilfe verschiedener Distanz- und Ähnlichkeitsmaße (z. B. Kosinus, Euklidisch, Manhattan, Bhattacharyya, Hamming, Approximate Nearest Neighbor (ANN) Cosine) verglichen, um die *top-k* ähnlichsten Bilder zu ermitteln.

3.1 RGB-Farbhistogramm

Diese Methode basiert auf der Analyse der Farbverteilung in Bildern mithilfe von Farbhistogrammen. Im Projekt haben wir ein dreidimensionales Histogramm im RGB-Farbraum erstellt, das mit der OpenCV-Funktion `cv2.calcHist()` berechnet wird. Dabei werden die Farbwerte in vordefinierte Intervalle (Bins) eingeteilt – wir haben 8 Bins pro Kanal gewählt.

Anschließend wird das Histogramm mit `cv2.normalize()` (L1-Norm) skaliert, sodass Bilder unabhängig von ihrer Größe vergleichbar bleiben, und mit `flatten()` in eine eindimensionale Form gebracht, um es als `numpy`-Array weiterzuverarbeiten.

Zur Bestimmung der Ähnlichkeit zwischen zwei Bildern werden die normalisierten Histogramme mit verschiedenen Distanzmaßen verglichen:

- **Kosinus-Similarität**: Misst den Winkel zwischen den Histogrammvektoren.
    - Vorteil: Vergleich der Form der Verteilung, unabhängig von der absoluten Helligkeit.
    - Nachteil: Bilder mit ähnlicher Verteilung, aber stark unterschiedlichen Farbintensitäten, erscheinen trotzdem ähnlich.
- **Euklidische Distanz**: Misst den *„Luftlinienabstand“* zwischen zwei Histogrammvektoren.
    - Vorteil: Intuitiv und leicht verständlich.
    - Nachteil: Empfindlich gegenüber Helligkeits- oder Kontraständerungen.
- **Manhattan-Distanz**: Summiert die absoluten Abweichungen der Histogrammwerte.
    - Vorteil: Weniger empfindlich gegenüber Ausreißern als die euklidische Distanz.
    - Nachteil: Kann Unterschiede gleichmäßig „aufsummieren“, auch wenn sie visuell kaum relevant sind.

- **Bhattacharyya-Distanz**: Bewertet die Überlappung zweier Wahrscheinlichkeitsverteilungen.
    - Vorteil: Sehr präzise, da Form und Lage der Verteilungen berücksichtigt werden.
    - Nachteil: Etwas rechenintensiver als die anderen Metriken.

Vorteile von Farb-Histogrammen

- **Robustheit gegenüber Skalierung und Rotation**: Da keine räumlichen Informationen berücksichtigt werden, sind Histogramme unempfindlich gegenüber Änderungen in Bildgröße oder Orientierung.
- **Effiziente Berechnung**: Schnell berechnet, ideal für große Bilddatenbanken.

Nachteile von Farb-Histogrammen

- **Verlust räumlicher Informationen**: Zwei Bilder mit gleicher Farbverteilung, aber unterschiedlicher Struktur, können fälschlicherweise als ähnlich gelten.
- **Eingeschränkte Genauigkeit**: Bei komplexen Bildern mit vielen Details oder Objekten reicht diese Methode oft nicht aus.

3.2 ResNet Embeddings

Ein zentraler Ansatz unseres Projekts ist die Nutzung von Convolutional Neural Networks (CNNs) zur Extraktion von Bildmerkmalen. Wir haben uns für ResNet18 entschieden, da es ein bewährtes und vergleichsweise leichtgewichtiges Modell ist, das dennoch sehr leistungsfähig ist. ResNet18 hat deutlich weniger Parameter als größere Varianten wie ResNet50 oder ResNet101, bietet aber eine gute Balance zwischen Genauigkeit und Rechenaufwand, ein wichtiger Faktor für unser Projekt mit einem großen Datensatz.

Bei der Feature-Extraktion wird die letzte Klassifikationsschicht durch eine Identity-Funktion ersetzt, sodass das Modell keine Klassenvorhersage mehr ausgibt, sondern einen Embeddingsvektor bereitstellt. Dieser Vektor (512 Dimensionen) enthält die wichtigsten visuellen Merkmale des Bildes, darunter Formen, Texturen und Muster.

