ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΚΡΟΙΤΟΡ ΚΑΤΑΡΤΖΙΟΥ ΙΩΑΝ Π21077

ΑΛΕΞΙΟΣ ΒΑΣΙΛΕΙΟΥ Π21009

ΡΟΥΤΣΗΣ ΑΛΕΞΙΟΣ Π21145

ΕΡΓΑΣΙΑ ΕΞΑΜΗΝΟΥ ΜΑΘΗΜΑΤΟΣ ΑΝΑΛΥΣΗ ΕΙΚΟΝΑΣ

ΠΕΙΡΑΙΑΣ

Φεβρουάριος 2024

ΠΡΟΛΟΓΟΣ

Η παρούσα εργασία αναπτύχθηκε ως μέρος του μαθήματος Ανάλυση Εικόνας, με κύριο στόχο τη ανάπτυξη γραφοθεωρητικών αλγορίθμων για την ανάκτηση εικόνων με βάση το περιεχόμενο.

ΕΚΦΩΝΗΣΗ

Θέμα

Στόχος της συγκεκριμένης υπολογιστικής εργασίας είναι ανάπτυξη γραφοθεωρητικών αλγορίθμων για την ανάκτηση εικόνων με βάση το περιεχόμενο. Τα βασικά βήματα της προτεινόμενης αλγοριθμικής προσέγγισης έχουν ως ακολούθως:

1. Κανονικοποίηση Σειράς Κατάταξης (**Rank Normalization**)
2. Κατασκευή Υπεργράφου (**Hypergraph Construction**)
3. Υπολογισμός Ομοιότητας Υπερακμών (**Hyperedge Similarities**)
4. Υπολογισμός Καρτεσιανού Γινομένου μεταξύ των στοιχείων των Υπερακμών (**Cartesian Product of Hyperedge Elements**)
5. Υπολογισμός Ομοιότητας βάσει του κατασκευασμένου Υπεργράφου (**Hypergraph – Based Similarity**)

Λεπτομερής περιγραφή της παραπάνω αλγοριθμικής διαδικασίας μπορείτε να βρείτε στο άρθρο με τίτλο “**Multimedia Retrieval through Unsupervised Hypergraph-based Manifold Ranking**”, IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 28, NO. 12, DECEMBER 2019 . Το άρθρο θα αναρτηθεί στα έγγραφα του μαθήματος.

Ζητούμενα:

i. Να παρουσιάσετε μια αναλυτική περιγραφή της υπολογιστικής διαδικασίας που παρουσιάζεται στο άρθρο ενσωματώνοντάς την στην τεκμηρίωση της εργασίας σας. (**20% του συνολικού βαθμού**)
ii. Να αναπτύξετε κώδικα σε Matlab ή Python για την προγραμματιστική υλοποίηση των παραπάνω υπολογιστικών βημάτων. (**20% του συνολικού βαθμού**)
iii. Το σύνολο των αντικειμενικών χαρακτηριστικών για την διανυσματική αναπαράσταση της κάθε εικόνας να εξαχθεί με την χρήση ενός προεκπαιδευμένου νευρωνικού δικτύου (**squeezenet**, **googlenet**, **resnet18**, **resnet50**, **resnet101**, κλπ.). Συγκεκριμένα, μπορείτε να χρησιμοποιήσετε ως αντικειμενικά χαρακτηριστικά της κάθε εικόνας έξοδο που αντιστοιχεί σε κάποιο από τα ενδιάμεσα κρυφά επίπεδα των προαναφερθέντων νευρωνικών δικτύων. (**20% του συνολικού βαθμού**)
iv. Να παρουσιάσετε παραδείγματα της ορθής εκτέλεσης του κώδικάς σας. Χαρακτηρίστε κάποιες από τις εικόνες της βάσης ως εικόνες στόχο (**target images**) και παρουσιάστε μια λίστα με τις συναφέστερες εικόνες της βάσης. (**20% του συνολικού βαθμού**)
v. Προτείνετε μια συστηματική διαδικασία για την μέτρηση της ακρίβειας του συγκεκριμένου αλγορίθμου και παρουσιάσετε τα αποτελέσματά της. (**20% του συνολικού βαθμού**)

ΠΕΡΙΕΧΟΜΕΝΑ

1. ΕΙΣΑΓΩΓΗ

1.1 ΣΤΟΧΟΙ ΕΡΓΑΣΙΑΣ

Βασικός στόχος της εργασίας είναι η εξερεύνηση της οπτικής ομοιότητας χρησιμοποιώντας σύγχρονες τεχνικές μηχανικής μάθησης για την εξαγωγή πληροφορίας υψηλού επιπέδου από τα δεδομένα εικόνων.

