Projekt "Image Recommender": Dokumentation

Teil 1 – Image recommender software

1. Motivation/Ziel des Projekts

Das Ziel des Projekts war die Entwicklung eines Python Programms zur Bildempfehlung („Image Recommender“) unter Verwendung verschiedenster vorgegebener (Big Data-) Programm Bausteine. Das Programm sollte möglichst optimiert und in der Lage sein, auf Grundlage eines Input Images andere, ähnliche Bilder zu empfehlen. Die Ähnlichkeit sollte mit verschiedenen Metriken bestimmt werden, beispielsweise auf Basis ähnlicher Farbschemas oder mit Hilfe von Embeddings (Technik der Dimensionsreduktion).

2. Programm Design

Das Programm besteht aus mehreren Hauptkomponenten, die miteinander interagieren (Grafik/Flowchart s. unten). Die wichtigsten Elemente hierbei sind ein Python-Generator zum Laden der Bilder, die interne Verwaltung der Bilder über eindeutige Bild-IDs und die Verwendung von Datenbanken zur Verknüpfung von Bild-IDs und Speicherorten. Außerdem verwenden wir sozusagen sieben verschiedene Metriken, um die Ähnlichkeit zwischen Bildern zu quantifizieren. Eine Metrik basiert auf dem Farbprofil bzw. der Farbähnlichkeit, während eine andere Metrik komprimierte Embeddings verwendet, die mithilfe von Dimensionsreduktionstechniken erstellt wurden. Mehr dazu weiter unten.

Im Folgenden werden die zentralen Bausteine vorgestellt.

Ein Generator von tensorflow lädt die Bilder des Trainingsdatensets. Mit Hilfe von MobileNet berechnet er die Embeddings der Bilder. Durch Aufrufen der Funktion „calculate_histogram“ werden außerdem RGB / HSV Werte mit Hilfe der openCV Bibliothek berechnet. Zum Schluss werden Embeddings, RGB / HSV Werte und die Pfade zu den Speicherorten der Bilder in pickle Dateien gespeichert. In jeder der Dateien sind die Daten der Bilder mit einer eindeutigen Image ID verbunden, so dass das Bild eindeutig in jeder pickle Datei gefunden werden kann.

Die Pfade zu den Speicherorten der Bilder werden mit den entsprechenden Bild IDs in einer Datenbank gespeichert (sqllight).

Ein weiterer zentraler Baustein des Programms – wenn nicht der zentralste für die eigentliche Bilderkennung - ist die Funktion find_all_similarities.
Diese erhält drei Inputs: Den Pfad zu dem Speicherort des Bildes, für das nach ähnlichen Bildern gesucht werden soll, die Anzahl der ähnlichsten Bilder, die angezeigt werden sollen und die Metrik, nach welcher nach Ähnlichkeiten gesucht werden soll. Hier kann der Nutzer zwischen sieben verschiedenen Versionen auswählen: embeddings (für die Suche nach Bildern auf Grundlage von erkannten Formen in den Bildern), hsv_euclidean, hsv_manhattan, hsv_cosine (Ähnlichkeit auf Grundlage von HSV Werten) oder rgb_euclidean, rgb_manhattan, rgb_cosine (Ähnlichkeit auf Grundlage von RGB Werten).

Aus dieser Funktion werden die IDs der fünf ähnlichsten Bilder zurückgegeben. Daraufhin wird eine Verbindung mit der Datenbank, in der die Bildpfade hinterlegt sind, hergestellt und die Pfade, die zu den IDs korrespondieren, ausgegeben.

Mithilfe der Pfade des Testbildes und der gefundenen ähnlichsten Bilder, werden die Bilder daraufhin geplottet (matplotlib) und der Nutzer kann überprüfen, ob die gefundenen Bilder Sinn ergeben.

Für die wichtigsten Funktionen, vor allem in Bezug auf Ähnlichkeit und Laden der Daten in Datenbanken, haben wir außerdem Testfunktionen erstellt. Diese überprüfen, entweder mit Dataframes/Listen aus Dummybilddaten oder echten Inputbildern, ob die einzelnen Funktionen, außerhalb des Workflows des Programmcodes, auch die Outputs liefern, die wir erwarten.