Die Bilder werden zunächst mit einer Preprocessing-Pipeline vorbereitet: Skalierung, Center Crop, Umwandlung in einen Tensor und Normalisierung basierend auf den Standardwerten von ResNet. Anschließend wird das vorbereitete Bild in die Funktion get_embedding() eingespeist, wo es mit ResNet18 verarbeitet wird. Das Modell wird einmalig beim Start geladen, auf das

verfügbare Gerät (GPU oder CPU) verschoben und im Evaluierungsmodus (eval()) betrieben. Die Berechnung erfolgt innerhalb eines *torch.no_grad()*-Blocks, um Speicher zu sparen und unnötige Gradientenberechnungen zu vermeiden.

Berechnung der Ähnlichkeit

Zur Bestimmung der Ähnlichkeit zweier Bilder haben wir zwei Verfahren eingesetzt:

- **Cosine Similarity**: Misst den Winkel zwischen zwei Embedding-Vektoren. Je kleiner der Winkel, desto ähnlicher die Bilder. Dies ist die exakte, aber rechenintensivere Variante.

- **ANN Cosine (Approximate Nearest Neighbor)**: Zur Beschleunigung auf großen Datensätzen wird FAISS (Facebook AI Similarity Search) eingesetzt. Dabei werden alle Embeddings einmalig L2-normalisiert und in einen IndexFlatIP-Index eingefügt. Die Cosine Similarity wird hier über das Skalarprodukt auf den normalisierten Vektoren approximiert. Dieses Verfahren erlaubt es, sehr schnell die Top-k ähnlichsten Bilder zu finden, ohne jedes Embedding exakt vergleichen zu müssen.

Vorteile der Embedding-Methode

- **Hohe Genauigkeit**: CNNs wie ResNet18 erfassen komplexe Strukturen im Bild (Texturen, Formen, Objekte), die weit über reine Farbverteilungen hinausgehen.

- **Generalisierbarkeit**: Funktioniert für eine große Bandbreite an Bildern, unabhängig vom Inhalt.

- **Robustheit**: Weniger anfällig für Veränderungen in Perspektive, Beleuchtung oder Skalierung.

- **Effizienz**: ResNet18 ist kleiner als ResNet50/101 und dadurch schneller und ressourcenschonender.

- **ANN Cosine**: Durch FAISS können auch bei sehr großen Datenbanken schnelle und skalierbare Abfragen durchgeführt werden.

Nachteile der Embedding-Methode

- **Rechenaufwand**: Auch ResNet18 benötigt im Vergleich zu klassischen Methoden (z. B. Histogramme) deutlich mehr Rechenleistung.

- **Blackbox-Phänomen**: Es ist schwer nachvollziehbar, welche Merkmale genau zur Ähnlichkeit beitragen.

- **Speicherintensiv**: Selbst bei 512 Dimensionen ist das Speichern und Vergleichen großer Datenmengen aufwendig, FAISS löst nur teilweise das Problem.

3.3 Hashing

Neben Farbhistogrammen und Embeddings haben wir als dritte Methode Image Hashing eingesetzt. Dabei wird jedem Bild ein kompakter, binärer *„Fingerabdruck“* zugeordnet, der die wesentlichen Strukturen des Bildes repräsentiert. Ziel ist es, dass visuell ähnliche Bilder ähnliche Hashwerte besitzen, während unterschiedliche Bilder klar unterscheidbare Hashes erzeugen.

In unserem Projekt haben wir drei Varianten implementiert, die der/die Nutzer:in auswählen kann:

- **Average Hash (aHash)**:
  Das Bild wird in Graustufen umgewandelt und verkleinert. Anschließend wird der Durchschnittswert der Pixelintensitäten berechnet. Jeder Pixel wird mit diesem Wert verglichen: kleiner → 0, größer → 1.
    - **Vorteil:** Sehr einfach und effizient zu berechnen.
    - **Nachteil:** Nur Helligkeitsverläufe werden erfasst; keine Berücksichtigung von Farben oder komplexeren Strukturen.
- **Difference Hash (dHash)**:
  Das Bild wird ebenfalls verkleinert und in Graustufen umgewandelt. Statt Mittelwerten werden Helligkeitsunterschiede zwischen benachbarten Pixeln betrachtet. Ist ein Pixel heller als sein Nachbar → 1, sonst → 0.
    - **Vorteil:** Erfasst Kanten und Strukturen besser als aHash.
    - **Nachteil:** Empfindlich gegenüber kleinen Änderungen
- **Perceptual Hash (pHash)**:
  Nach der Umwandlung in Graustufen wird eine Diskrete Kosinustransformation (DCT) angewendet. Aus dem niederfrequenten Bereich der DCT werden die wichtigsten Koeffizienten ausgewählt. Diese werden mit ihrem Durchschnittswert verglichen und als 0/1 kodiert.
    - **Vorteil:** Deutlich robuster gegenüber Transformationen wie Skalierung, leichter Rotation oder Kompression. Liefert meist stabilere Ergebnisse als aHash oder dHash.

