Dokumentation:
Big Data Image Recommender

D4.1.2 Big Data Engineering

Projekt von:
Nikita Rosik (922938),
Pepe Krumminga (924164),
Alina Weidemann (909551)

Lehrende:
Dr. Florian Huber,
David Sokalski

Abgabedatum:
11.02.2025

Inhaltsverzeichnis

1. Ziel und Motivation des Projekts

Unser Image Recommender ist eine Anwendung, die auf Grundlage eines und/oder mehrerer eingegebener Bilder ähnliche Bilder aus einem umfangreichen Datensatz von rund 450.000 Bildern identifiziert und vorschlägt. Dafür werden Merkmale der Bilder mithilfe von vier verschiedenen Methoden extrahiert: über Embeddings, A-Hashes, D-Hashes und Farbhistogramme. Somit werden die charakteristischen Eigenschaften der Bilder in komprimierter Form dargestellt und miteinander verglichen, um das ähnlichste Ergebnis zu liefern.

Um eine effiziente und gezielte Suche zu gewährleisten, vergleichen wir die extrahierten Merkmale der Eingabebilder mit den zuvor berechneten Merkmalen der Datenbankbilder. Zudem kann beim Ausführen der dafür vorgesehenen Datei “show_similarities.ipynb” flexibel zwischen den vier Metriken gewählt werden sowie die Auswahl an einem oder mehreren Eingabebildern getroffen werden. Durch das Benutzen von verschiedenen Metriken können unterschiedliche Ergebnisse erzielt werden, basierend darauf wonach die Ähnlichkeiten zwischen Bildern gemessen werden. Die Dauer der Ausgabe beträgt im Durchschnitt 3 Sekunden.

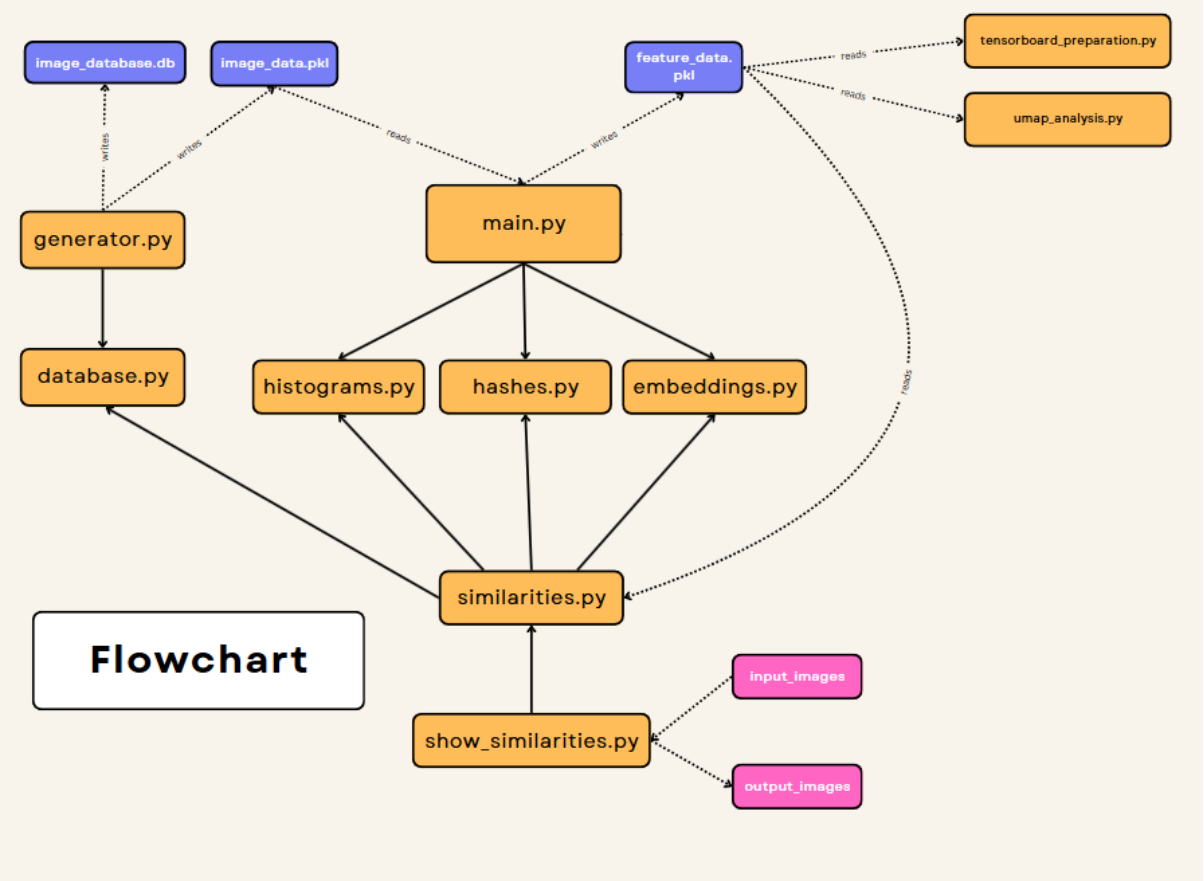
Dieses Projekt wurde von Nikita Rosik, Pepe Krumminga und Alina Weidemann entwickelt. Während der Projektarbeit wurde besonders Wert auf eine klare Code-Struktur, eine strukturierte Dokumentation bzw. Vorgehensweise sowie regelmäßige Kommunikation untereinander, gelegt. Die Bewertung des Wahlmoduls umfasst das GitHub-Repository, folgende Projektdokumentation sowie eine Präsentation der Ergebnisse.