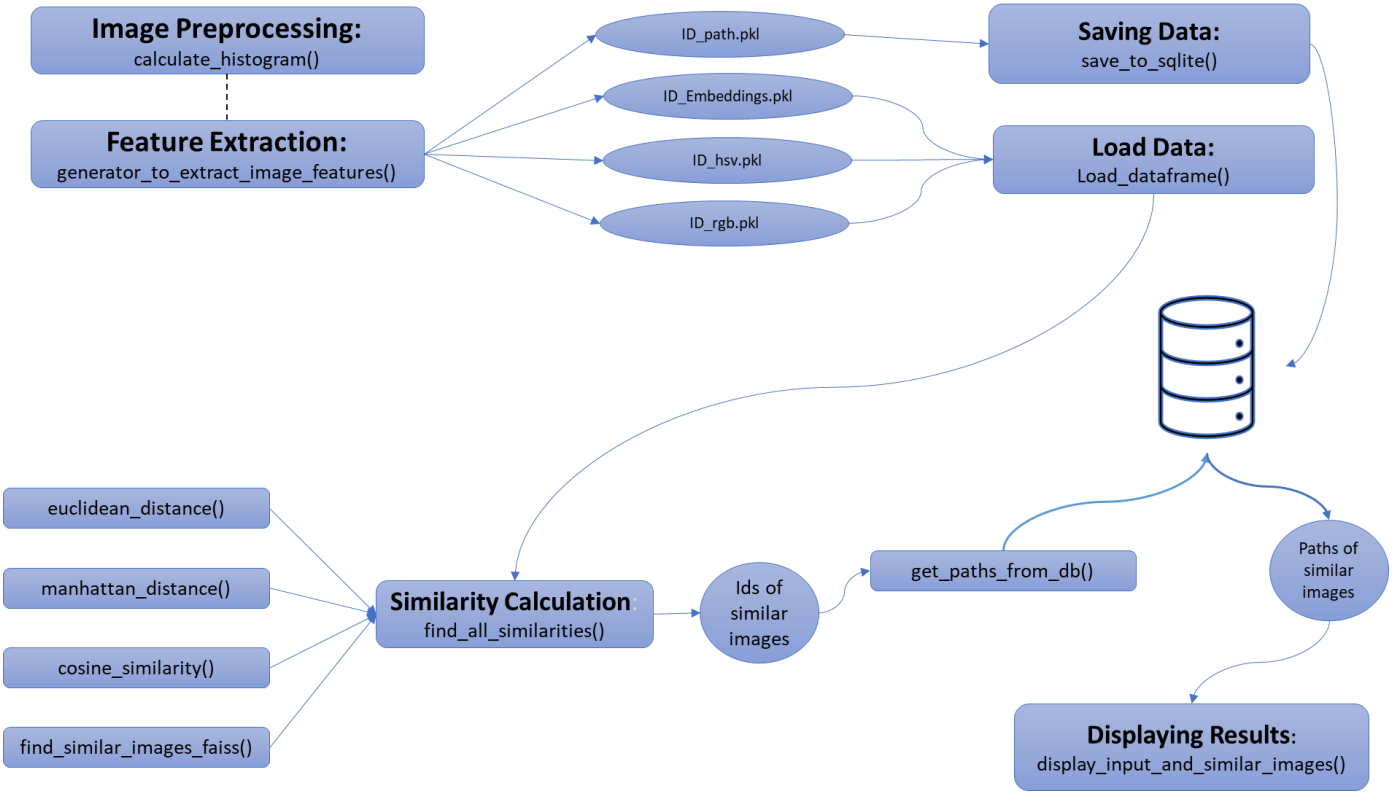

Abb. 1: Flowchart Programmdesign. Die Pfeile verdeutlichen, in welcher Reihenfolge die verschiedenen Programmbausteine miteinander interagieren.

3. Image similarity

Wir benutzen verschiedene Metriken. Durch Kombination dieser kommen wir am Ende auf sieben verschiedene Methoden, mit denen der Nutzer sich ähnliche Bilder berechnen lassen kann.