2. ΠΕΡΙΓΡΑΦΗ ΛΥΣΗΣ

2.1 Περιγραφή Υπολογιστικής Διαδικασίας

Αυτή βρίσκεται στο αρχείο *i_greek_ltx.pdf* για την αναλυτική εκδοχή στα ελληνικά (ορισμένοι όροι μπορεί να μην έχουν μεταφραστεί με απόλυτη ακρίβεια, π.χ. manifold καθώς δεν υπάρχει ακριβέστατη μετάφραση), και μία συντομότερη εκδοχή στα αγγλικά, στο αρχείο *i_english_ltx.pdf*.

2.2 Προγραμματιστική Υλοποίηση

Η υλοποίηση έγινε σε Python και παρακάτω θα εξηγηθεί η υλοποίηση σε κώδικα του αλγορίθμου εύρεσης σχετικών εικόνων. Το σχετικό script είναι το manifold_ranking.py. Όσον αφορά την υλοποίηση ακολούθησε την σειρά και την δομή που περιγράφεται στο άρθρο με τίτλο "**Multimedia Retrieval through Unsupervised Hypergraph-based Manifold Ranking**", IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 28, NO. 12, DECEMBER 2019 , ενώ τυχόν **αλλαγές ή διαφορετικές επιλογές από το άρθρο που οδήγησαν σε καλύτερη απόδοση (τουλάχιστον στη δική μου περίπτωση και στο συγκεκριμένο dataset που χρησιμοποίηση), αναφέρονται τόσο σε αυτήν όσο και σε επόμενη ενότητα. Να σημειωθεί πως όλη η εργασία και όλε οι εκδοχές που οδήγησαν στην τελική της μορφή υπάρχουν στο GitHub Repository του Π21077:** https://github.com/ioannisCC/image-analysis .

2.2.1 Βήμα 1: Υπολογισμός αποστάσεων και ομοιότητας

Αυτό το κομμάτι του κώδικα έχει ως στόχο τον υπολογισμό των αποστάσεων και ομοιοτήτων μεταξύ εικόνων ενός συνόλου δεδομένων, καθώς και τη δημιουργία ενός πίνακα κατάταξης με βάση την ομοιότητα. Αρχικά, υπολογίζεται ένας πίνακας αποστάσεων χρησιμοποιώντας την **απόσταση συνημιτόνου (cosine distance)** ανάμεσα στα χαρακτηριστικά των εικόνων. Στη συνέχεια, η απόσταση αυτή μετατρέπεται σε τιμή ομοιότητας μέσω exponential kernel, και οι εικόνες ταξινομούνται βάση τιμών ομοιότητας.

```
# --- step 1: compute distances and similarities ---

# calculate the distance of each images with all the training images using cosine distance
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html
def calculate_distances(features, features_array):
    return cdist(features, features_array, metric='cosine')
```

Συγκεκριμένα, η συνάρτηση calculate_distances(features, features_array) υπολογίζει τις αποστάσεις συνημίτονου (**cosine distances**) μεταξύ όλων των ζευγών εικόνων. Η απόσταση

συνημίτονου (**cosine distance**) μετρά τη διαφορά γωνίας μεταξύ δύο διανυσμάτων χαρακτηριστικών, ανεξάρτητα από το μέγεθός τους. Ο πίνακας αποστάσεων που επιστρέφεται έχει διάσταση **n x n**, όπου **n** είναι ο αριθμός των εικόνων, και κάθε στοιχείο **d(i, j)** αντιπροσωπεύει την απόσταση μεταξύ των εικόνων **i** και **j**.