2. Die Struktur des Image Recommenders

In unserem Github-Repo (https://github.com/honnigmelone/image_recommender) haben wir einen Image-Recommender entwickelt, der insgesamt drei Methoden kombiniert, um Bilder miteinander zu vergleichen und die ähnlichsten zueinander zu finden. Das System basiert auf verschiedenen Techniken wie Deep Learning anhand eines ResNet18 Netzwerks, Bildverarbeitung, Hashing und Histogramm-Analyse, sowie der Verwendung einer relationalen SQLite-Datenbank zur Verwaltung der geladenen Bildmetadaten und Pickle-Dateien zur Speicherung von den Embeddings. Es umfasst separate Dateien, die jeweils unterschiedliche Funktionen ausführen. Die ganzen Pfade wurden vereinheitlicht und in der config.py gespeichert, damit diese einfach importiert werden können. Diese Konfiguration stellt sicher, dass alle anderen Module auf dieselben grundlegenden Daten zugreifen können und somit keine fehlerhaften Lücken

entstehen. Der einzige Pfad, welcher angepasst werden muss, ist der IMAGE_DATA_PATH, welcher den Pfad zu den eigenen Bildern darstellt. Alles andere läuft automatisch. Die config.py haben wir aus dem Flowchart weggelassen, da nur Pfade importiert werden und diese zu den Dateien zeigen. Das wäre nur unübersichtlich. Um die Pipeline selbst zu starten, muss zuerst der Pfad zu den eigenen Daten geändert werden, dann führt man die Dateien generator.py und danach main.py aus. Wenn eine Feature-Data Pickle generiert wurde, kann das Jupyter-Notebook show_similarities.py ausgeführt werden, wo man Ähnliche Bilder generiert bekommt. Die oder das input image wird dabei als Liste übergeben.

Hier ist ein Flow-Chart mit dem Allgemeinen Fluss des Repos.



Der Flow startet mit der generator.py Datei. Dort wird ein Generator gestartet, welcher rekursiv root, file aus den Bildern lädt und mit einer eindeutigen image_id in der image_data.pkl (Pickle-Datei) speichert. Zudem wird dort direkt danach die Sqllite3 Datenbank erstellt und befüllt. Dieser Prozess dauert nur wenige Sekunden. Andere Metadaten haben wir in unserem Prozess nicht genutzt und fanden es nicht nötig diese zu speichern. Um die Datenbank zu generieren und zu befüllen wird die Methode generate_insert_into_database() aus database.py aufgerufen. Darin wird die Datenbank erstellt und die relevanten Daten eingefügt. In database.py finden sich dafür Funktionen wie create_table(), die (falls noch nicht vorhanden) eine Tabelle für die Bildmetadaten

erstellt, insert_data(), was die Metadaten einträgt, sowie load_data(), um Metadaten aus einer bereits gespeicherten Pickle-Datei zu laden. Über select_image_from_database(image_id, cursor) kann zudem später der Dateipfad eines Bildes anhand seiner ID abgefragt werden, was für die Ähnlichkeitssuche relevant ist. Weil wir in diesem Prozess nur root, file und image_id benötigen, war das Speichern weiterer Metadaten nicht nötig.

Wenn die image_data.pkl erstellt wurde, ist der nächste Schritt die main.py auszuführen. Das kann lange dauern (bis zu einem Tag), da dort alle relevanten Embeddings für die jeweiligen Features erstellt werden. Dort wird die Pickle Datei mit den Metadaten geladen und durch die filepaths iteriert. Wir haben eine Checkpoint-Funktion eingefügt, sodass bei Problemen problemlos abgebrochen werden kann. Beim erneuten Starten werden die zwischengespeicherten Daten geladen und es wird bei der nächsten image_id weitergemacht, welche noch nicht gespeichert ist. Daraufhin wird das Resnet18-Model geladen und dann wird erst durch die Daten iteriert. Wir haben die Library tqdm benutzt, um stets den Fortschritt beobachten zu können. Das image wird geöffnet und in rgb umgewandelt. Dies erleichtert die Vergleichbarkeit und das Resnet-Modell erwartet auch rgb-Bilder. Anschließend werden die Features für alle Modes erstellt und in einer Liste aus dictionaries gespeichert. Dafür werden die Funktionen, wie get_embedding, get_histogram, get_ahashes aus den dazugehörigen Dateien importiert. Wenn ein Checkpoint von z.B. 2000 Bildern oder das Ende erreicht ist, werden diese automatisch in dem data Folder abgespeichert, wo auch die Metadaten und die Database liegt.

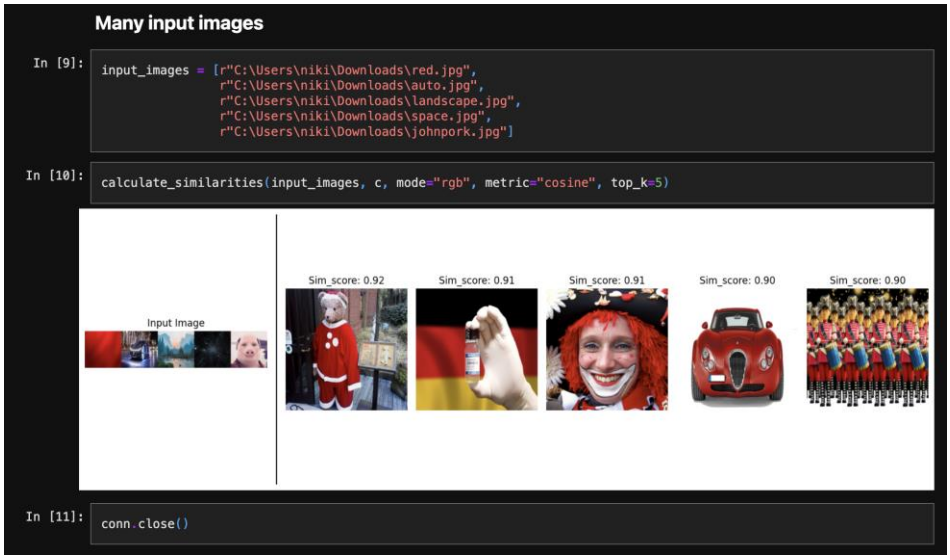
Die Datei embeddings.py enthält den Code zur Extraktion von Bildmerkmalen, den Embeddings, aus den Bildern. Hier wird ein vortrainiertes Deep Learning-Modell, das ResNet18-Modell von torchvision, verwendet, um Merkmalsvektoren für jedes Bild zu berechnen. Die Funktion “get_embedding(image, model, device)” nimmt ein Bild, verarbeitet es entsprechend und verwendet das ResNet-Modell, um einen Merkmalsvektor zu extrahieren. Dies dient als eine der Hauptmethoden zur Ähnlichkeitsberechnung.

Für die Analyse von Farbverteilungen enthält die Datei histograms.py Funktionen zur Berechnung von Farb-Histogrammen von Bildern. Die Funktion “get_histogram(image, bins)” berechnet und normalisiert das Histogramm des Bildes auf Basis von RGB-Kanälen. Histogramme werden oft als nützliche Methode für den Bildvergleich verwendet, insbesondere bei der Suche nach ähnlichen Bildern basierend auf ihren Farben.