Embeddings:
Embeddings sind kompakte, numerische Repräsentationen von Bildern. Die Repräsentationen enthalten berechnete Charakteristika der Bilder. Dies erlaubt, die

wichtigsten Features der Bilder einerseits platzsparend zu speichern, aber andererseits die wichtigsten Erkennungsmerkmale für das Finden von Ähnlichkeiten zu behalten. In unserem Code werden die Embeddings für alle Trainingsbilder direkt mit Hilfe von MobileNet berechnet und dann in pickle Format gespeichert. Die Embeddings des Bilds, das auf ähnliche Bilder untersucht werden soll, werden später in einer anderen Funktion berechnet, dies geschieht auch mit MobileNet.
Vorteile der Benutzung von Embeddings sind die kompakte Repräsentation der Bilder, was effiziente Speicherung, Abrufung und Vergleichen der Bilder ermöglicht. Der Code der der Berechnung von Embeddings zu Grund liegt, ist aus Deep Learning Netzen abgeleitet, welche darauf optimiert sind Charakteristiken der Bilder extrahieren zu können. Durch die Nutzung der Algorithmen aus komplexen Deep Learning Modellen, die auf eine große Zahl an Bildern trainiert worden sind, können Embeddings sehr gut generalisieren und damit sehr gut ähnliche Bilder finden für Bilder, die sie vorher noch nicht gesehen haben. Beide dieser Vorteile haben wir in unserem Programm bemerkt, das arbeiten mit berechneten Embeddings geht sehr schnell (v.A. nach Implementierung von FAISS, mehr dazu unter 4.)) und die Vorhersagen für verschiedenste Testbilder, mit variierendem Komplexitätsgrad (von Wiesen bis Kinder), waren sehr treffend.
Mit der Verwendung von Deep Learning Netzen bei Embeddings kommen auch Nachteile. Die Generierung der Embeddings kann sehr zeit- und arbeitsspeicheraufwendig sein, da jedes einzelne Bild durch ein Deep Learning Netz geschleust werden muss. Außerdem ist die Qualität der Embeddings abhängig von dem jeweils genutzten DeepLearning Netz. Weiterhin ist die Berechnung der Embeddings auch eine Art „Black Box“, da die Charakteristiken der Bilder als numerische Repräsentationen gespeichert werden. Wenn ein Mensch diese Vektoren betrachten würde, könnte er/sie daraus vermutlich nicht schließen, wieso ein bestimmter Vektor ein bestimmtes Bildfeature repräsentiert. Bei Recherchen haben wir außerdem gelesen, dass Embeddings verhältnismäßig stark zu overfitting neigen – gleichzeitig hat die Image Recommendation mit Embeddings bei uns sehr gut (von allen Metriken am besten) funktioniert. Möglicherweise war unser Trainingsdatensatz groß genug/divers genug, um Overfitting zu vermeiden. Eine andere Möglichkeit wäre, dass wir bei unseren Testbildern noch keine Nichenkategorie getroffen haben, bei der sich eine schlechte Vorhersage durch Overfitting zeigen würde.

Farbwerte:
Weiterhin haben wir Histogramme für HSV und RGB Werte berechnet, um diese dann mit den HSV oder RGB Werten eines Input Image vergleichen zu können. Histogramme sind die Repräsentation der Farbverteilung in einem Bild.
RGB Histogramme betrachten dabei die Intensität von Rot, Grün und Blau, HSV Histogramme betrachten die Schattierung („hue“), Sättigung und wie hell/dunkel eine Farbe in einem Bild ist. Soweit wir herausfinden konnten, ähnelt die Betrachtung von HSV Werten eher der Art, wie Menschen Bilder wahrnehmen, als RGB.
Die Histogramme werden als Vektoren gespeichert. Durch Vergleichen des entsprechenden Histogramm Vektors des Inputbildes (aus der Funktion „calculate_histogram“) mit denen aus dem bereits gespeicherten Datensatz (ebenfalls berechnet mit der Funktion „calculate_histogram“ und abgespeichert in pickle Dateien) können die ähnlichsten Vektoren gefunden werden.

Auch bei diesen Methoden gibt es einige Vor- und Nachteile, die zu beachten sind.
Vorteile von RGB sind, dass es das verbreiteste digitale Farbspeichermodell ist, und es deshalb für diese Art der Bilddatenspeicherung viele optimierte Python Bibliotheken gibt. Da RGB der normalerweise verwendete Farbraum für digitale Bildanalyse ist, gibt es viele Tools, um RGB einfach zu extrahieren und zu analysieren. Da die einzelnen Farbkanäle gespeichert werden, gibt RGB sehr genaue Informationen über die Intensitäten der genauen Farben wieder. Dies spiegelte sich auch in unserem Code wider, wir verwendeten die viel genutzte openCV Bibliothek für die Histogramme. Obwohl openCV die Farbkanäle in umgekehrter Reihenfolge angibt, war die Extrahierung und Speicherung der RGB Werte in drei Zeilen Code möglich.
Es gibt auch Nachteile. Da RGB „nur“ die Intensitäten der drei Farbkanäle speichert, ist es unempfindlich gegenüber der Helligkeit des Bildes, weshalb der direkte Vergleich zwischen Bildern nur unter Nutzung von RGB z.T. unzureichend ist. Bei den Bildern, die wir testeten fiel uns auf, dass HSV etwas besser funktioniert, aber alles in allem war der Unterschied nicht übermäßig groß. Außerdem ist RGB limitiert auf drei verschiedene Farbkanäle und ihre Mischung, aber Wahrnehmung von Farbe/Ähnlichkeit von Farben wird durch mehr beeinflusst, z.B. Helligkeit oder Sättigung.