Σημαντικό να σημειωθεί πως δοκιμάστηκε και η Ευκλείδεια απόσταση (**Euclidean distance**), η οποία μετρά την πραγματική απόσταση στο χώρο μεταξύ δύο διανυσμάτων. Ωστόσο, συνολικά παρουσίασε χειρότερη απόδοση στον αλγόριθμο, καθώς δεν κατάφερε να αποτυπώσει αποτελεσματικά τις ομοιότητες των εικόνων σε υψηλές διαστάσεις χαρακτηριστικών.

```
# compute the distance matrix
distance_matrix = calculate_distances(features_array, features_array)
print(f'Distance Matrix (first 10 rows):\n{distance_matrix[:10]}\n')
```

Στη συνέχεια, ο πίνακας αποστάσεων μετατρέπεται σε πίνακα ομοιότητας **(similarity matrix)** μέσω της σχέσης **s(i, j) = e^(-d(i, j))**, όπου μικρότερες αποστάσεις (**distances**) αντιστοιχούν σε μεγαλύτερες τιμές ομοιότητας (**similarity values**). Έτσι, προκύπτει ο πίνακα ομοιότητας (**similarity matrix**) ίδιων διαστάσεων, κάθε στοιχείο του οποίου αντιπροσωπεύει πόσο όμοιες είναι δύο εικόνες.

Δοκιμάστηκε επίσης και ο πυρήνας Gauss (**Gaussian kernel**), ο οποίος βασίζεται στη χρήση της κανονικής κατανομής για τη μετατροπή των αποστάσεων σε ομοιότητες. Ωστόσο, ο εκθετικός πυρήνας (**exponential kernel**) έδειξε καλύτερη συνολική απόδοση στον αλγόριθμο, επιτυγχάνοντας πιο ακριβή αποτελέσματα στις κατατάξεις των εικόνων.

```
# initial ranking order (indices in descending order of similarity)
initial_ranks = np.argsort(similarity_matrix, axis=1)[:, ::-1]
print(f'Initial Ranks (first 10 rows):\n{initial_ranks[:10]}\n')


# initial ranking order values (values in descending order of similarity)
initial_ranks_values = np.sort(similarity_matrix, axis=1)[:, ::-1]
print(f'Initial Sorted Similarity Values (first 10 rows):\n{initial_ranks_values[:10]}\n')
```

Για την κατάταξη των εικόνων (**image ranking**), ο πίνακας ομοιότητας (**similarity matrix**) ταξινομείται σε φθίνουσα σειρά βάση των τιμών ομοιότητας (**similarity values**). Η συνάρτηση *np.argsort(similarity_matrix, axis=1)[:, ::-1]* ταξινομεί τους δείκτες των εικόνων σε κάθε γραμμή του πίνακα ομοιότητας (**similarity matrix**) σε **φθίνουσα σειρά**.

- *np.argsort(similarity_matrix, axis=1):* Ταξινομεί τις τιμές ομοιότητας κατά μήκος κάθε γραμμής, αλλά σε αύξουσα σειρά (από την μικρότερη στην μεγαλύτερη).
- *[:, ::-1]:* Αντιστρέφει την κατάταξη ώστε να γίνει φθίνουσα (από την μεγαλύτερη ομοιότητα στην μικρότερη).

Κάθε γραμμή του πίνακα κατάταξης (**ranking matrix**) δείχνει τη σειρά με την οποία οι εικόνες ταξινομούνται με βάση την ομοιότητά τους σε σχέση με την εικόνα-αναφορά (**query image**).

```
# initial ranking order values (values in descending order of similarity)
initial_ranks_values = np.sort(similarity_matrix, axis=1)[:, ::-1]
print(f'Initial Sorted Similarity Values (first 10 rows):\n{initial_ranks_values[:10]}\n')
```

Τέλος, οι τιμές ομοιότητας (**similarity values**) ταξινομούνται όμοια με το προηγούμενο βήμα.