Das Modul hashes.py implementiert zwei gängige Algorithmen zur Generierung von perzeptuellen Hashes (A-Hash und D-Hash), die eine schnelle Methode zur Bildähnlichkeitsbestimmung bieten. Hier werden die Funktionen “get_ahash(image)”

und “get_dhash(image)” definiert, die jeweils ein Bild in Graustufen umwandeln und dann entweder den Durchschnitts-Hash oder den Differenz-Hash berechnen.

Wenn man die Features geladen hat, kann man das Jupyter-Notebook show_similarities.ipynb ausführen, welches als Benutzeroberfläche für die Ausführung der Ähnlichkeitsberechnungen dient. Hier können die Benutzer*innen eine oder mehrere Bilddateien in einer Liste angeben und die calculate_similarities-Funktion ausführen, um die ähnlichsten Bilder zu ermitteln und visuell darzustellen. Zu Beginn des Notebooks wird die Verbindung zur Datenbank hergestellt und ein Cursor-Objekt übergeben in die Funktion übergeben. Es werden verschiedene Modus- und Metrik-Kombinationen angeboten, um unterschiedliche Arten von Bildvergleichen durchzuführen. Das Notebook erlaubt es auch, die Ergebnisse zu visualisieren, indem es die Eingabebilder zusammen mit den ähnlichsten Bildern anzeigt:



Ausschnitt des Jupyter Notebooks mit der Ausgabe der Ergebnisse

Die Datei similarities.py enthält die Hauptlogik zur Berechnung der Ähnlichkeit zwischen einem Eingabebild und allen bisher geladenen Bildern in unserer Datenbank. Die Funktion “calculate_similarities(input_images, cursor, mode, metric, top_k, verbose)” übernimmt die Aufgabe, die Ähnlichkeit zwischen den Eingabebildern und den Bildern in der Datenbank zu berechnen. Dabei kann der Benutzer den Modus der Ähnlichkeitsberechnung auswählen – etwa basierend auf “rgb”, “color” oder “ahashes”. Zudem kann eine der verschiedenen Ähnlichkeitsmetriken wie “cosine”, “euclidean”, “manhattan” und “hamming” gewählt werden. Die Funktion gibt die Top-K ähnlichsten Bilder zurück und visualisiert diese. Top_k bestimmt wie viele ähnliche Bilder ausgegeben werden sollen und der Parameter verbose lässt die execution-time anzeigen. In dieser Funktion werden die input_embeddings des Bildes berechnet und dafür werden die jeweiligen Methoden aus hashes.py, embeddings.py und histograms.py importiert. Dazu gibt es dort die Methode load_feature_data(), welche die

Daten nach einmaligen laden in einer globalen Variable speichert und plot_similar_images() zum Plotten der Bilder.

In Tensorboard_preperation.py und umap_embeddings.py können auf Basis der feature_data.pkl Visualisierungen erstellt werden. Man muss nur den jeweiligen Mode eingeben. Dafür werden libraries wie tensorboard, tensorflow und umap benutzt. Das Laden des Tensorboards kann lange dauern, da zuerst ein Sprite-Image aus den Daten erstellt wird.

3. Ähnlichkeitsmaße

Um die Ähnlichkeit zwischen unseren Eingabe- und Datenbankbildern zu messen, haben wir drei verschiedene Methoden implementiert: Neural-Network embeddings, Average-Hashes, sowie RGB-Farbhistogramme. Jede dieser Methoden analysiert unterschiedliche visuelle Merkmale eines Bildes und ermöglicht so unterschiedliche Ergebnisse bei der Ähnlichkeitsbestimmung. Die berechneten Merkmale wurden anschließend mithilfe verschiedener Distanz- oder Ähnlichkeitsmaßen (cosinus, euclidean,manhattan, hamming) verglichen, um die top_k ähnlichsten Bilder zu einem oder mehreren gegebenen Eingabebildern zu bestimmen. Im Mainloop werden die Bilder mit der Pillow (PIL) library geöffnet und in RGB umgewandelt, falls sie es noch nicht sind. Es gibt für jede Extraktion eine separate Methode für das input-image, wo diese Vorbereitungen ebenfalls getroffen werden.

3.1 Embeddings

Die erste Methode basiert auf den Embeddings eines Bildes, die mit einem vortrainierten ResNet-18-Modell extrahiert wurden. Dabei wird das Bild in einen hochdimensionalen NumPy-Vektor umgewandelt, der die komplexen visuellen Merkmale enthält. So können beispielsweise ähnliche Motive oder Objekte unter den Bildern erkannt werden.

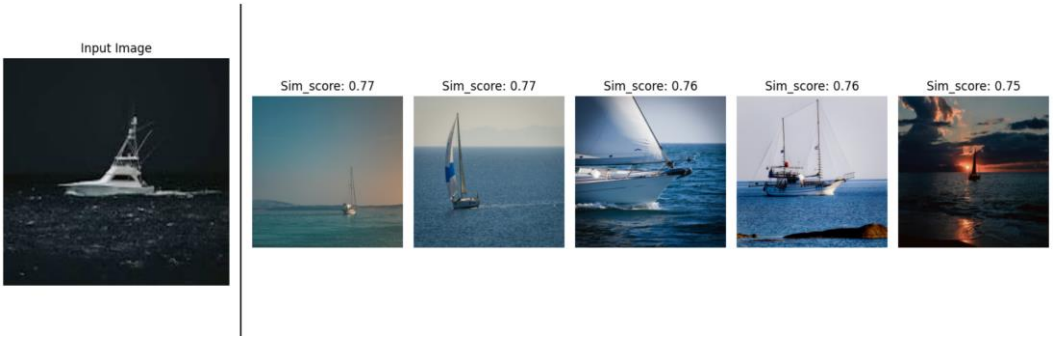
Die Embeddings werden mit ResNet-18 erzeugt, wobei das Bild zunächst mit preprocess() vorbereitet wird, sodass es den Anforderungen des Modells entspricht. Dazu gehören Skalierung, Center Crop, Umwandlung in einen Tensor und Normalisierung, basierend auf den Standardwerten von ResNet. Das vorbereitete Bild wird dann an get_embedding() übergeben, wo es mit dem Modell verarbeitet wird. Das Modell wird einmalig im Main Loop geladen und auf das verfügbare Gerät (GPU oder CPU) verschoben. Die letzte Fully-Connected-Schicht wurde durch eine Identity-Funktion ersetzt, sodass der Ausgabevektor die rohen Embeddings enthält. Die Berechnung erfolgt im Evaluierungsmodus (eval()) und innerhalb eines torch.no_grad()-Blocks, um Speicher zu sparen und unnötige Gradientenberechnungen zu vermeiden. Das

ausgegebene Embedding ist ein 512-dimensionaler NumPy-Vektor, der anschließend zur Ähnlichkeitsberechnung genutzt werden kann.