Einige dieser Probleme werden durch die Vorteile der HSV Analyse adressiert.

Da HSV Werte die Helligkeit eines Bildes mitbeachten, ist es weniger sensitiv als RGB für Fehler durch variierende Helligkeit in ähnlichen Bildern. Außerdem wird in HSV ein „Perceptually uniform color space“ verwendet, was bedeutet, dass die Art, wie Ähnlichkeiten zwischen Bildern gefunden werden, dem, was Menschen als ähnlich empfinden, mehr ähnelt als RGB. Wie schon gesagt konnten wir bei Testbildern die Ähnlichkeit, die durch HSV Analyse gefunden wurde, z.T. besser nachempfinden als die Similarity, die durch RGB Analyse gefunden wurde.

Nachteile der HSV Analyse sind, dass sie Farbkanäle nicht ganz so genau einfängt wie RGB und auch nicht so verbreitet ist wie RGB. Dadurch gibt es weniger Python Bibliotheken für die Arbeit mit HSV, als für RGB.
Diese Nachteile haben uns nicht so sehr berührt. Da die HSV Ähnlichkeit mehr der menschlichen Wahrnehmung nachempfunden ist, war es für uns nicht störend, dass RGB Kanäle nicht so genau betrachtet werden. Auch haben wir in openCV alle Methoden gefunden, die wir für die Berechnung von HSV Werten brauchten.

Die Ähnlichkeit zwischen den Histogrammvektoren wird, je nach Wahl des Nutzers, über Euclidean Distance, Manhatten Distance oder Cosine Similarity berechnet.
Der Vorteil der Euclidean Distance ist, dass sie sehr intuitiv verständlich und dadurch einfach anwendbar für Menschen ist. Ein Nachteil kann sein, dass sie anfällig für sehr stark voneinander abweichende Wertspannen in den zu vergleichenden Histogrammen ist, weshalb man darauf achten muss, dass alle Werte auf gleiche Weise normalisiert werden, bevor Euclidean Distance angewendet wird.

Manhattan Distance ist weniger anfällig für Unterschiede in der Wertspanne der betrachteten Histogramme als Euclidean Distance. Da diese Metrik die Ähnlichkeit aufgrund der Schritte berechnet, die man braucht um von einem Bildvektor zum nächsten zu kommen,

werden keine Winkel oder Richtungen zwischen den Vektoren betrachtet, wodurch Bildfeatures übersehen werden können, die durch diese Eigenschaften repräsentiert werden.

Cosine Similarity vergleicht die Winkel zwischen den Werten der Histogramme, und ist dadurch unempfindlicher gegenüber verschiedenen Größenordnungen in den zu vergleichenden Vektoren. Cosine Similarity vergleicht Muster innerhalb der Histogramme zwischen verschiedenen Bildern. Dies bedeutet auch, dass Bilder, die sehr unterschiedlich sind, weil sie z.B. verschiedene Helligkeits-Abstufungen aufweisen, von der Cosine Similarity als ähnlich empfunden werden können, wenn sich diese unterschiedlichen Abstufungen innerhalb der Histogramme in einem ähnlichen Verhältnis befinden.

Auf uns wirkten die Vor- und Nachteile von Euclidean und Manhatten Distance und Cosine Similarity sehr ausgeglichen, in der Verwendung von Testbildern sahen wir kaum unterschiedliche Ergebnisse zwischen den verschiedenen Methoden.

