Maurice Füsser und
Nikolas Predescu

15.08.2024

Projekt-Dokumentation: Image Recommender

1. Motivation und Zielsetzung des Projekts

Das Ziel dieses Projekts war es, eine leistungsfähige Software zu entwickeln, die basierend auf verschiedenen Ähnlichkeitsmaßen ähnliche Bilder zu einem gegebenen Bild ausgibt. Der Datensatz enthält über 400.000 Bildern. Die Software sollte die besten fünf Übereinstimmungen für ein einzelnes Bild sowie für bis zu fünf Eingabebilder finden und dabei eine hohe Effizienz und Genauigkeit gewährleisten. Außerdem wollen wir anhand von unterschiedlichen Methoden den Datensatz visualisieren, um weitere Erkenntnisse zu gewinnen. Die Visualisierung erfolgt in 2D oder 3D, da wir eine gut geeignete Bibliothek gefunden haben.

2. Programmdesign

Unser Programm besteht aus mehreren wesentlichen Komponenten:

- **Daten-Handling und Generator:**
  - **generator.py**: Dieses Modul enthält einen Generator, der die Bilder lädt und ihnen eine eindeutige Bild-ID zuweist. Es bildet die Basis für das Laden und Verarbeiten der großen Bildmengen und sorgt für eine effiziente Speicherverwaltung.
    Die Datenbank wird verwendet, um Bild-IDs, Dateinamen, Speicherorte und relevante Metadaten zu verknüpfen. Die Struktur ermöglicht eine schnelle und effiziente Bildabfrage. Wir haben uns für eine SQLite Datenbank entschieden, da diese am simpelsten zu implementieren war und für unsere Zwecke vollkommen ausreicht.
    (Wir haben die Datenbank für 2 von 3 Ähnlichkeitsmaßen verwendet, um zu testen, ob es Auswirkungen auf die Rechenzeit hat)

- **Ähnlichkeitsmaße:**
  - **extract_rgb.py**: Zunächst werden die Pfade der zu verarbeitenden Bilder aus einer SQLite-Datenbank geladen. Diese Datenbank enthält eine Tabelle, die die Dateipfade der Bilder speichert.
    Für jedes Bild wird das RGB-Histogramm extrahiert. Hierbei wird das Bild zunächst in den RGB-Farbraum konvertiert und auf eine feste Größe (224x224 Pixel) skaliert. Anschließend wird das Histogramm berechnet und normalisiert.

Wir verwenden Batches, wodurch regelmäßig der Fortschritt gespeichert wird, sodass der Prozess bei Unterbrechungen fortgesetzt werden kann. Die extrahierten Histogramme werden in einer Pickle-Datei gespeichert und können für Analysen oder zur Berechnung der Bildähnlichkeit verwendet werden.

- **extract_embedding.py**: Zunächst überprüfen wir, ob eine GPU verfügbar ist. Ist dies der Fall, wird das Modell und die Bilddaten auf die GPU geladen, um die Berechnungen zu beschleunigen.
  Ein vortrainiertes ResNet50-Modell wird geladen und die Klassifikationsschicht entfernt. Die Bilder werden in ein einheitliches Format konvertiert, das für das ResNet50-Modell geeignet ist. Das umfasst das Anpassen der Größe, das Zuschneiden auf die Mitte, die Umwandlung in Tensoren und die Normalisierung.
  Alle Bildpfade aus dem angegebenen Verzeichnis werden gesammelt. Es werden nur unterstützte Bildformate (.jpg / .jpeg / .png) verarbeitet.
  Die Bilder werden in Batches verarbeitet, um den Speicher weniger zu belasten. Für jedes Bild wird ein Embedding extrahiert, das die Merkmale des Bildes repräsentiert. Die Embeddings werden zusammen mit einer eindeutigen ID und dem Pfad des jeweiligen Bildes gespeichert. In regelmäßigen Abständen werden Checkpoints erstellt, um den Fortschritt zu sichern. (Hierbei werden .pkl erstellt die viel Speicher in Anspruch nehmen können).
  Die Hauptfunktion sammelt die Bildpfade, legt die Batch-Größe und das Intervall für die Checkpoints fest und startet die Generierung der Embeddings.
  Wir haben bei diesem Modul versucht die Pfade zu den Bildern in die .pkl Datei einzubetten. Der Gedanke war Rechenzeit zu sparen, da die Einträge nicht aus der Datenbank ausgelesen werden müssen. Im Endeffekt ist uns aufgefallen, dass dies keine Rechenzeit spart und in Zukunft eher zu Problemen führen kann, da die Pfade fest definiert sind. Wenn die Festplatte auf einem anderen Buchstaben (Pfad) definiert ist, werden die Bilder nicht gefunden. Wir haben die Festplatte als den Datenträger F:\ definiert.

- **Externe Module**
  - Wir haben einige externe Module verwendet, um die Rechenzeit als auch den Programmieraufwand zu beschleunigen. Wir verwenden folgende Module:
    Pillow (PIL):
    Bild öffnen, bearbeiten und konvertieren (z. B. Größenanpassung, Farbkonvertierung).

PyTorch (torch):
Implementierung und Ausführung neuronaler Netze.
Extraktion von Bild-Embeddings mit vortrainierten Modellen.