Der Vorteil dieser Methode liegt in ihrer Fähigkeit, tiefgehende und zusammenhängende Informationen zu erfassen, wodurch sie weniger anfällig für Veränderungen in Perspektive, Beleuchtung oder Skalierung ist. Besonders Objekte und Motive werden sehr gut erkannt.

Ein Nachteil ist der hohe Rechenaufwand sowie das Blackbox-Phänomen: Es ist nicht vollständig interpretierbar, welche Informationen das Modell genau speichert.

Trotz dieser Einschränkungen kommt diese Methode unserer menschlichen Interpretation von Ähnlichkeit am nächsten und liefert in den meisten Fällen die besten Ergebnisse. So wird zum Beispiel das Boot richtig als Boot erkannt.



3.2 RGB-Histogramme

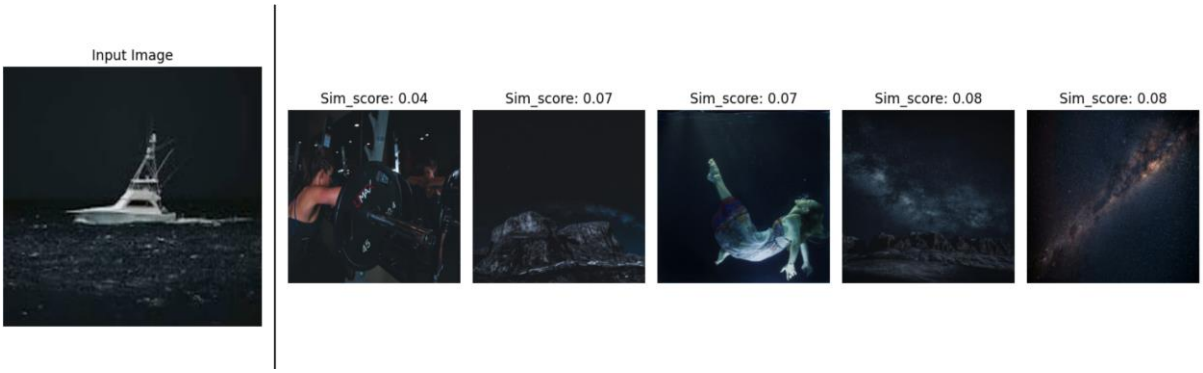
Zusätzlich haben wir Farb-Histogramme berechnet, um die Verteilung der Farben innerhalb eines Bildes zu analysieren – eine der beliebtesten Methoden, um die Ähnlichkeiten zwischen Bildern darzustellen. Dabei wird ein dreidimensionales Histogramm im RGB-Farbraum erstellt und als numpy-array gespeichert.

Die Histogramme werden mithilfe der OpenCV-Funktion “cv2.calcHist()” berechnet, wobei die Farbwerte in vordefinierte Bins unterteilt werden. Wir haben 8 Bins pro Farbkanal gewählt, was zu einer Gesamtanzahl von 8^3 = 512 Dimensionen führt. Anschließend wird das Histogramm mit “cv2.normalize()” mithilfe der L1-Normalisierung skaliert, um sicherzustellen, dass Bilder unabhängig von ihrer Größe miteinander vergleichbar bleiben. Abschließend wird das Histogramm mit “flatten()” in eine eindimensionale Form gebracht, um es weiterverarbeiten zu können.

Die Farbmethode ist besonders nützlich, wenn Farbähnlichkeiten ein entscheidender Faktor sind, da sie unempfindlich gegenüber anderen Lichtverhältnissen und kleinen Verzerrungen auf Bildern ist. Allerdings kann sie keine strukturellen oder objektbasierten Ähnlichkeiten erfassen wie z.B. die Embeddings-Methode und ist daher für inhaltsbasierte Vergleiche weniger geeignet. Dadurch kann es vorkommen, dass zwei

Bilder als ähnlich eingestuft werden, obwohl sie völlig unterschiedliche Inhalte zeigen, weil ihre Farbverteilungen übereinstimmen.

Diese Methode eignet sich daher eher für Farbvergleiche oder Stimmungsanalysen als für inhaltsbasierte Bildvergleiche. In dem Beispiel ist die Stimmung bei allen Bildern etwas dunkel und düster, aber es werden unterschiedliche Dinge gezeigt – es eignet sich dennoch als gutes Similarity-measure. Diese Methode ist ein effektives Maß für Ähnlichkeiten, wenn die Farbstimmung eines Bildes die zentrale Rolle spielen soll.



3.3 Hashes

Ein weiterer Ansatz sind unsere Image-Hashes, bei denen wir A-Hash (Average Hash) und D-Hash (Difference Hash) verwendet haben. Letztlich haben wir uns aber für den Average-Hash entschieden. Diese Methoden erzeugen kompakte Binärwerte die als Fingerabdruck eines Bildes dienen und somit Vergleiche ermöglichen. Beim Hashing wird das Bild in Graustufen umgewandelt, verkleinert und mithilfe der imagehash-library der Hash ausgerechnet.

Beim Average Hash basiert dieser auf dem Durchschnittswert der Pixel-Intensitäten und jeder Pixel wird mit diesem Wert verglichen. Ist der Pixelwert kleiner wird eine 0 gesetzt, ist der Pixelwert größer, eine 1.

Ein Vorteil ist die einfache und effiziente Berechnung. Man kann zudem auch erkennen, wo die Einschätzungen herkommen. Es ist also gut dafür ähnliche Helligkeitsverlaufe zu erkennen. Das kann aber je nach Bild für uns Menschen nicht ähnlich aussehen. Es werden keine wirklichen Inhalte oder Farben beachtet, weswegen es nicht sehr robust und gut ist. Die Ähnlichkeit wird hier meistens durch die sogenannte Hamming-Distanz zwischen den Hash-Werten bestimmt, da es sich um Binärwerte handelt.