- **Nachteil:** Etwas aufwendiger zu berechnen.

Ähnlichkeitsberechnung

Für alle Hash-Varianten wird die Hamming-Distanz eingesetzt. Diese misst, wie viele Bits zwischen zwei Hashes unterschiedlich sind.

- **Kleiner Wert → hohe Ähnlichkeit**
- **Großer Wert → große Unterschiede**

Da Hashes nur 64 Bits (oder vergleichbar kompakt) umfassen, ist der Vergleich extrem schnell und speichereffizient.

Vorteile der Hashing-Methode

- **Kompakte Repräsentation**: Jedes Bild wird auf wenige Bits reduziert, was Speicherung und Vergleiche sehr effizient macht.

- **Schnelligkeit**: Hamming-Distanzen sind sehr einfach zu berechnen, auch bei großen Datenmengen.

- **Robustheit (v. a. pHash)**: Kleine Transformationen (Skalierung, Kompression, leichte Rotation) ändern den Hash kaum.

Nachteile der Hashing-Methode

- **Keine semantische Information**: Hashes erfassen visuelle Strukturen, aber nicht den Bildinhalt (z. B. Hund vs. Katze können ähnliche Hashes haben).

- **Verlust von Farbinformationen**: Da nur Graustufen betrachtet werden, gehen Farbunterschiede vollständig verloren.

- **Begrenzte Aussagekraft**: Bei komplexen, detailreichen Bildern können Unterschiede im Hash auftreten, obwohl die Bilder inhaltlich ähnlich sind.

3.4 Vergleich der Methoden

Farb Histogramme

- **Stärken**: Schnell zu berechnen, benötigen wenig Speicher und sind robust gegenüber Skalierung und Rotation. Besonders dann, wenn die Farbverteilung das zentrale Unterscheidungsmerkmal ist (z. B. bei Bildern von Landschaften, Flaggen oder Logos), liefern Histogramme gute Ergebnisse.

- **Schwächen**: Vernachlässigen räumliche Informationen und können dadurch Bilder fälschlich als ähnlich einstufen, wenn diese nur ähnliche Farbpaletten besitzen, aber völlig unterschiedliche Inhalte darstellen. Zudem können manche Metriken (z. B. Euclidean) empfindlich auf Helligkeits- oder Kontraständerungen reagieren.

ResNet18-Embeddings

- **Stärken**: CNN-Embeddings erfassen komplexe Merkmale wie Formen, Texturen und Objekte im Bild. Dadurch liefern sie die genauesten Ergebnisse, insbesondere bei semantisch anspruchsvollen Vergleichen (z. B. „Hundebilder mit ähnlicher Pose“). Mit Cosine Similarity werden die Vektoren exakt verglichen, während ANN Cosine (FAISS) eine deutlich schnellere und speichereffiziente Suche in großen Datenbanken ermöglicht, ohne nennenswerte Qualitätseinbußen.

- **Schwächen**: Die Extraktion der Embeddings ist rechenintensiv und benötigt GPUs oder lange Laufzeiten auf der CPU. Zudem sind die Embeddings schwer interpretierbar, da sie aus einem hochdimensionalen Vektorraum stammen.

Image Hashing (aHash, dHash, pHash)

- **Stärken**: Sehr effizient in Speicher und Geschwindigkeit. Hashes eignen sich besonders für *Near-Duplicate Detection* (z. B. gleiche Bilder mit leichter Rotation, Skalierung oder Kompression). Unter den Varianten ist pHash am robustesten, da es über die DCT auch Frequenzinformationen berücksichtigt.

- **Schwächen**: Keine semantische Information, keine Farbinformationen. Zwei Bilder mit ähnlichen Strukturen, aber völlig unterschiedlichen Objekten, können fälschlich als ähnlich eingestuft werden. aHash und dHash sind zudem weniger robust als pHash.