Torchvision:
Vortrainierte Modelle (z. B. ResNet50).
Nützliche Bildtransformationen (z. B. Zuschneiden, Normalisierung).

NumPy:
Numerische Berechnungen und Array-Manipulationen.

Scikit-learn:
Hauptkomponentenanalyse (PCA) zur Dimensionsreduktion.
Berechnung von Ähnlichkeitsmetriken (z. B. Cosine similarity).

SciPy:
Spezielle mathematische Operationen (z. B. Manhattan distance).

SQLite:
Verwaltung und Abfrage von Bild-Metadaten in einer relationalen Datenbank.

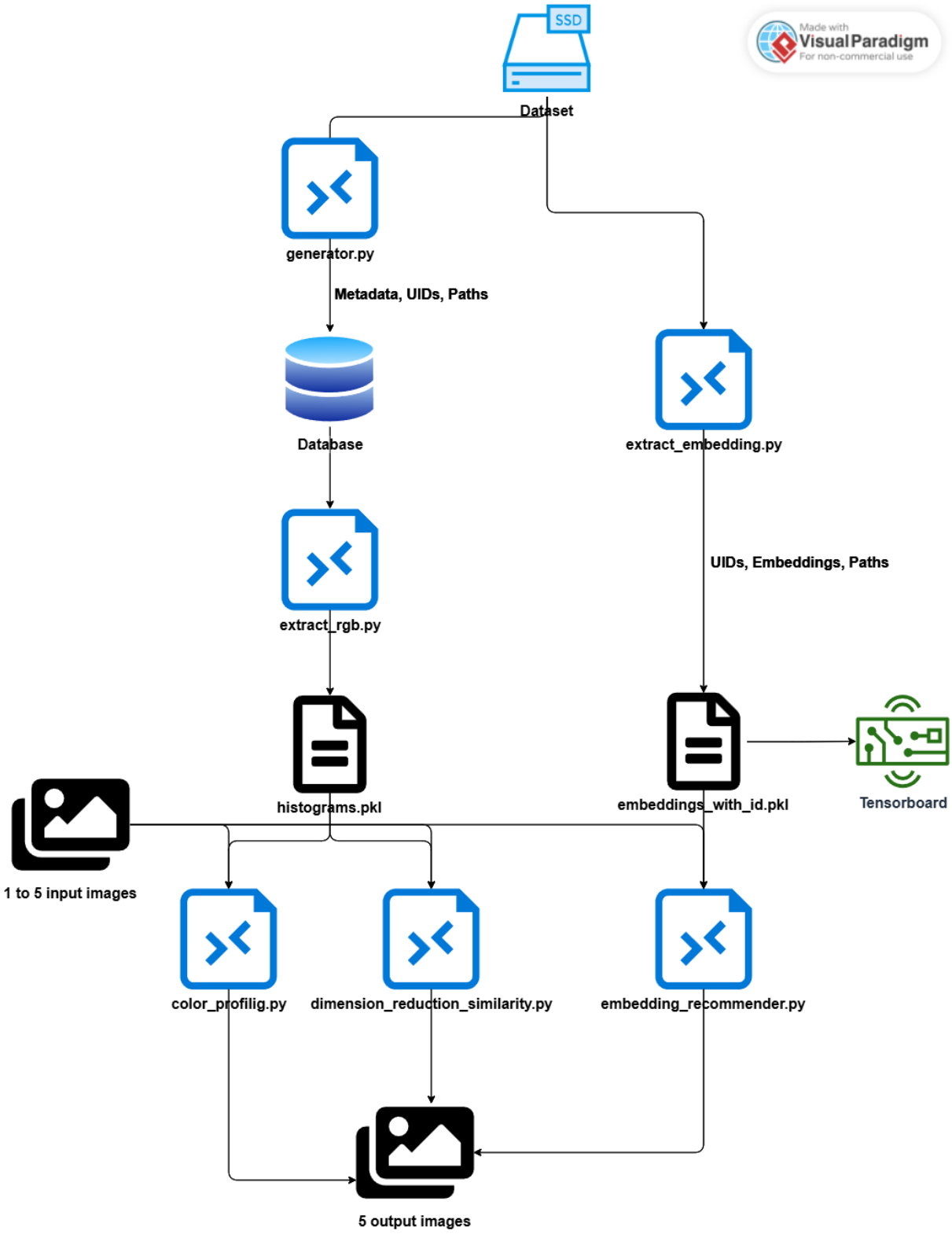
Matplotlib:
Grafische Darstellung von Bildern und Ähnlichkeitswerten.

tqdm:
Fortschrittsbalken für lange Prozesse. Nicht wirklich genau, aber besser als nichts.

os:
Dateisystemoperationen (z. B. Durchsuchen von Verzeichnissen oder Verarbeiten von Dateipfaden).

Pickle:
Speichern und Laden von Python-Objekten (z. B. Histogramme, Embeddings).
Binäres Dateiformat. Ist schnell und nice anzuwenden.