In dem Beispiel wird auch der Helligkeitsverlauf gut erkannt, aber die Bilder sind ziemlich verschieden und habe teilweise auch verschiedene Farben.

Der Difference-Hash basiert auf Intensitätsunterschieden zwischen benachbarten Pixeln und ist dadurch empfindlicher gegenüber Strukturänderungen. Dieser hat aber noch schlechtere und schwer interpretierbare Ergebnisse geliefert, weswegen wir und für den Average-Hash entschieden haben.

Nachdem die Merkmale für ein Eingabebild extrahiert wurden, haben wir verschiedene Metriken zur Ähnlichkeitsberechnung verwendet. Die meisten funktionieren für alles, aber eignen sich besser für bestimmte Analysen.

- Die “Kosinussimilarität” eignet sich gut für Embeddings, da sie misst, wie ähnlich zwei Vektoren in Bezug auf ihre Richtung sind.

- Euklidische Distanz eignet sich gut, um Unterschiede zwischen Histogrammen oder Embeddings zu berechnen.

- Hamming-Distanz wurde für Hash-Werte genutzt, da sie misst, wie viele Bits sich zwischen zwei Hashes unterscheiden.

Anhand dieser Methoden und Metriken haben wir für jede Eingabebilder die Top-k ähnlichsten Bilder aus unserer Datenbank (ca. 450.000 Bilder) ermittelt. Die Ergebnisse werden dann visuell dargestellt, indem die Eingabebilder sowie die ähnlichsten Treffer nebeneinander ausgegeben werden (siehe “show_similaritites.ipynb”).

Zusammenfassend lässt sich sagen, dass jede der verwendeten Methoden ihre eigenen Stärken und Schwächen hat. Embeddings und Histogramme sind eher stärker, A-Hashes performen durchschittlich und D-Hashes sind noch schwerer zu interpretieren - während Embeddings am besten für komplexe Muster- und thematische Objektvergleiche geeignet sind. Farb-Histogramme wiederum sind besonders geeignet, wenn die Farbverteilungen im Vordergrund stehen. In vielen Anwendungsfällen könnte eine Kombination mehrerer dieser Methoden zu präziseren, somit zu besseren Ergebnissen führen.

4. Analyse der Performance und Optimierung

Die Pipeline lässt sich im Wesentlichen in 2 Schritte aufteilen. Die Feature-Generierung und die Similarity-Search. Bei dem zweiten Punkt ist eine schnelle Ausgabe von besonderer Bedeutung. Wir starten mit der Feature Generierung.

Dort musste man sich darauf einstellen, dass der gesamte Durchlauf mindestens mehrere Stunden in Anspruch nimmt. Bei uns waren es circa 20 Stunden für die gesamte Pipeline. Das ist auf die Größe der Daten und Berechnung der Features zurückzuführen. Somit betraf der erste Optimierungsschritt die Generierung der Feature-Vektoren. Ursprünglich hatten wir für jede Ähnlichkeitsmetrik eine eigene Hauptschleife, wobei die Daten jeweils in separaten Pickle-Dateien gespeichert wurden. Dies führte dazu, dass jedes Bild für jede Metrik mehrfach geöffnet und verarbeitet werden musste. Um diesen Overhead zu reduzieren, haben wir einen zentralen Main Loop eingeführt, in dem das Bild einmal geöffnet und dann für alle Ähnlichkeitsberechnungen weitergegeben wird. Zuvor wurde das Bild für jede Metrik separat geöffnet, was unnötige Redundanz erzeugte. Jetzt erfolgt das Laden und die Konvertierung in RGB direkt im Main Loop, sodass dieser Schritt nicht für jede Methode erneut ausgeführt werden muss.

Bei der Extraktion der Embeddings haben wir ebenfalls einige Optimierungen vorgenommen, da diese sehr rechenintesiv ist. Zunächst haben wir das ResNet50-Modell verwendet, uns dann aber für ResNet18 entschieden. Dies führte zu einer schnelleren Berechnung der Features, während die Qualität der Ergebnisse weiterhin überzeugend blieb. Zusätzlich wird das vortrainierte ResNet18-Modell samt Gewichten nur einmal im Main Loop geladen, anstatt es bei jedem einzelnen Aufruf von `get_embedding()` neu zu initialisieren. Außerdem ist die Berechnung auf der GPU möglich, wenn entsprechende Hardware vorhanden ist, was die Verarbeitungsgeschwindigkeit weiter verbessert.

Der Speicherplatz wurde für alle Features optimiert, indem wir den Datentyp vorgegeben haben von z.b. float64 auf float32. Des weiteren haben wir mit Checkpoints gearbeitet, damit bei möglichem Abbruch des main-loops, von dem jeweiligen Speicherpunkt weitergeladen werden konnte. Wo es ging, haben wir effiziente Bibliotheken, wie numpy, cv2 oder scikit-learn oder pickle benutzt.

Beim Generator haben wir die Size und andere Metadaten entfernt, da wir diese nicht weiter benutzt haben. Somit lädt dieser die image_id, root, file in wenigen Sekunden. Und wenige Sekunden später ist die Database erstellt.

Jetzt kommen wir zum zweiten und wichtigeren Teil in Bezug auf runtime. Es zählt die Geschwindigkeit, mit der nach Eingabe eines Bildes die ähnlichsten Bilder ausgegeben

werden. Wir nutzen vektorisierte NumPy-Berechnungen, um die Similarity-Werte zu bestimmen, also die Distanzen und die Kosinus-Ähnlichkeit. Dies ist vergleichweise schnell.

Zunächst haben wir alle Einträge mit `np.argsort` sortiert, was eine Laufzeit von O(n log n) hat. Bei immer größeren Datensätzen kann das jedoch lange dauern. Deshalb haben wir stattdessen `np.argpartition` verwendet, um nur die `top_k` Werte ohne vollständige Sortierung zu finden, was eine Laufzeit von O(n) hat. Anschließend werden die `top_k` Werte sortiert, was insgesamt einer Laufzeit von o(n) + O(k log n) anstelle von O(n log n) entspricht. Diese Änderung hat die Performance der Ähnlichkeitssuche verbessert.