2.2.2 Βήμα 2: Κανονικοποίηση Σειράς Κατάταξης

Ο σκοπός αυτής της διαδικασίας είναι να βελτιώσει τη διαδικασία κατάταξης χρησιμοποιώντας μια μορφή κανονικοποίησης βάση **αμοιβαίων θέσεων κατάταξης (reciprocal ranks)**. Αυτό γίνεται με την ανακατασκευή των κατατάξεων έτσι ώστε να λαμβάνονται υπόψη όχι μόνο οι άμεσες αλλά και οι αμοιβαίες ομοιότητες μεταξύ εικόνων.

```
# --- step 2: rank normalization ---

def rank_normalization(ranks, L): # L are top positions to consider
    num_items = ranks.shape[0]

    # reciprocal rank positions
    reciprocal_ranks = 1.0 / (ranks + 1)

    # Compute the new similarity measure ⇢n
    rho_n = np.zeros((num_items, num_items))
    for i in range(num_items):
        for j in range(num_items):
            rho_n[i, j] = reciprocal_ranks[i, j] + reciprocal_ranks[j, i]
                        # rho_n[i, j] = 2 * L - (ranks[i, j] + ranks[j, i])


    # update the top-L positions based on the new similarity measure
    updated_ranks = np.argsort(-rho_n, axis=1)[:, :L]

    return updated_ranks

L = 5
normalized_ranks = rank_normalization(initial_ranks, L)
print(f'Normalized Ranks (first 10 rows):\n{normalized_ranks[:10]}\n')
```

Η διαδικασία ξεκινάει με τη συνάρτηση rank_normalization(ranks, L), η οποία λαμβάνει ως είσοδο τον πίνακα αρχικών κατατάξεων (**initial ranks**) και τον αριθμό **L**, που αντιπροσωπεύει τις top θέσεις (εικόνες) που θα ληφθούν υπόψη. Το πρώτο βήμα είναι ο υπολογισμός των **αμοιβαίων θέσεων κατάταξης (reciprocal rank positions)** μέσω της σχέσης *reciprocal_ranks = 1.0 / (ranks + 1)*. Αυτή η προσέγγιση εξασφαλίζει ότι οι πρώτες θέσεις έχουν υψηλές τιμές αμοιβαίας κατάταξης, ενώ οι χαμηλότερες θέσεις επηρεάζουν λιγότερο το αποτέλεσμα.

Ο υπολογισμός του νέου μέτρου ομοιότητας πραγματοποιείται μέσω του πίνακα *rho_n*, ο οποίος είναι διαστάσεων **n x n**, όπου **n** είναι ο αριθμός των εικόνων. Για κάθε ζεύγος εικόνων **i** και **j**, η νέα τιμή ομοιότητας προκύπτει από το άθροισμα των αμοιβαίων θέσεων κατάταξης: *rho_n[i, j] = reciprocal_ranks[i, j] + reciprocal_ranks[j, i]*. Αυτή η μέτρηση λαμβάνει υπόψη τόσο τη σχετικότητα της εικόνας **j** σε σχέση με την **i**, όσο και το αντίστροφο. Να σημειωθεί ότι δοκιμάστηκε και η μέθοδος του άρθρου ρ_n(i, j) = 2L - (τ_i(j) + τ_j(i)), αλλά είχε χειρότερη απόδοση.

Στη συνέχεια, η ταξινόμηση των εικόνων πραγματοποιείται όμοια με το την αρχική ταξινόμηση. Το αποτέλεσμα είναι ο πίνακας *updated_ranks*, ο οποίος περιέχει τις κορυφαίες **L** εικόνες που έχουν την υψηλότερη αμοιβαία ομοιότητα με την εικόνα αναφοράς **(query image)**.

2.2.3 Βήμα 3: Κατασκευή Υπεργράφου

Αυτό το κομμάτι του κώδικα έχει ως στόχο την κατασκευή ενός **υπεργράφου (hypergraph)**, το οποίο αναπαριστά τη σύνδεση των εικόνων μεταξύ τους με βάση την ομοιότητά τους.

```
# --- step 3: build hypergraph with weights ---

# logarithmic decay to assign weights
def calculate_wp(i, x, initial_ranks, k):
    # i: item of which the weight is calculated
    # x: item of which weight we want to calculate
    # k: top items to retrieve (controls rate of decay)
    tau_ix = np.where(initial_ranks[i] == x)[0][0] + 1  # Position of x in τi
    return 1 - np.log(tau_ix) / np.log(k)

# represent the incidence matrix
def build_hypergraph(features_array, initial_ranks, k):
    n = len(features_array)
    H = np.zeros((n, n))
    for i in range(n):
        # get the top-k neighbors for image i
        N_i = initial_ranks[i, :k]
        for j in range(n):
            if i == j:
                continue
            r_sum = 0
            # for each neighbor x in N(i,k)
            for x in N_i:
                N_x = initial_ranks[x, :k]
                if j in N_x:
                    try:
```

```
                    wp_ix = calculate_wp(i, x, initial_ranks, k)
                    wp_xj = calculate_wp(x, j, initial_ranks, k)
                    r_sum += wp_ix * wp_xj
                except Exception as e:
                    print(f"Error at i={i}, j={j}, x={x}: {e}")
            H[i, j] = r_sum
            if r_sum > 0:
                print(f"Non-zero weight: H[{i},{j}] = {r_sum}")
    return H

k = 5 # top neighbors to consider
H = build_hypergraph(features_array, initial_ranks, k)

print(f"\nMatrix H statistics:")
print(f"Shape: {H.shape}")
print(f"Non-zero elements: {np.count_nonzero(H)}")
print(f"Max value: {np.max(H)}")
print(f"First 5 rows of H:\n{H[:5, :5]}\n")
```