Programmskizze:



3. Bildähnlichkeit

Die Messung der Bildähnlichkeit wurde durch drei unterschiedliche Methoden realisiert:

Chi-Quadrat-Distanz (color_profiling.py)

- Eine unserer Methoden zur Berechnung der Ähnlichkeit zwischen Bildern basiert auf der Chi-Quadrat-Distanz. Diese Metrik wird verwendet, um den Unterschied zwischen den Farb-Histogrammen zweier Bilder zu vergleichen. Ein Histogramm repräsentiert die Verteilung der RGB-Farbwerte eines Bildes, und die Chi-Quadrat-Distanz misst, wie stark sich die Histogramme zweier Bilder unterscheiden. Die Funktion chi2_distance berechnet den Unterschied zwischen zwei Histogrammen histA und histB und gibt einen Distanzwert zurück. Ein kleinerer Wert deutet auf eine größere Ähnlichkeit hin.
  Jedes Bild wird in den RGB-Farbraum konvertiert und auf eine feste Größe skaliert (224x224 Pixel). Anschließend wird das Histogramm des Bildes extrahiert und normalisiert.
  Für ein einzelnes Eingabebild wird das Histogramm extrahiert und mit den Histogrammen aller anderen Bilder in der Datenbank verglichen. Die chi2_distance wird verwendet, um die Distanz zu berechnen, und die top_n ähnlichsten Bilder werden zurückgegeben.
  Wenn mehrere Eingabebilder vorhanden sind, wird für jedes Bild eine Chi-Quadrat-Distanz berechnet und die Distanzen über alle Eingabebilder hinweg gesammelt. Dadurch wird eine Liste von Bildern erstellt, die im Durchschnitt am ähnlichsten zu allen Eingabebildern sind.

Vorteile:

Diese Metrik ist besonders gut geeignet, um Unterschiede in der Farbverteilung von Bildern zu erfassen. Außerdem bekommen wir schnell die Ergebnisse zu den Inputbildern

Nachteile:

Diese Methode kann bei Bildern, die sehr ähnliche Histogramme haben, jedoch unterschiedliche visuelle Inhalte aufweisen, zu ungenauen Ergebnissen führen.

Embedding Ähnlichkeitsmaß (embedding_recommender.py)

- Zunächst wird für jedes Bild ein Bild-Embedding extrahiert, dass die hochdimensionalen Merkmale des Bildes repräsentiert. Dies geschieht mithilfe eines vortrainierten ResNet50-Modells, bei dem die Klassifikationsschicht entfernt wurde, um nur die Embeddings zu extrahieren. Für ein gegebenes Eingabebild wird das Embedding extrahiert und anschließend die Ähnlichkeit zwischen diesem Embedding und den Embeddings einer Datenbank berechnet. Dabei werden drei verschiedene Metriken verwendet:
  - **Kosinus-Ähnlichkeit**: Diese Metrik misst den Winkel zwischen den Vektoren der Embeddings und ist besonders nützlich, um die Richtung der Vektoren zu vergleichen, unabhängig von ihrer Länge.
  - **Euklidische Distanz**: Diese Metrik misst die „geradlinige" Entfernung zwischen den Vektoren im Embeddingraum. Sie ist empfindlich gegenüber Unterschieden in der gesamten Größe der Embeddings.
  - **Manhattan-Distanz** (Cityblock-Distanz): Diese Metrik misst die Summe der absoluten Unterschiede entlang jeder Dimension. Sie ist robuster gegenüber Ausreißern als die euklidische Distanz.

  Nach der Berechnung der Ähnlichkeiten werden die Bilder anhand der höchsten Ähnlichkeitswerte sortiert, und die Top 5 Bilder für jede Metrik werden ausgewählt. Um eine umfassendere Auswahl zu gewährleisten, werden die Top-Bilder aus allen drei Metriken kombiniert und nach der höchsten cosine similarity sortiert.
  Die ähnlichsten Bilder werden zusammen mit ihren Ähnlichkeitswerten in einer gridartigen Anordnung angezeigt, wobei die erste Spalte das Eingabebild und die folgenden Spalten die ähnlichsten Bilder enthalten.

<u>Vorteile:</u>

- **Kosinus-Ähnlichkeit**: Diese Metrik ist besonders robust gegenüber Skalierungsunterschieden in den Embeddings, da sie nur die Richtung der Vektoren betrachtet. Sie eignet sich gut für Fälle, in denen die absolute Größe der Embeddings nicht entscheidend ist.
- **Euklidische Distanz**: Diese Metrik ist intuitiv und misst die direkte Entfernung zwischen den Vektoren. Sie ist nützlich, wenn die Größe der Embeddings eine Rolle spielt.
- **Manhattan-Distanz**: Diese Metrik ist robuster gegenüber Ausreißern und misst die Summe der absoluten Differenzen. Sie ist besonders nützlich in

hochdimensionalen Räumen, wo Ausreißer einen großen Einfluss auf die euklidische Distanz haben könnten.

Nachteile:

- **Kosinus-Ähnlichkeit**: Obwohl sie robust gegenüber Skalierungsunterschieden ist, kann sie in Fällen, in denen die absolute Größe der Embeddings wichtig ist, weniger informativ sein.
- **Euklidische Distanz**: Diese Metrik kann empfindlich auf Ausreißer reagieren und ist nicht ideal für hochdimensionale Daten.
- **Manhattan-Distanz**: Obwohl robuster gegenüber Ausreißern, kann diese Metrik in einigen Fällen weniger intuitiv sein als die euklidische Distanz.