Nachdem wir die Laufzeiten analysiert hatten, fiel uns auf, dass der größte Performance-Verlust durch das Laden der Pickle-Datei verursacht wurde. Um dies zu optimieren, wird der Inhalt der Pickle-Datei nach dem ersten Aufruf von `calculate_similarities()` in einer globalen Variable gespeichert. Dadurch entfällt das erneute Laden der Datei, was zu einer drastischen Geschwindigkeitssteigerung geführt hat. Damit waren wir dann bei einer Ausgabezeit von 2-3 Sekunden mit Plotten des Bildes nach dem ersten Aufruf. Das kann je nach Größe des Bildes leicht variieren.

```
Time to load feature data: 4.7782 second
Time to extract features: 0.6981 seconds
Time to compute similarities: 0.6622 seconds
Time to retrieve top-K images: 0.0000 seconds
Total execution time: 6.1406 seconds
```

```
Time to load feature data: 0.0000 second
Time to extract features: 0.6493 seconds
Time to compute similarities: 0.6413 seconds
Time to retrieve top-K images: 0.0010 seconds
Total execution time: 1.2935 seconds
```

Laufzeit einer Similarity Ausgabe mit (links) und ohne (rechts) laden des Datensatzes für ein Embedding

Unser gesamter Workflow im Jupyter-Notebook für die top-5 similar images dauert somit circa 15 Sekunden, wobei dort auch das Laden der pickle-file einmal eingebunden ist. Damit sind wir zufrieden.

Run	Load Feature Data (s)	Extract Features (s)	Compute Similarities (s)	Retrieve Top-K (s)	Complete Similarity Retrieval (s)	Plot Time (s)	Total Time (s)
Embeddings (Cosine Similarity)	4.5129	0.6672	0.6253	0.001	5.8065	1.2646	7.07
Color Histograms (Euclidean Distance)	0.0	0.4328	0.5585	0.001	0.9933	1.29	2.28
A-Hashes (Hamming Distance)	0.0	0.3421	0.8088	0.001	1.1529	1.0173	2.17
Embeddings (Cosine Similarity) - 5 Input Imag	0.0	1.506	0.6443	0.001	2.1522	0.8796	3.03
Total Time for Whole Run	nan	nan	nan	nan	nan	nan	14.8

Bei der weiteren Recherche sind wir auf ANNs (Approximate Nearest Neighbors) wie FAISS gestoßen, die die Laufzeit weiter verbessern könnten. FAISS indexiert die Vektoren, sodass die Ähnlichkeitssuche schneller erfolgt, allerdings mit einem kleinen

Genauigkeitsverlust. Je größer der Datensatz ist, desto eher lohnt sich der Einsatz solcher Algorithmen. Da unsere Laufzeit bereits im Bereich von wenigen Sekunden lag, haben wir jedoch darauf verzichtet.

Zusammenfassend haben diese Optimierungen dazu geführt, dass die Ähnlichkeitssuche wesentlich schneller ist. Sollte der Datensatz weiter anwachsen, wäre es sinnvoll, Methoden wie FAISS zu testen oder eine effizientere Speicherung in mehreren Pickle-Dateien auszuprobieren. Des weiteren kann man über Batches oder Parallelisierung nachdenken.

5. “Feasibility” Analyse / Diskussion

Für die gegebenen Daten ist unsere Pipeline grundsätzlich machbar und arbeitet effizient. Allerdings nimmt die Extraktion der Bildmerkmale eine gewisse Zeit in Anspruch. Ein wesentlicher Bottleneck ist das Laden der Bilder von der SSD. Besonders große Bilder, die aus dem Bereich zwischen 50 % und 75 % der verarbeiteten Daten stammen, verlangsamen den Prozess erheblich. Eine mögliche Lösung wäre, die Bilder bereits in der Pipeline zu verkleinern, doch dies würde die Verarbeitung zusätzlich verlangsamen.

Ein weiterer limitierender Faktor ist der Arbeitsspeicher (RAM). Momentan werden alle extrahierten Features in einer einzigen Pickle-Datei gespeichert und beim Abruf komplett in den Speicher geladen. Aktuell ist dies mit einer Größe von 2,6 GB noch machbar. Mit steigendem RAM könnten wenige Millionen Bilder verarbeitet werden. Selbst wenn der Speicher voll wird, hat man zumindest die Checkpoint-gespeicherte pickle Datei, mit welcher man weiterarbeiten kann.

Um Speicherplatz zu sparen, haben wir die Datentypen der Feature-Vektoren optimiert, sodass nur der notwendige Speicher belegt wird. Sollte der Datensatz jedoch weiterwachsen, wäre es notwendig, entweder externe Rechenressourcen zu nutzen oder die Pipeline so zu modifizieren, dass die Daten in mehreren Pickle-Dateien mit einer festgelegten Kapazitätsgrenze gespeichert werden. Sobald diese Grenze erreicht wird, könnte eine neue Datei erstellt werden. Allerdings würde dies die Ähnlichkeitssuche etwas komplizierter machen.

Eine weitere Möglichkeit wäre, die Daten schrittweise in Batches aus der Pickle-Datei zu laden, anstatt sie vollständig in den Speicher zu laden. Dadurch würde der RAM-Bedarf reduziert, allerdings könnte sich die Geschwindigkeit der Ähnlichkeitssuche verschlechtern, wenn Millionen von Bildern verarbeitet werden müssen.

Wie genau die Performance bei riesigen Datensätzen ausfallen würde, können wir nicht mit Sicherheit sagen. Allerdings gibt es mehrere Ansätze, um die Effizienz weiter zu

verbessern. Eine Möglichkeit wäre der Einsatz von GPUs, um die Berechnungen zu beschleunigen. Besonders bei der Extraktion der Embeddings könnte dies einen Geschwindigkeitsvorteil bringen, wenn entsprechende Hardware verfügbar ist. Zudem könnten Approximate Nearest Neighbor (ANN)-Methoden wie FAISS verwendet werden, um die Ähnlichkeitssuche in großen Datensätzen zu optimieren. Diese Methoden erlauben eine wesentlich schnellere Ausgabe ähnlicher Bilder, da sie auf effiziente Vektorindizierung setzen. Zwar gibt es dabei einen kleinen Genauigkeitsverlust, doch je größer der Datensatz wird, desto mehr würde sich der Einsatz von ANN-Methoden lohnen

Trotz dieser Herausforderungen ist die Pipeline für die aktuelle Datenmenge solide und leistungsfähig. Für kleinere bis mittelgroße Datensätze mit bis zu wenigen Millionen Bildern bleibt die Software noch performant, doch bei noch größeren Datenmengen stößt sie an ihre Grenzen. Eine Skalierung auf das Niveau großer Bildsuchmaschinen wie Google Image Search wäre mit der aktuellen Struktur nicht möglich und würde entweder eine bessere inhaltliche Organisation der Daten oder den Zugriff auf externe Rechenressourcen erfordern.