Gesamtfazit

- **Genauigkeit**: Die Embedding-basierte Methode mit ResNet18 liefert die präzisesten Ergebnisse und ist die erste Wahl für inhaltlich komplexe Vergleiche.

- **Effizienz**: Hashing ist unschlagbar, wenn es um Geschwindigkeit und Speicher geht, eignet sich aber primär zur Erkennung nahezu identischer Bilder.

- **Balance**: Color Histogramme bieten eine gute Zwischenlösung, da sie einfach, schnell und für farbdominierte Vergleiche sehr nützlich sind.

Insgesamt zeigt sich, dass keine einzelne Methode „die beste“ ist, sondern dass der Einsatz stark vom Anwendungskontext abhängt. Für semantisch komplexe Vergleiche sind CNN-Embeddings die erste Wahl, für Duplikaterkennung eignet sich Hashing, und für Farbanalysen oder stilistische Vergleiche bieten Histogramme eine solide Grundlage.

4. Performanceanalyse und Laufzeitoptimierung

Die Pipeline lässt sich in zwei Hauptschritte gliedern:

1. **Feature-Generierung** *(Embeddings, Histogramme, Hashes, Metadaten)*
2. **Similarity Search** *(Vergleich und Ausgabe ähnlicher Bilder)*

Die Analyse hat gezeigt, dass die Feature-Generierung erwartungsgemäß den größten Zeitaufwand beanspruchte, während die Similarity Search vor allem durch algorithmische Details und Datenzugriff beeinflusst wurde.

1. Erstellung der Embeddings (ResNet18)

- **Laufzeit**: deutlich reduziert gegenüber ResNet50 (ursprünglich 15 h 48 min → nun ca. 8 h bei gleichem Datensatz)
- **Probleme**: CNN-Feature-Extraktion ist rechenintensiv.
- **Optimierungen**:
    - Umstieg von ResNet50 auf ResNet18, wodurch die Laufzeit halbiert werden konnte, ohne die Ergebnisqualität wesentlich zu beeinträchtigen.
    - Modell wird einmalig im Main Loop geladen statt bei jedem Aufruf.
    - Nutzung von CUDA (GPU), falls verfügbar.
    - Datentyp von `float64` auf `float32` reduziert → geringerer Speicherverbrauch.

- Checkpoints zur Fortsetzung bei Abbrüchen.

2. Berechnung der Farbhistogramme

- **Laufzeit**: ca. 15 h 18 min bei 8 Bins pro Kanal.
- **Probleme**: Mehrfaches Laden derselben Bilder und Rechenlast durch große Bin-Größe.
- **Optimierungen**:
  - Einführung eines zentralen Main Loops: Jedes Bild wird nur einmal geladen und kann für alle genutzten Metriken (Cosine, Euclidean, Manhattan, Bhattacharyya) weiterverwendet werden.
  - Reduktion redundanter Rechenoperationen.

3. Hashing (aHash, dHash, pHash)

- **Laufzeit**: vernachlässigbar (< 1 h für gesamten Datensatz).
- **Probleme**: Keine gravierenden, da Hashes klein und schnell berechenbar sind.
- **Optimierungen**:
  - Vergleich ausschließlich über Hamming-Distanz.

4. Einrichtung der Datenbank

- **Laufzeit**: ~8 h
- **Optimierungen**:
  - Entfernen unnötiger Attribute (z. B. Bildgröße).
  - Speicherung von Metadaten zusätzlich in Pickle-Dateien, um wiederholtes Laden zu beschleunigen.

5. Similarity Search

- **Laufzeit**: ~7,3 s pro Abgleich (inkl. Plotten)
- **Optimierungen**:
    - Nutzung vektorisierter NumPy-Berechnungen für Distanzen und Cosine Similarity.
    - Ersatz von `np.argsort` (O(n log n)) durch `np.argpartition` (O(n) + O(k log n)) zur Ermittlung der Top-k Ergebnisse.
    - **Caching**: Einmaliges Laden der Pickle-Dateien, danach Zugriff aus dem Speicher → drastische Beschleunigung.
    - **Besonderheit Cosine**: langsamer als Euclidean/Manhattan, da Normalisierungen nötig.
    - **FAISS** *(ANN Cosine)*: Aufbau eines Index über alle Embeddings → Suche nach Top-k Nachbarn in 2–3 Sekunden möglich, auch bei großen Datenmengen.