Farbanalyse (dimension_reduction_similarity.py):

Zunächst wird für jedes Bild ein Farb-Histogramm extrahiert, das die Verteilung der RGB-Werte im Bild repräsentiert. Das Histogramm wird normalisiert, um sicherzustellen, dass die Summe aller Werte gleich eins ist. Bis hier ist kein Unterschied zur color_profiling.py.
Nach der Extraktion der Histogramme wird eine Hauptkomponentenanalyse (PCA) durchgeführt. PCA reduziert die Dimensionen der Histogramme, wodurch die Rechenleistung und die Effizienz der nachfolgenden Berechnungen verbessert werden. Die Anzahl der Komponenten, auf die reduziert wird, kann angepasst werden. Wir haben festgestellt das alles über 75 Komponenten, ein gutes Ergebnis liefert. Darunter ist der Informationsverlust so groß, dass unpassende Bilder vorgeschlagen werden.
Nach der PCA-Transformation der Histogramme wird die Kosinus-Ähnlichkeit berechnet, wie gehabt. Es gilt wieder: Je kleiner der Winkel, desto ähnlicher sind die Bilder.

Wie zuvor kann man auch bei dieser Methode bis zu 5 Eingabebilder nehmen.

Vorteile:

- **PCA**: Durch die Reduktion der Dimensionen mittels PCA werden die Berechnungen effizienter und weniger speicherintensiv. PCA hilft auch dabei, Rauschen in den Daten zu reduzieren und die wichtigsten Merkmale der Bilder hervorzuheben.

- **Cosine similarity**: Diese Metrik ist robust gegenüber Skalierungsunterschieden in den Histogrammen und eignet sich gut, um die Richtung der Vektoren zu vergleichen.

Nachteile:

PCA: Obwohl PCA die Dimensionen reduziert und die Effizienz verbessert, kann es auch zu Informationsverlust führen, wenn zu viele Komponenten entfernt werden.

Cosine similarity: Diese Metrik berücksichtigt nicht die Größe der Vektoren, was in einigen Fällen zu unerwarteten Ergebnissen führen kann, wenn die absoluten Unterschiede in den Histogrammen wichtig sind.

4. Performanceanalyse und Laufzeitoptimierung

Meine Hardware:

CPU: I7-10700KF @ 3,8 GHz base clock speed mit bis zu 4,5 GHz

8 cores -> 16 Threads

GPU: GTX 3070 INNO 3D mit 5,888 CUDA cores und 8Gb GDDR6 Speicher

RAM: 64GB DDR4 mit 3200 MT/s

Um diesen Teil der Dokumentation passend einzuleiten, habe ich ein Bild von meiner modernen Festplattenkühlung gemacht.

Bottlenecks:

Bei dem Auslesen der Farbhistogramme und der Embeddings gab es unterschiedliche Bottlenecks, abhängig davon wie man optimieret hat.

Ohne wesentliches optimieren war das Auslesen der Bilder von der Festplatte die größte Hürde. Bei einem USB 3.0 Anschluss haben wir im Idealfall eine Übertragungsgeschwindigkeit von um die 40 Mb/s gehabt. Die USB-Ports direkt am Mainboard schienen oft eine stabilere Übertragungsrate aufzuweisen als die am Frontpanel.

Direkt davon abhängig ist die CPU. Durch das Parallelisieren der Auslesungen konnte man die Festplatte an das Limit bringen. Diese wurde dadurch sehr heiß, was mein ausgeklügeltes Setup im Bild erklärt.

Durch die Auslagerung der Berechnung von Embeddings auf die Grafikkarte konnte man viel Rechenzeit sparen. Vor allem bei der Visualisierung im Tensorboard hat das eine große Rolle gespielt.

Bei dem Auslesen der Embeddings und Farbhistogramme haben wir verhältnismäßig wenig optimiert, da diese nur einmal richtig durchlaufen mussten. Hier haben wir uns auf die Auslagerung der Berechnung auf die GPU verlassen und folgende Berechnungszeiten gehabt:

Embeddings um die 7 Stunden

Farbhistograms um die 11 Stunden

Tensorboard Berechnung haben wir von 11 Stunden auf 2,5 Stunden verkürzen können.

Durch das verwenden von .pkl Dateien kann bei der oben genannten Hardware eine Embedding-Ähnlichkeitsmessung pro Bild von 40 Sekunden, beim ersten Durchlauf, gewährleistet werden. Die nachfolgenden Durchläufe werden immer unter 20 Sekunden sein, da der Code schon kompiliert ist.

Bei den Farbhistogrammen geht es schneller. Hier sind wir bei 30 Sekunden im ersten Durchlauf und wieder bei unter 20 Sekunden, in an folgenden Berechnungen.

Die Nutzung der GPU und Parallelisierung war es uns möglich in vielen einzelnen Bereichen Rechenzeit zu sparen.