Zusammenfassend ist die Software optimal für kleinere bis mittelgroße Bilddatenbanken, erreicht jedoch bei sehr großen Datenmengen ihre technischen Grenzen. Für eine Echtzeit-Suchanwendung mit Millionen von Bildern wären zusätzliche Optimierungen notwendig, insbesondere durch GPU-Beschleunigung oder Approximate Nearest Neighbor (ANN)-Methoden wie FAISS, um die Ähnlichkeitssuche effizienter zu gestalten

6. Bilderanalyse

Für die Analyse der Bilddaten haben wir UMAP zur Dimensionsreduktion und KMeans zum Clustering verwendet. Da es bei diesem Datensatz schwierig ist, eine optimale Anzahl an Clustern festzulegen, haben wir die Elbow-Methode (Ellbogenanalyse) angewandt. Dabei sucht man den Punkt, an dem die Reduktion der Clustering-Fehlerrate nicht mehr signifikant abnimmt – also dort, wo sich der "Ellbogen" knickt. Dieser Wert ist zwar nicht perfekt, ermöglicht aber eine sinnvolle Einordnung.

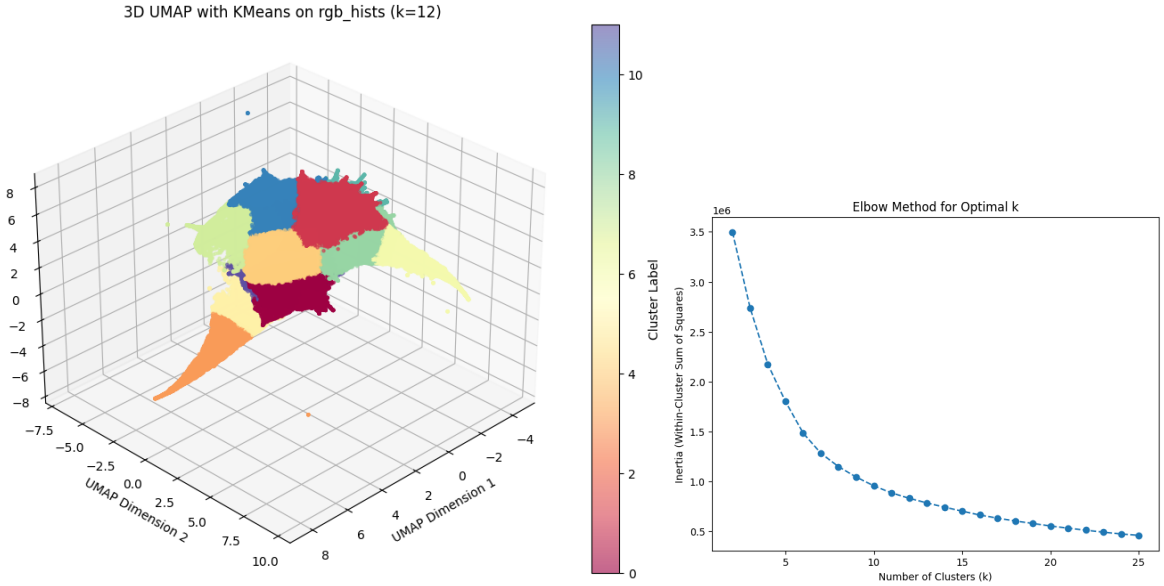
Wir haben uns für UMAP entschieden, da es deutlich schneller als t-SNE ist und dennoch qualitativ hochwertige Resultate liefert. Die Dimensionen wurden auf drei reduziert, um eine 3D-Visualisierung zu ermöglichen. Diese Analyse wurde für RGB-Histogramme sowie für Embeddings durchgeführt.

6.1 Analyse der RGB-Histogramme

Bei den Farbverteilungen fällt auf, dass sich ein großes Cluster in der Mitte der Darstellung bildet. Das deutet darauf hin, dass viele Bilder ähnliche Farbprofile besitzen und entsprechend nahe beieinander liegen. Auffällige Ausreißer sind kaum vorhanden.

Besonders interessant sind die scharfen Spitzen in der Verteilung, die darauf hinweisen, dass bestimmte Bilder besser vom Hauptcluster differenziert werden können. Wahrscheinlich befinden sich am Rand der Darstellung Bilder mit sehr eindeutigen Farbprofilen, z. B. komplett weiße, schwarze oder farblich stark gesättigte Bilder. Das zeigt sich auch daran, dass orangefarbene und gelbe Cluster weit voneinander entfernt sind, was auf klare Unterschiede in ihren Farbkompositionen hindeutet.

Eine mögliche Weiterführung wäre die Entwicklung eines Labels, das Bilder basierend auf ihren Farbprofilen klassifiziert.



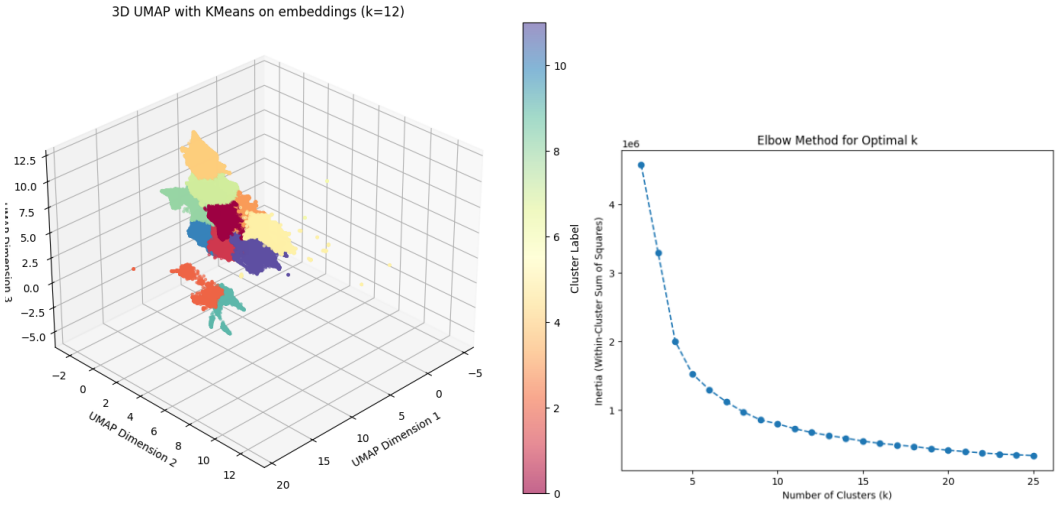
6.2 Analyse der Embeddings

Im Gegensatz zu den RGB-Histogrammen ist die Struktur der Embeddings deutlich differenzierter. Statt eines einzelnen großen Clusters sehen wir klarere Grenzen und erkennbare Substrukturen. Das deutet darauf hin, dass die Embeddings eine stärkere Trennbarkeit der Bilder ermöglichen.