Η διαδικασία ξεκινά με τον υπολογισμό των βαρών μεταξύ των εικόνων χρησιμοποιώντας λογαριθμική απόσβεση (**logarithmic decay**). Η συνάρτηση *calculate_wp(i, x, initial_ranks, k)* υπολογίζει το βάρος της σύνδεσης μεταξύ της εικόνας **i** και της εικόνας **x**, λαμβάνοντας υπόψη τη θέση της **x** στη λίστα κατάταξης της **i**. Το βάρος δίνεται από τη σχέση 1 - (log(τ_i(x)) / log(k)), όπου **τ_i(x)** είναι η θέση του **x** στη λίστα κατάταξης του **i**. Αυτό διασφαλίζει ότι οι γείτονες που βρίσκονται κοντά στην κορυφή της λίστας λαμβάνουν μεγαλύτερο βάρος, ενώ οι πιο απομακρυσμένοι γείτονες έχουν μικρότερη επιρροή, διαδικασία η οποία συμπίπτει με αυτή του άρθρου.

Η κατασκευή του υπεργράφου πραγματοποιείται στη συνάρτηση *build_hypergraph(features_array, initial_ranks, k)*, η οποία ξεκινά με τη δημιουργία ενός πίνακα συνάφειας **H** με αρχικές τιμές μηδέν. Για κάθε εικόνα **i**, βρίσκονται οι κορυφαίοι **k** γείτονές της από τη λίστα **initial_ranks**. Στη συνέχεια, ελέγχεται αν κάποιος γείτονας **j** της εικόνας **i** έχει κοινό γείτονα με την εικόνα **i**. Αν ναι, τότε το συνολικό βάρος της σύνδεσης υπολογίζεται από το γινόμενο των βαρών wp(i, x) και wp(x, j), όπου **x** είναι ο κοινός γείτονας. Το άθροισμα των βαρών αποθηκεύεται στο στοιχείο **H[i, j]** του πίνακα συνάφειας, το οποίο υποδηλώνει τη σύνδεση μεταξύ των εικόνων **i** και **j**, μία διαδικασία η οποία και πάλι συμπίπτει με αυτή του άρθρου.

2.2.4 Βήμα 4: Υπολογισμός Ομοιότητας Υπερακμών

Ο κώδικας αυτός ασχολείται με τον υπολογισμό διαφορετικών πινάκων ομοιότητας βάσει του υπεργράφου (**hypergraph**) για την κατασκευή ενός τελικού πίνακα ομοιότητας.

```
# --- step 4: hypergraph-based similarity matrices ---

# hyperedge-based similarity matrix (Sh)
Sh = np.dot(H, H.T)
print("Sh (Hyperedge-based Similarity Matrix):")
print(Sh)

# vertex-based similarity matrix (Sv)
Sv = np.dot(H.T, H)
print("Sv (Vertex-based Similarity Matrix):")
print(Sv)

# combine the two similarities using Hadamard (elementwise) product
S = np.multiply(Sh, Sv)
print("Combined Similarity Matrix S (Hadamard product):")
print(S)
```

Η διαδικασία ξεκινά με τον υπολογισμό του **Sh (Hyperedge-based Similarity Matrix)** που υπολογίζει την ομοιότητα μεταξύ των εικόνων λαμβάνοντας υπόψη τις κοινές υπερακμές τους. Ουσιαστικά, αν δύο εικόνες συμμετέχουν σε πολλές κοινές υπερακμές (δηλαδή έχουν κοινά γειτονικά σύνολα), η τιμή ομοιότητας τους στον **Sh** θα είναι υψηλή. Ο πίνακας αυτός δίνεται από τη σχέση: **Sh = H * H.T**.