Unit tests:

Die Unit-Tests wurden mit pytest entwickelt, um die korrekte Funktion verschiedener Module zur Verarbeitung von Bildmetadaten, Histogrammen, Embeddings und Ähnlichkeitsmessungen sicherzustellen. Die Tests decken dabei die wesentlichen Schritte von der Speicherung und dem Abruf von Daten in einer SQLite-Datenbank, über die Nutzung von Pickle-Dateien, bis hin zu komplexeren Berechnungen wie der Hauptkomponentenanalyse (PCA) und unterschiedlichen Ähnlichkeitsmetriken ab.

Zunächst werden die notwendigen Bibliotheken und Module importiert, und durch ein Setup via sys.path.insert wird sichergestellt, dass die zu testenden Module korrekt importiert werden können. Die Tests nutzen Fixtures, um eine saubere Testumgebung zu schaffen. Die sqlite_connection-Fixture erstellt eine temporäre In-Memory-Datenbank mit Tabellen für Bildmetadaten und Histogramme, während die temp_pickle_file-Fixture eine temporäre Pickle-Datei für Histogrammdaten generiert, die in den Tests verwendet wird.

Die Testfunktionen sind darauf ausgelegt, zentrale Aspekte der Datenverarbeitung zu überprüfen. So wird etwa in test_histogram_and_metadata_connection sichergestellt, dass die unique_id korrekt zwischen der Datenbank und der Pickle-Datei übereinstimmt und die Metadaten richtig abgerufen werden. Weitere Tests wie test_load_histograms und test_load_histograms_color überprüfen, ob Histogrammdaten korrekt aus Pickle-Dateien geladen werden können. Wir testen für die Embeddings die test_cosine_similarity, test_euclidean_distances und test_cityblock. Für die RGB-Histogramme testen wir test_chi2_distance und test_pca_cosine_similarity. Hiermit wollen wir sicherstellen das die Implementierung verschiedener Ähnlichkeitsmetriken richtig berechnet werden.

Insgesamt gewährleisten diese Unit-Tests, dass alle kritischen Funktionen zur Verarbeitung von Bilddaten korrekt arbeiten, was die Stabilität und Zuverlässigkeit des Systems sichert.

5. Machbarkeitsanalyse/Diskussion

Basierend auf unserer Performanceanalyse ist die Skalierbarkeit der Software ein entscheidender Faktor, um die Effizienz und Nützlichkeit als Suchmaschine zu bewerten. Unsere Software zeigt vielversprechende Ergebnisse bei der Ähnlichkeitssuche, aber es gibt einige wichtige Punkte zu beachten:

Skalierbarkeit der Software:

Unsere Software ist in der Lage, effizient mit einem Datensatz von bis zu 400.000 Bildern zu arbeiten. Die verwendeten Techniken wie Parallelisierung und GPU-

Nutzung haben es uns ermöglicht, die Berechnungszeiten erheblich zu reduzieren. Insbesondere die Verwendung von vortrainierten Modellen wie ResNet50 zur Embedding-Generierung und die Optimierung der Histogramm-Berechnungen durch PCA und Chi-Quadrat-Distanz haben gezeigt, dass die Software gut mit großen Bildmengen umgehen kann.

Hauptlimitierungen

Die größten Hürden der aktuellen Implementierung liegen in der hohen Rechenlast, insbesondere bei der Verwendung komplexer Bild-Embeddings. Die Erstellung und Speicherung von .pkl-Dateien für Embeddings und Histogramme kann viel Speicherplatz beanspruchen, was bei extrem großen Datenmengen problematisch werden kann. Ein weiteres Problem ist die Abhängigkeit von der I/O-Leistung der Festplatte, insbesondere beim Laden und Speichern großer Dateien.

Mögliche Verbesserungen

Um diese Limitierungen zu überwinden und die Laufzeit des Programms weiter zu optimieren, könnten folgende Maßnahmen ergriffen werden:

- **Nutzung von Cloud-Diensten**: Durch die Verlagerung der Rechenlast in die Cloud könnten Ressourcen nach Bedarf skaliert werden, um mit sehr großen Datenmengen effizient umzugehen. Dies würde insbesondere bei der parallelen Verarbeitung von großen Bildmengen helfen, jedoch auch eine gute Menge Geld kosten.

- **Verwendung von Speichercaching**: Das Implementieren von Caching-Mechanismen, um bereits berechnete Ergebnisse zwischenzuspeichern, könnte die Antwortzeiten bei wiederholten Abfragen erheblich verkürzen. Werkzeuge wie Redis oder Memcached könnten hier nützlich sein.

- **Optimierung der Parallelisierung**: Die Parallelisierung könnte weiter verfeinert werden, indem Workloads auf mehrere GPUs verteilt werden (Multi-GPU-Betrieb) oder durch die Implementierung von verteiltem Rechnen über mehrere Server hinweg. Dies würde die Rechenzeit insbesondere bei der Erstellung von Embeddings und der Verarbeitung großer Bildmengen reduzieren.