6. Bilddarstellung *(Plotting)*

- **Laufzeit**: ~3,8 s pro Durchlauf.
- **Probleme**: Im Verhältnis zur Ähnlichkeitsberechnung relativ hoch.
- **Optimierungsmöglichkeiten**: Batch-Plotting oder GPU-gestützte Visualisierung.

Gesamtfazit zur Performance

- Feature-Generierung bleibt der größte Zeitfaktor (v. a. Embeddings, Histogramme).
- Main Loop + Caching waren die effektivsten Optimierungen.
- ANN Cosine (FAISS) macht die Similarity Search skalierbar für sehr große Datensätze.
- Mit den Optimierungen liegt die Laufzeit einer typischen Anfrage nach dem ersten Aufruf im Bereich von 2–3 Sekunden, was für interaktive Anwendungen praktikabel ist.

6. Feasibility Discussion

Auf Basis der Performanceanalyse lässt sich festhalten, dass unsere Software für den gegebenen Datensatz grundsätzlich machbar und effizient arbeitet. Die Pipeline ist in der Lage, Bildmerkmale zu extrahieren, in einer Datenbank abzulegen und anschließend eine Ähnlichkeitssuche mit akzeptablen Laufzeiten durchzuführen.

Skalierbarkeit nach Verfahren

- **Color Histogramme**: Sehr effizient, da die Vektoren klein sind. Für einige hunderttausend Bilder problemlos nutzbar, insbesondere wenn Farbverteilungen im Vordergrund stehen. Bei Millionen von Bildern steigen die I/O-Kosten für das Laden und Vergleichen der Histogramme, aber prinzipiell bleibt die Methode relativ leichtgewichtig.

- **ResNet18-Embeddings**: Genaueste, aber speicher- und rechenintensivste Methode. Für Datenmengen im Bereich einiger hunderttausend Bilder gut nutzbar, da die Embeddings (512 Dimensionen, float32) im Speicher ~2–3 GB ausmachen. Ab mehreren Millionen Bildern stößt die Pipeline jedoch an RAM-Grenzen, da aktuell alle Features in einer Pickle-Datei geladen werden.

- **Image Hashing (aHash, dHash, pHash)**: Extrem effizient in Speicher und Laufzeit, da Hashes kompakte Binärvektoren sind. Diese Methode lässt sich problemlos auf Millionen von Bildern anwenden, eignet sich aber inhaltlich nur für Near-Duplicate Detection und nicht für semantische Suche.

Haupt Limitationen

1. **Feature-Generierung**: Das Erstellen der Embeddings ist trotz ResNet18 zeitintensiv (mehrere Stunden für Hunderttausende Bilder). Hier sind GPUs entscheidend, um Laufzeiten akzeptabel zu halten.

2. **Speicherverwaltung**: Momentan werden alle Features in einer großen Pickle-Datei gespeichert und beim Abruf komplett in den Arbeitsspeicher geladen (aktuell ~2,6 GB). Dies ist für kleinere bis mittelgroße Datenbanken tragbar, skaliert aber schlecht bei Millionen von Bildern.

3. **I/O-Bottleneck**: Das Laden sehr großer Bilder von der SSD kann den Durchsatz erheblich verlangsamen.

Mögliche Lösungen

- **Numba**: Könnte einige Python-Loops (z. B. Hamming- oder Bhattacharyya-Berechnung in reiner Python-Implementierung) beschleunigen; aktuell jedoch nur begrenzter Bedarf, da NumPy und OpenCV bereits sehr effizient sind.

- **Parallelisierung**: Durch den Einsatz von Multiprocessing oder Dask ließen sich Berechnungen auf größere Datenmengen verteilen, was insbesondere bei sehr großen Bilddatenbanken sinnvoll sein könnte.

Fazit

Die Software ist optimal für kleine bis mittelgroße Bilddatenbanken *(bis einige hunderttausend Bilder)*. Für sehr große Datenmengen *(Millionen von Bildern)* stößt die aktuelle Architektur jedoch an technische Grenzen (RAM, I/O, Rechenzeit).