4. Performance Analyse + Laufzeit Optimierung

Wir haben für alle Metriken die Laufzeit (mit der time Bibliothek) aller Funktionen gemessen, die wir direkt in main() aufrufen. Dies sind die Ergebnisse:

Verwendete Metrik	Funktion	Execution Time (s)
hsv_euclidean	load_dataframes	4,10
	find_all_similarities()	2,80
	get_path_from_db()	0,14
	display_input_and_similar_images	1,81
	CPU times	9,12
hsv_cosine	load_dataframes	3,32
	find_all_similarities()	8,71
	get_path_from_db()	0,15
	display_input_and_similar_images	1,20
	CPU times	14,10
hsv_manhattan	load_dataframes	3,50
	find_all_similarities()	2,31
	get_path_from_db()	0,14
	display_input_and_similar_images	0,92
	CPU times	7,41
rgb_euclidean	load_dataframes	3,90
	find_all_similarities()	3,37
	get_path_from_db()	0,15
	display_input_and_similar_images	1,58
	CPU times	9,78
rgb_manhattan	load_dataframes	3,91
	find_all_similarities()	2,76
	get_path_from_db()	0,17
	display_input_and_similar_images	1,86
	CPU times	9,38
rgb_cosine	load_dataframes	3,22
	find_all_similarities()	9,26
	get_path_from_db()	0,16
	display_input_and_similar_images	1,62
	CPU times	15,00
Embeddings	load_dataframes	4,15
	find_all_similarities()	2,40
	get_path_from_db()	0,15
	display_input_and_similar_images	1,00
	CPU times	8,16

Tab. 1: Gemessene Laufzeit in Sekunden der Funktionen, die in main() aufgerufen werden. In der Spalte ganz links steht die Metrik, für die main() jeweils ausgeführt wurde

Generell befinden sich alle Messungen in der Tabelle im Bereich weniger Sekunden. Es fällt auf, dass die Funktion „find_all_similarities()“ bei Verwendung der Cosine Similarity viel länger dauert, als bei allen anderen Metriken. Dies kann verschiedene Gründe haben: Die Berechnung des Kosinus erfordert mehr Rechenschritte als die Euclidean oder Manhattan Distance, außerdem sind die verwendeten Rechenschritte bei der Cosine Similarity komplexer (Division, Quadratwurzel) als bei der Euclidean oder Manhattan Distance. Dazu kommt, dass für diese Rechenoperation auch die Norm jedes Vektor gespeichert werden muss, auch dies kann die Laufzeit verlängert.

Generell ist unser Code schon sehr optimiert, dies konnten wir unter anderem dadurch feststellen, dass es schwierig war, durch Bibliotheken wie Dask oder Numba weitere Optimierung zu erreichen.

Vor allem das Finden ähnlicher Bilder mit Embeddings ging schnell, wahrscheinlich, weil wir hier Funktionen aus dem FAISS Modul (Facebook AI Similarity Search) eingebaut haben. FAISS ist eine Bibliothek von Facebook, die Algorithmen für effiziente Ähnlichkeitssuche und Clustering im Big Data Bereich bereitstellt, vor Allem für Suchen in hochdimensionalen Räumen, wie Bild Embeddings.

Zwei offensichtliche Bottlenecks die wir aber gefunden, und dann z.T. optimieren konnten, sind einerseits der Bildgenerator (also das Laden der Bilder und speichern der jeweiligen Werte), und die Berechnung von TSNE für die spätere Analyse.

In den ersten Wochen des Projekts verwendeten wir einen Bildgenerator, den wir selbst geschrieben hatten. Um Sicherzugehen, dass die individuelle Bild ID bei RGB und HSV Values, den Embeddings und dem Speicherort der Bilder gleich sind, luden wir alle diese Informationen eines Bildes auf einmal und speicherten sie dann in einem Dataframe. Dies funktionierte gut, als wir noch den kleinen Datensatz mit den Wetterbildern hatten. Auf dem großen Datensatz von 140.000 Bildern bekamen wie damit große Probleme, da ein Dataframe, wenn Zeilen hinzugefügt werden, nicht einfach verlängert wird, sondern gelöscht und komplett neu erstellt wird mit der neuen Zeile. Auf diese Weise hätte unser Training ca. einen Tag gebraucht.
Wir lösten dieses Problem, in dem wir den Tensorflow Image Generator verwendeten. Dieser ist bereits sehr optimiert und ermöglicht das Einlesen und Verarbeiten von Bildern in Batches, was die Verarbeitungsgeschwindigkeit sehr verbessert und den Speicherbedarf reduziert, da die Batches parallelisiert verarbeitet werden. Außerdem speicherten wir die Bilder in Pickle Dateien. Vorteile von Pickle für die Optimierung sind, dass Python-Objekte einfach in ein binäres Format serialisiert werden können und dadurch weniger Speicherplatz benötigen (dabei wird aber die ursprüngliche Datenstruktur beibehalten). Außerdem wird für die Erstellung und Auslesung von Pickle Dateien jeweils nur eine Zeile Code benötigt.