Στη συνέχεια, υπολογίζεται ο **Sv (Vertex-based Similarity Matrix)** που υπολογίζει την ομοιότητα μεταξύ των εικόνων με βάση τις κοινές κορυφές που συνδέονται μεταξύ τους μέσω γειτόνων. Εδώ, δύο εικόνες έχουν υψηλή ομοιότητα στον **Sv** αν οι γείτονές τους (κορυφές) είναι κοινοί. Ο υπολογισμός αυτός δίνεται από τη σχέση: **Sv = H.T * H**.

Ο τελικός πίνακας ομοιότητας **S** προκύπτει από το **γινόμενο Hadamard (Hadamard product)** των δύο πινάκων **Sh** και **Sv**, δηλαδή πραγματοποιόντας στοιχειο-προς-στοιχείο **(element-wise)** πολλαπλασιασμό των δύο πινάκων, έτσι ώστε να λαμβάνονται υπόψη τόσο οι κοινές υπερακμές όσο και οι κοινές κορυφές των εικόνων.

Τα παραπάνω βήματα έγιναν όπως αναγράφονται στο άρθρο.

2.2.5 Βήμα 5: Υπολογισμός Καρτεσιανού Γινομένου μεταξύ των στοιχείων των Υπερακμών & Υπολογισμός Ομοιότητας βάσει του κατασκευασμένου Υπεργράφου

Αυτό το μέρος του κώδικα ασχολείται με τον υπολογισμό του τελικού πίνακα συσχέτισης **(affinity matrix W)**, ο οποίος αναπαριστά τη συνολική ισχύ των συνδέσεων μεταξύ εικόνων, λαμβάνοντας υπόψη τόσο τις τοπικές **(local)** όσο και τις συνολικές **(global)** σχέσεις που έχουν ήδη προκύψει από τα προηγούμενα βήματα.

```
# --- step 5: cartesian product ---
```

```
# compute hyperedge weights by summing each row of H
w = np.sum(H, axis=1)
H_sparse = sp.csr_matrix(H)

# use sparse matrix operations since H is sparse

#  Cartesian product matrix C
C = (H_sparse.T.multiply(w)) @ H_sparse
# combine with S to form final affinity matrix W
W = S * C.toarray()
print("Final Affinity Matrix W:")
print(W)
```

Ξεκινά με τον υπολογισμό των βαρών των υπερακμών μέσω του αθροίσματος κάθε γραμμής του πίνακα **H**: *w = np.sum(H, axis=1)*. Το κάθε στοιχείο **w[i]** αντιπροσωπεύει το συνολικό βάρος των συνδέσεων της υπερακμής που σχετίζεται με την εικόνα **i**. Στη συνέχεια, ο πίνακας **H** μετατρέπεται σε μία **αραιή μήτρα (sparse matrix)** μέσω της συνάρτησης *csr_matrix* της βιβλιοθήκης **scipy**: *H_sparse = sp.csr_matrix(H)*. Αυτή η προσέγγιση είναι κρίσιμη για την αποδοτική αποθήκευση και επεξεργασία δεδομένων σε περιπτώσεις όπου οι πίνακες είναι αραιοί (δηλαδή περιέχουν πολλά μηδενικά).

Το επόμενο βήμα είναι ο υπολογισμός του **καρτεσιανού γινομένου (Cartesian product)**, που αποτυπώνει τις συσχετίσεις μεταξύ των εικόνων βάσει των κοινώω τους υπερακμών, μέσω της σχέσης *C = (H_sparse.T.multiply(w)) @ H_sparse*. Το αποτέλεσμα, ο πίνακας **C**, εκφράζει τη συνολική συνάφεια μεταξύ των κορυφών που ανήκουν στις ίδιες υπερακμές και έχουν κοινούς γείτονες. Αυτό σημαίνει ότι αν δύο εικόνες συνδέονται συχνά μέσω κοινών γειτόνων, το βάρος της σχέσης τους στον πίνακα **C** θα είναι υψηλό.