Diese Beobachtung ist sinnvoll, da Embeddings nicht nur Farbverteilungen, sondern komplexere Merkmale wie Formen, Texturen und Objekte erfassen. Dadurch entstehen klarere Cluster-Grenzen, während es bei Farbprofilen oft fließende Übergänge gibt.

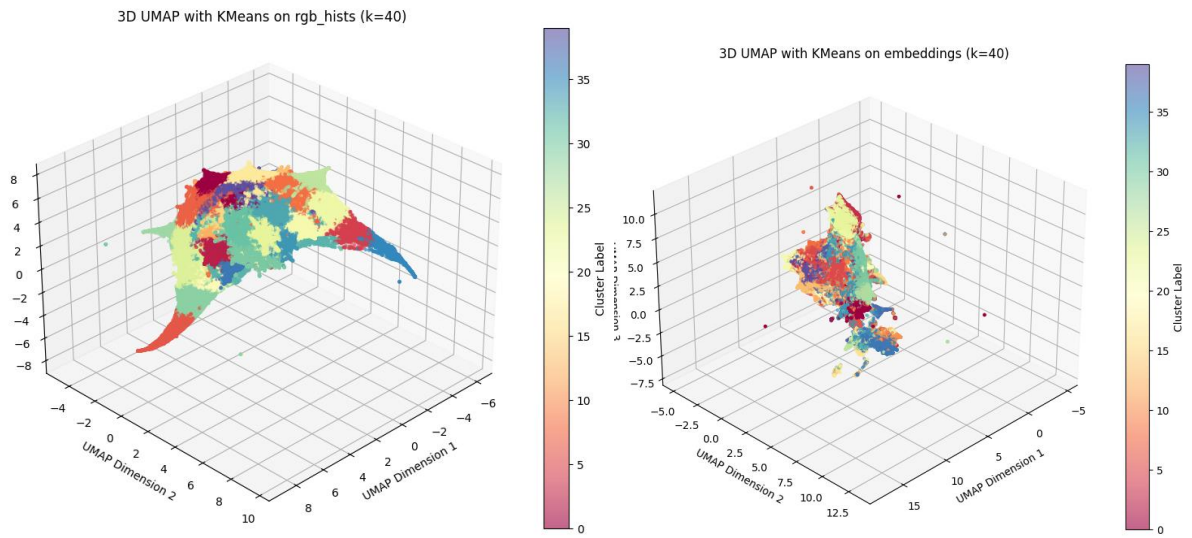
Obwohl sich auch hier eine größere Cluster-Konzentration in der Mitte zeigt, sind einige klar getrennte Gruppen sichtbar. Ein interessanter Punkt ist der abgeschnittene Cluster am unteren Rand – dies könnten möglicherweise Bilder von Gesichtern sein, die sich stark von anderen Bildkategorien unterscheiden.

Diese Beobachtungen deuten darauf hin, dass Embeddings eine bessere Differenzierung zwischen Bildern ermöglichen als Farbprofile.



6.3 Clustern im hoherdimensionalen Featurespace

Ein weiterer Schritt war, die Cluster bereits im hochdimensionalen Raum zu bilden und diese als Labels für den reduzierten Raum zu verwenden. Dadurch könnte überprüft werden, inwiefern die Struktur und Trennbarkeit der Embeddings nach der Dimensionsreduktion erhalten bleiben. Das wirkt für die Embeddings und Histogramme insgesamt wilder. Besonders bei den Embeddings sind einige Werte etwas off und es gibt mehr Ausreiser. Das kann an dem Informationsverlust der Dimensionsreduzierung liegen. Man erkennt, dass im Großen und Ganzen die Cluster trotzdem korrekt zugordnet werden, aber die Trennungen etwas schwammiger sind. Einige Punkte überschreiten ihre Clustergrenzen, aber im Wesentlichen werden diese gut erkannt. Die generelle Struktur ist aber gleichgeblieben.



6.4 TensorBoard-Analyse

Zusätzlich haben wir eine Visualisierung im TensorBoard durchgeführt, um die Bilder direkt in ihrer reduzierten Form betrachten zu können. Allerdings gibt es hier Beschränkungen:

- PCA ist auf 100.000 Bilder limitiert.
- UMAP kann nur mit 5.000 Bildern dargestellt werden.

Dennoch ermöglicht diese Methode eine schnelle visuelle Analyse. Beim Betrachten der ersten 5.000 Bilder mit UMAP ergeben sich einige interessante Muster:

- Farbprofile sind klar gruppiert, was darauf hindeutet, dass Bilder mit ähnlichen Farben oft nahe beieinander liegen.
- Bei den Embeddings fällt auf, dass Gesichter besonders gut erkannt und getrennt werden. Die Trennlinien zwischen Gesichtsbildern und anderen Kategorien sind deutlich erkennbar.
- Farben verlaufen teilweise ineinander, was eine klare Cluster-Trennung erschwert. Dies erklärt auch einige der zuvor beobachteten Unschärfen in der 3D-Visualisierung.

RGB-Histogramme und Network Embeddings in Tensorboard

Fazit:

- Die Embeddings liefern insgesamt bessere Trennungsergebnisse als RGB-Histogramme.
- Die Farbverläufe sind gut dargestellt, aber Clustergrenzen sind nicht immer scharf.
- TensorBoard ist für eine schnelle Einsicht nützlich, aber aufgrund der Datenbeschränkung nicht für eine vollständige Analyse geeignet.