Die Feature Extraction führten wir einerseits mit openCV (für RGB/HSV) und mit MobileNet (für die Embeddings) durch. Da das Laden und die Feature Extraction der Bilder mit den openCV Funktionen schnell genug war (ca. 2h auf dem Rechner im Daisy-Pool) und die Ergebnisse mit openCV sehr gut waren, entschieden wir uns, dies nicht zu verändern, ansonsten hätte man versuchen können, den Code mit Funktionen aus anderen Bildbearbeitungsbibliotheken weiter zu optimieren.

Außerdem ist die Frage, ob ein anderes pretrained Model schneller die Embeddings berechnet hätte. MobileNet ist bereits „lightweight“, und mit ResNet50, was hier eine Alternative wäre, dauerte das Laden unserer Bilder länger.
Weiterhin hätten wir überlegen können, die GPU für die Feature Extraction zu aktivieren. Da die Feature Extraction mit ca. 2h schon ziemlich schnell ging, und wir nicht unendlich viel Zeit verwenden wollten, um einen Prozess, der in unseren Augen für unseren Bedarf „gut genug“ war, mit ungewissem Ergebnis weiter zu optimieren, sind wir bei der CPU geblieben.
Bei der Erstellung der Histogramme könnte man beispielsweise mit Dask die Threads parallelisieren, und so eine Steigerung der Geschwindigkeit erreichen, oder auch hier überlegen, die GPU zu verwenden.

Um den einmaligen Prozess der Speicherung der Bildpfade in der Datenbank zu beschleunigen, wäre es möglich, die Daten nicht einzeln zu speichern, sondern in Batches, z.B. von 100 Pfaden. Außerdem gibt es Datenbanken, die besser auf schnelleren Zugriff und „Big Data Handling“ optimiert sind, z.B. NoSQL Datenbanken oder distributed file systems.
Eine weitere Frage wäre, ob eine Parallelisierung bei dem Schritt, die Pfade ähnlicher Bilder aus der Datenbank abzurufen, möglich wäre (also z.B. immer nach zwei Pfaden gleichzeitig zu suchen). Generell war unsere Speicherung der Bilder in der Datenbank aber so schon sehr schnell, da wir die Daten zuvor in einer pickle Datei abgelegt hatten.

Auf das Bottleneck der Dimensionsreduktion wird unter 2B) eingegangen.

Wir hatten versucht, einige unserer Funktionen mit Dask oder Numba zu optimieren, im folgenden sind Screenshots der entsprechende Code Modifikation am Beispiel der Berechnung der Cosine Similarity und der dabei gemessene Zeit.

Unser Code (ohne Dask/Numba):

```
CPU times: total: 8.91 s
Wall time: 8.88 s
```

Optimierungsversuch Dask:

```
CPU times: total: 12.8 s
Wall time: 11.6 s
```

```python
import dask.dataframe as dd
from scipy.spatial import distance

def cosine_similarity(df, test_color, top_n):
    dask_df = dd.from_pandas(df, npartitions=100) # you can adjust npartitions based on your cpu cores
    dask_df['hist_similarity'] = dask_df['Histogram'].apply(lambda x: 1 - distance.cosine(test_color, x), meta=('float'))
    dask_df = dask_df.nlargest(top_n, 'hist_similarity')
    return dask_df.compute()
```

Optimierungsversuch Numba:

```
CPU times: total: 9.58 s
Wall time: 9.22 s
```

```python
import numba
from numba import jit
import numpy as np

@jit(nopython=True)
def cosine_similarity_numba(v1, v2):
    # Numba optimized function to compute cosine similarity between two vectors
    norm_v1 = np.sqrt(np.sum(v1 ** 2))
    norm_v2 = np.sqrt(np.sum(v2 ** 2))
    cos_sim = np.dot(v1, v2) / (norm_v1 * norm_v2)
    return cos_sim
```