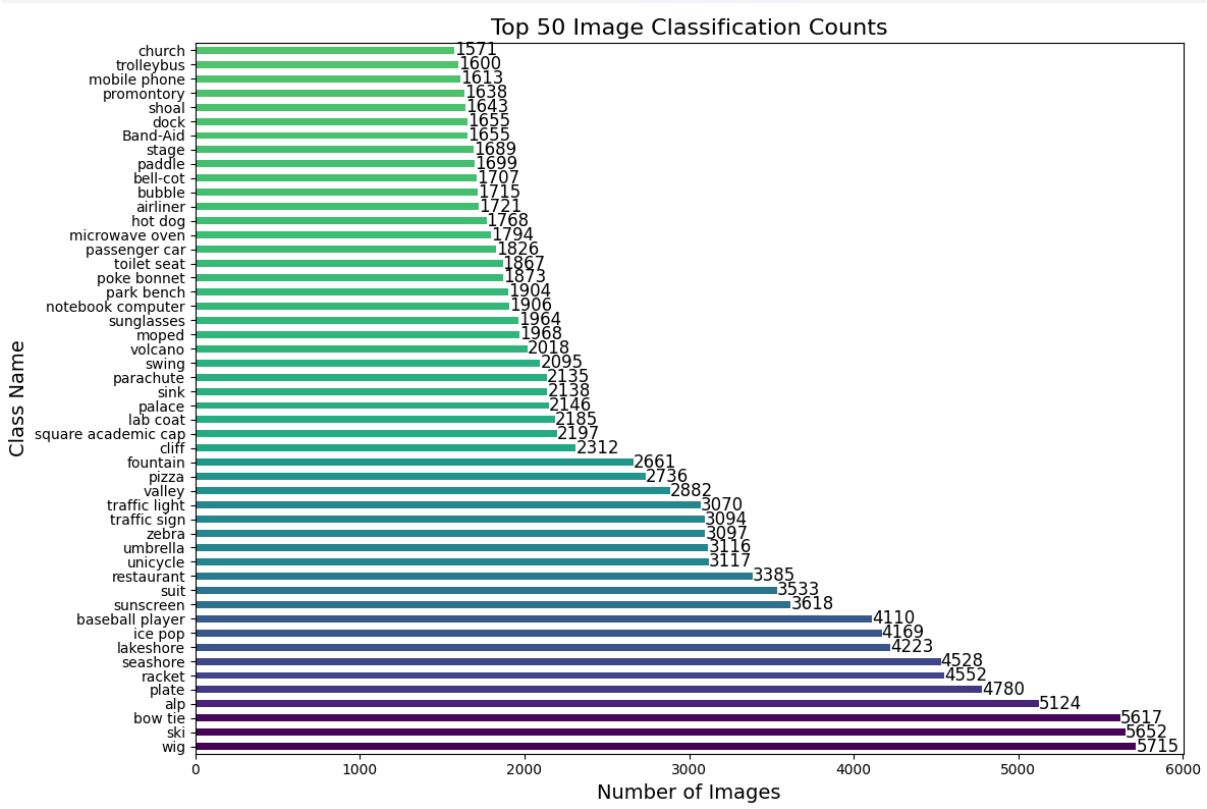
Fazit zur Skalierbarkeit

Insgesamt zeigt unsere Software eine gute Skalierbarkeit für mittelgroße bis große Bilddatensätze und könnte mit zusätzlichen Optimierungen auch für sehr große Bildmengen eingesetzt werden. Die wichtigsten Herausforderungen liegen in der effizienten Speicher- und Rechenlastverwaltung, die durch spezialisierte Hardware und optimierte Algorithmen verbessert werden können.