7. Big Data Image Analysis

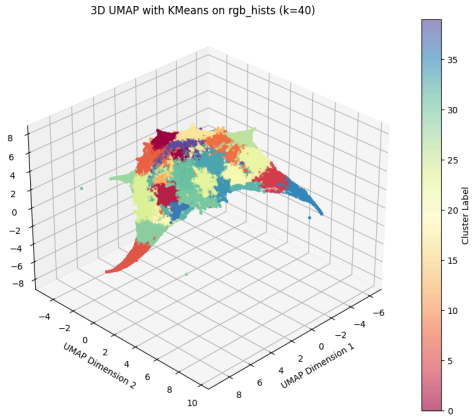
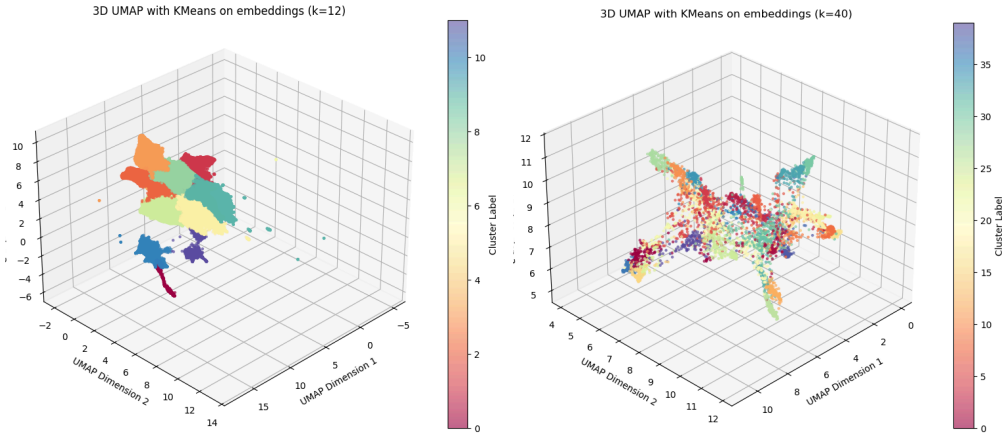


Abbildung A



Abbildungen B & C

Zur Analyse haben wir sowohl RGB-Histogramme als auch ResNet18-Embeddings mit UMAP in 2D/3D-Scatter Plots projiziert.

- RGB-Histogramme *(Abbildung A)*: Die Projektion zeigt ein großes, zentrales Cluster, in dem die meisten Bilder zusammenliegen, da viele ähnliche Farbprofile haben. Bilder mit sehr eindeutigen Farbverteilungen (z. B. fast nur Weiß, Schwarz oder stark gesättigte Farben) erscheinen am Rand als Ausreißer. Die Clustertrennung ist weniger scharf, da Histogramme nur Farbverteilungen, aber keine Formen oder Objekte erfassen.

- Embeddings *(Abbildungen B & C)*: Die ResNet18-Embeddings liefern eine deutlich differenziertere Struktur. Bei k=12 Clustern *(Abb. B)* entstehen große, klar erkennbare Gruppen, die semantische Ähnlichkeiten widerspiegeln. Bei k=40 Clustern *(Abb. C)* zerfallen diese in kleinere Untergruppen, wodurch feine Unterschiede sichtbar werden, gleichzeitig aber mehr Überschneidungen auftreten. Das zeigt, dass die Embeddings deutlich mehr Informationen (Texturen, Objekte, Muster) erfassen als Histogramme.

Als Verfahren haben wir uns bewusst für UMAP entschieden, da es im Gegensatz zu t-SNE wesentlich schneller arbeitet und trotzdem hochwertige Resultate liefert.

Die größte Herausforderung lag in der hohen Dimensionalität (512-d) und Datenmenge. Dies haben wir mit float32-Features, Caching und einer ANN-Suche via FAISS (Cosine Similarity) gelöst. Dadurch lassen sich auch große Datenmengen (Hunderttausende bis Millionen Bilder) effizient verarbeiten:

Fazit: Die Positionen der Bilder im reduzierten Raum machen Sinn:

- RGB-Histogramme → gruppieren Bilder hauptsächlich nach Farben.

- Embeddings → liefern eine semantische Struktur, die Inhalte und Objekte berücksichtigt. Die Wahl der Clusterzahl bestimmt den Detailgrad: wenige Cluster = Übersichtlichkeit, viele Cluster = feinere Unterschiede, aber weniger klar trennbar.