Der Einbau von Dask hat die Ausführungszeiten des Codes verlangsamt, bei Numba gibt es keinen Unterschied.
Zu der Verlängerung der Zeit bei Dask könnte beitragen, dass Dask Overhead hat, um bestimmte Tasks zu planen und zu koordinieren. Bei komplizierten/größeren Operationen kann dies zu signifikanten Zeitersparnissen führen, bei eher kleinen kann die Overhead Zeit die gewonnen Zeitersparnis überwiegen. Außerdem verwendet Dask zur Optimierung verteilte Arrays und Dataframes, für die Daten hin und geschoben und serialisiert/ deserialisiert werden müssen. Auch das kann die durch Dask-Optimierung gewonnene Zeit überwiegen.
Numba ist vor allem geeignet für numerische Operationen, nicht so sehr für Sortierung oder String-Operationen. Bei uns handelt es sich allerdings um eine numerische Operation, eine andere Möglichkeit wäre, dass die Berechnung der Cosine Similarity von Haus aus so eine schnelle Operation ist, dass die Just-in-Time Compilation von Numba hier keinen großen Unterschied macht.

5. Durchführbarkeitsanalyse / Diskussion

Die Frage nach der Skalierbarkeit schließt sich eigentlich direkt an die Diskussion der möglichen Bottlenecks, und wie wir diese adressiert haben, an.
Die meiste Zeit in der Codeausführung brauchten wir einerseits beim Laden der Daten und der Feature Extraction (ca. 2h) und bei der Berechnung von t-SNE (ca. 4-5 min), was, da es sich um einmalige Prozesse in der Programmausführung handelt, ganz in Ordnung ist. Wäre der Datensatz 10 oder 10^2 so groß, würde aber auch die Ausführung viel länger dauern. Die „Main Limitation“ wäre dann wieder das Laden der Bilder und Extraktion/ Speicherung ihrer Features. In diesem Fall wäre es auf jeden Fall spannend, die Möglichkeiten, die wir unter 4. genannt haben, um den Code weiter zu optimieren, auszuprobieren. Mit dem aktuellen Datensatz könnte man unseren Image Recommender aber auf jeden Fall als Search Engine benutzen, die Ausgabe ähnlicher Bilder dauert für den Nutzer nur einige Sekunden.

Teil 2 – Big data image analysis

2A – Big data dimensionality reduction

Wir haben die Dimensionsreduktionsalgorithmen UMAP und t-SNE verwendet und damit die Position jedes Bildes in einem 2D Graph geplottet. Die große Anzahl an Bildern konnten wir mit UMAP und t-SNE verwenden, weil wir die Charakteristiken der Bilder schon zuvor in Embeddings gespeichert hatten.
Der User kann mit der Maus über die verschiedenen Punkte in den Plots hovern, bekommt so die ID angezeigt und kann sich diese Bilder dann plotten lassen.
Die Plots waren auf jeden Fall aufschlussreich. Wir wendeten auch noch Kmeans an, um die gefundenen Bilder in farblich markierten Gruppen zu sammeln. Dabei ist uns aufgefallen, dass Punkte, die nebeneinander liegen, auch zu Bildern gehören, die das gleiche zeigen (beispielsweise zeigte jedes der Bilder, die zu vier nebeneinander liegenden Punkten gehörten, Darstellungen von Ananas). Bei einem anderen Beispiel wählten wir drei Bilder, deren Punkte in der Nähe voneinander lagen und eines, das weiter weg im Plot war, aber nach KMeans zu der gleichen Gruppe gehörte, wie die anderen drei Bilder. Das Ergebnis war sehr interessant:

Abb. 2: Beispielbilder, die mit t-SNE / KMeans als ähnlich klassifiziert worden sind.

Das Bild mit der Katze und dem Schirm ist dabei das, dessen Punkt weiter weg von den anderen lag.

Daraus schließen wir, dass in dem t-SNE/UMAP Plot durch örtliche Nähe der Punkte relativ gut die Ähnlichkeit der Bilder angezeigt wird. Durch die Färbung der Punkte bei KMeans bekommen können wir außerdem noch Bilder finden, die (auch für Menschen erkennbar) nicht genau die Struktur zeigen, wie Bilder, die nebeneinander liegen, aber sehr ähnliche (wie die Katze).

Das zweitgrößte Bottleneck in unserem Code war die Dimensionsreduktion. PCA, t-SNE und UMAP sind vor allem bei „Big Data“ mit großem Rechenaufwand verbunden. Vor allem bei t-SNE ist uns dies aufgefallen.