6. Big Data Bildanalyse

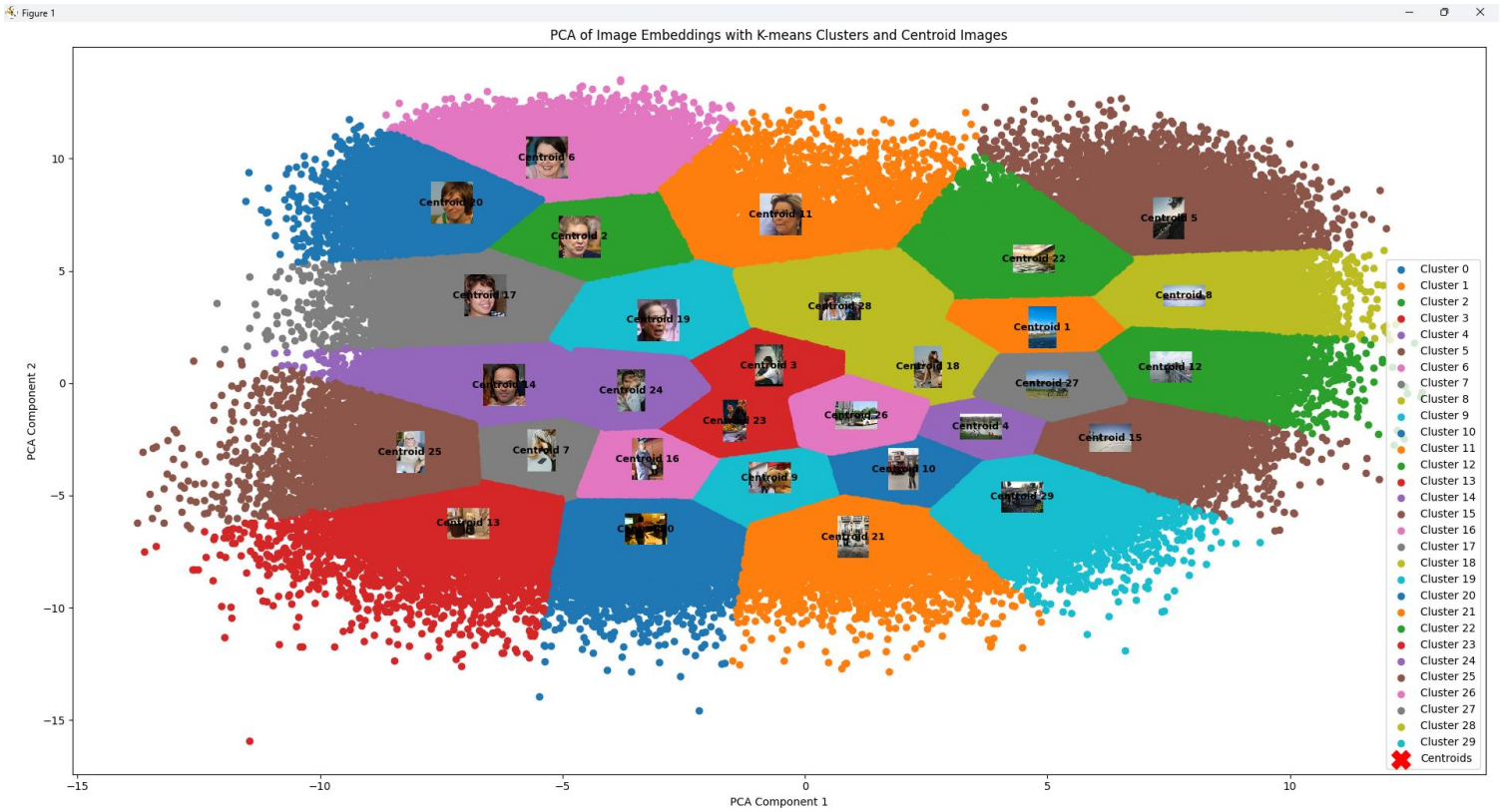
Für die Analyse der Bilddaten haben wir verschiedene Methoden angewandt:

- **Klassifikation mit ResNet50:** Wir haben das Modell ResNet50 verwendet, um alle Bilder in der Datenbank zu klassifizieren. Eine interessante Feststellung ist das am häufigsten Perücken klassifiziert wurden. Wie wir festgestellt haben werden vom Resnet nicht direkt Menschen klassifiziert. Das ResNet Klassifiziert Baseballspieler oder Tennisspieler beispielsweise. Hier eine Übersicht der Top 50 Klassifizierungen:



Darüber hinaus hatten wir auch mit einem Inception v3 Model klassifiziert. Uns ist jedoch erst im Anschluss aufgefallen das die Klassifizierungen gleich sind, da beide Modelle auf dem ImageNet aufgebaut sind. Diese Erkenntnis hat uns geschätzt um 10 Stunden Rechenzeit gekostet.

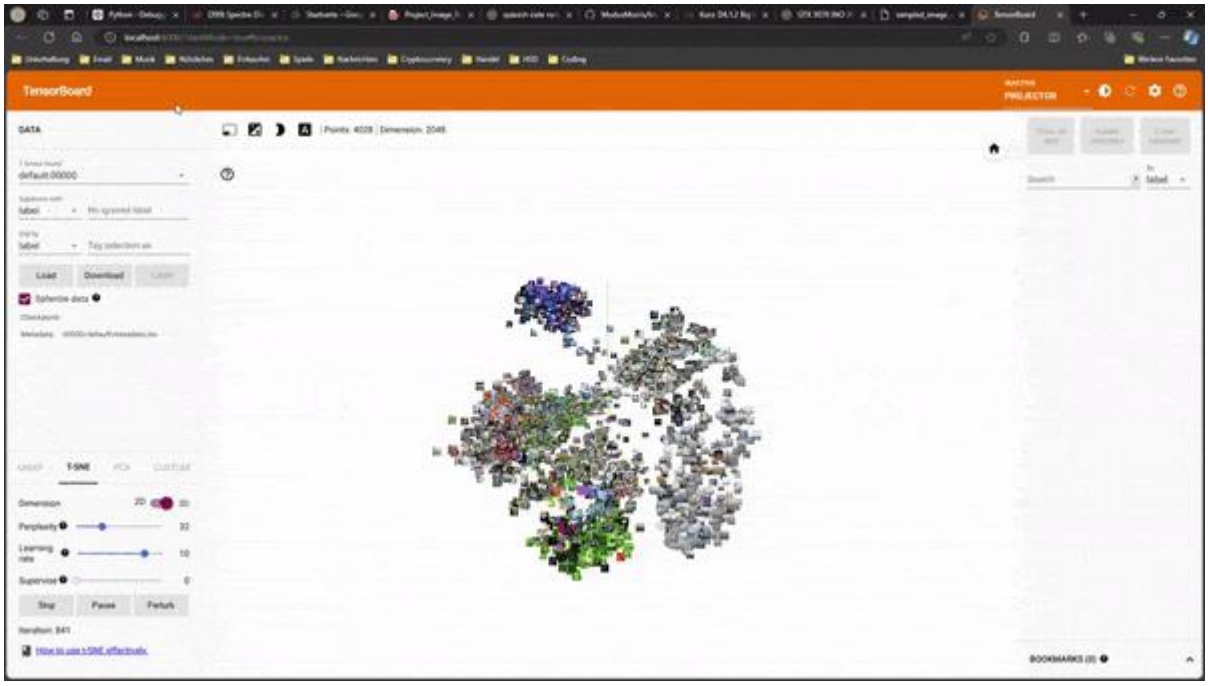
- **PCA:** Mit einer PCA-Dimensionsreduzierung können wir sehr schnell alle Punkte plotten. Der Graph sieht mit 30 Clustern so aus:



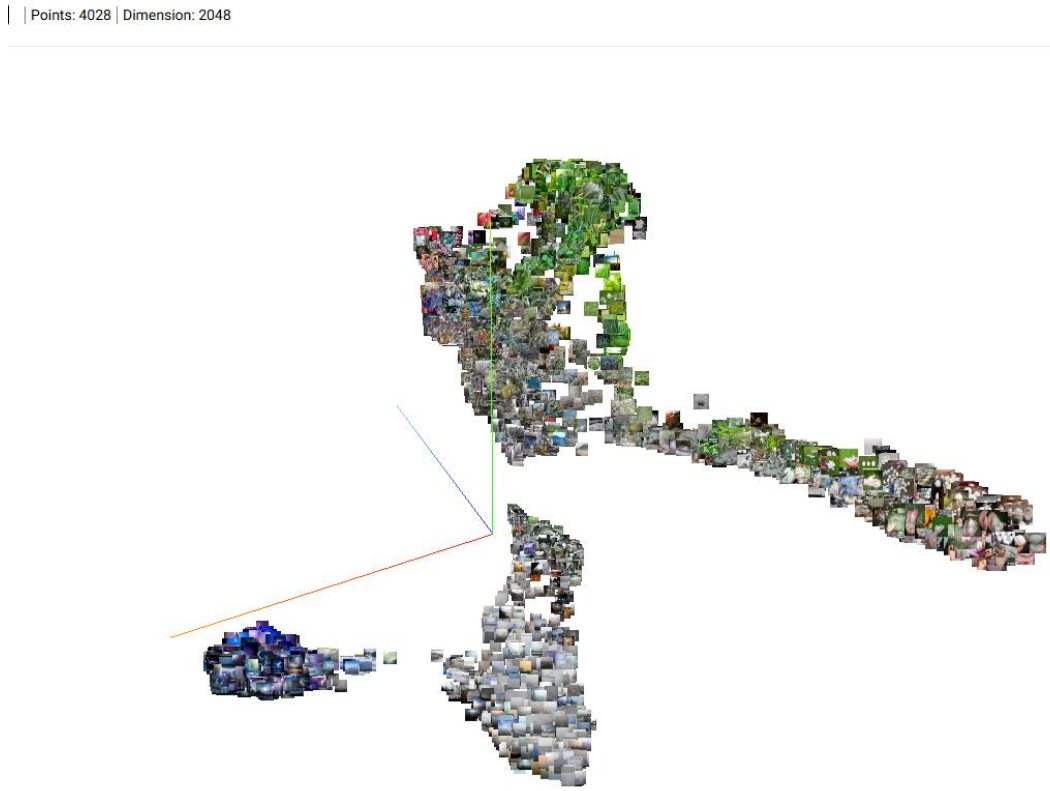
Wir haben die Mittelpunkte der Cluster dargestellt und die jeweiligen Bilder dazu geplottet. Man kann sogar schön erkennen wie sich die Menschenbilder in der oberen linken Ecke des Plots sammeln. Auf der anderen Seite sind Landschaften zu erkennen. Obwohl sehr viele Informationen verloren gehen, sind selbst noch auf 2 Dimensionen noch Gruppierungen zu erkennen.

Erster Batch mit PCA:

Erster Batch mit T-SNE:



Erster Batch mit UMAP:



- **Networkgraph:** Unser eigentliches Ziel war es einen großen Networkgraph zu erstellen, in welchem alle Bilder geplottet sind. Ähnliche Bilder sollen beieinander sein und wenn sie fast identisch sind ein großes Cluster bilden. Wir haben viel Bibliotheken probiert und Ansätze, jedoch haben wir keinen wirklichen Erfolg hier gehabt. Wir probierten Bibliotheken wie Pyvis oder Pyplot. Unterschiedliche Ansätze haben uns teilweise zu über einer Woche an Rechenzeit gekostet, was nicht gut sein kann. Ich habe einen unserer Networkgraph**s** auch hochgeladen. Daher wird im Repository angezeigt das wir zu 99,5% HTML code haben.
  In dem Graphen unten haben wir die Louvain community detection verwendet, um überlappende Knoten zu Bündeln und als einen darzustellen. Diese hat nicht so gut funktioniert wie erhofft und dauert auch lange zum Laden im Browser. Oft wurde sogar angezeigt, dass das Laden sich aufgehangen hat. Man muss jedoch nur bei dem Popup im Browser auf „warten“ drücken und nach ungefähr 15 Minuten, hat der Graph geladen.