Bei manchen Dimensionsreduktionstechniken (z.B. t-SNE) kann man Parameter verändern, um einen „trade-off“ zwischen Geschwindigkeit und Genauigkeit zu optimieren. Ein Beispiel hierfür ist der „perplexity value“. Eine Veränderung könnte die Berechnung beschleunigen, aber zu dem Preis, feinere Details in der Visualisierung zu verlieren. Genauso verhält es sich mit einer Anpassung der learning rate.

Außerdem können Dimensionsreduktionen durch „Approximation techniques“ optimiert werden.
Ein Ansatz ist, randomisierte Versionen der Dimensionsreduktionsalgorithmen zu verwenden (z.B. Randomized PCA), bei denen nicht der klassische Algorithmus verwendet wird. Randomisierte Projektionen können schneller sein, aber auch weniger komplex und dadurch auch ungenauer.
Ein weiterer Ansatz ist, verkürzte Versionen der klassischen Dimensionsreduktionsalgorithmen zu verwenden. Auch dies würde die Dimensionsreduktion ungenauer machen.
Da in dem Datensatzgrößenbereich, in dem wir arbeiten, die Zeit für die Dimensionreduktion immer noch sehr praktikabel war, haben wir uns entschieden, bei den klassischen Algorithmen zu bleiben und somit die Genauigkeit der Dimensionsreduktion zu erhalten.

Da wir nicht genau wussten, wie viele Labels wir in dem Datensatz eigentlich haben, konnten wir auch für die Label bei KMeans keine genaue Zahl angeben. Wir haben es mit 50 und 100 Labeln versucht, und damit auch schon relativ genaue Gruppen identifizieren können. Wahrscheinlich sind dadurch aber auch Vorhersagen wie die in Abbildung 2 entstanden, bei denen zwar ähnliche Strukturen auf den Bildern zu erkennen sind, für einen Menschen aber klar sichtbar ist, dass es sich nicht um das gleiche Motiv handelt. Dies ließe sich noch weiter optimieren, wenn wir die genaue Anzahl der Klassen in dem Datensatz kennen würden,

wahrscheinlich wurden die Katze und die Blitze in die gleiche Gruppe eingeordnet, weil wir bei Aufruf des KMeans Algorithmus eine zu kleine Anzahl an Klassen angegeben haben.

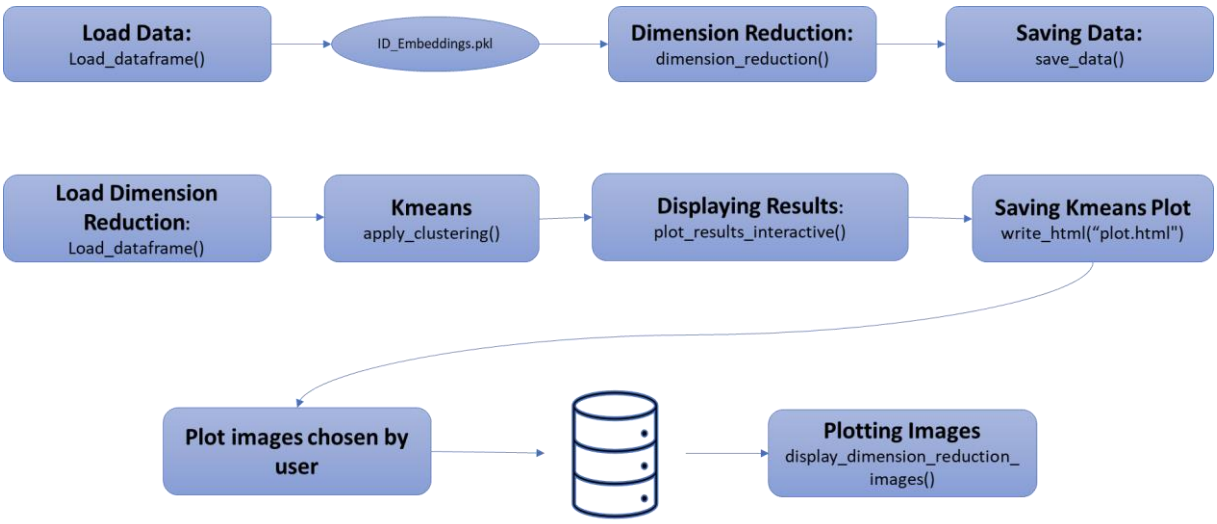

Abb. 3: Flowchart Dimensionsreduktion. Die Pfeile verdeutlichen, in welcher Reihenfolge die verschiedenen Programmbausteine miteinander interagieren.