Ο τελικός πίνακας συσχέτισης **W** υπολογίζεται μέσω του γινομένου Hadamard μεταξύ του πίνακα **C** με τον πίνακα ομοιότητας **S**: *W = S * C.toarray()*. Με αυτόν τον τρόπο, οι αρχικές τιμές ομοιότητας μεταξύ εικόνων ενισχύονται από το καρτεσιανό γινόμενο, το οποίο λαμβάνει υπόψη τόσο τις τοπικές όσο και τις γενικές συσχετίσεις που προκύπτουν από τη δομή του υπεργράφου. Το αποτέλεσμα είναι ένας πίνακας που περιέχει τις τελικές τιμές συσχέτισης μεταξύ όλων των εικόνων, επιτρέποντας την αποδοτική κατηγοριοποίηση και ανάκτηση εικόνων.

Τα παραπάνω βήματα έγιναν όπως αναγράφονται στο άρθρο.

2.2.5 Βήμα 6: Επαναληπτική Κατάταξη

Αυτό το βήμα του κώδικα αφορά την επαναληπτική κατάταξη **(iterative ranking)**, η οποία στοχεύει στη σταδιακή βελτίωση των κατατάξεων των εικόνων μέσω της συνεχούς χρήσης του πίνακα **W**.

```
# --- step 6: iterative ranking ---
```

```
# refine rankings based on W
# iterations = 2
# W_t = W.copy()  # initial W(0)
# T = initial_ranks.copy()  # initial T(0)

# for t in range(iterations):
#     # update T(t) based on current W(t)
#     T = np.argsort(W_t, axis=1)[:, ::-1]
#     print(f"Iteration {t+1} updated rankings (first 10 rows):")
#     print(T[:10])

#     # reconstruct hypergraph using new rankings T to get W(t+1)
#     # build new hypergraph with new rankings
#     H = build_hypergraph(features_array, T, k)

#     # compute new Sh and Sv
#     Sh = np.dot(H, H.T)
#     Sv = np.dot(H.T, H)
#     S = np.multiply(Sh, Sv)

#     # compute new Cartesian product
#     w = np.sum(H, axis=1)
#     H_sparse = sp.csr_matrix(H)
#     C = (H_sparse.T.multiply(w)) @ H_sparse

#     # update W(t+1)
#     W_t = S * C.toarray()

# # final W_t back to W
# W = W_t.copy()
```

Ξεκινά με τον αρχικό πίνακα συσχέτισης **W(0)** και τις αρχικές κατατάξεις **T(0)**. Σε κάθε επανάληψη t, οι κατατάξεις **T(t)** ενημερώνονται με βάση τον τρέχοντα πίνακα συσχέτισης W(t) μέσω της ταξινόμησης των εικόνων σε φθίνουσα σειρά συσχέτισης με τη συνάρτηση *np.argsort(W_t, axis=1)[:, ::-1]*. Με βάση τις νέες κατατάξεις **T**, κατασκευάζεται ένας νέος πίνακας **H** χρησιμοποιώντας τη συνάρτηση *build_hypergraph*, ο οποίος επαναπροσδιορίζει τις σχέσεις γειτνίασης μεταξύ εικόνων.

Μετά επαναλαμβάνονται τα βήματα 4 και 5 και νέος πίνακας συσχέτισης **W(t+1)** ενημερώνεται μέσω του πολλαπλασιασμού **S * C**, που ενσωματώνει τις βελτιωμένες σχέσεις μεταξύ των εικόνων. Η διαδικασία αυτή επαναλαμβάνεται δύο φορές **(iterations = 2)**,

επιτρέποντας στις κατατάξεις να προσαρμόζονται σταδιακά καθώς ανανεώνονται οι σχέσεις μεταξύ εικόνων.

Παρόλο που υλοποιήθηκε αυτό το βήμα στον τελικό αλγόριθμο δεν χρησιμοποιήθηκε (για αυτό είναι με σχόλια), αφού παρατηρήθηκε ότι χρησιμοποιώντας τον η απόδοση έπεφτε στο ½ σε σχέση με την απόδοση που είχε χωρίς αυτόν.

Έτσι ολοκληρώνεται ο βασικός αλγόριθμος ανάκτησης εικόνων.

2.3 Εξαγωγή αντικειμενικών χαρακτηριστικών

Για την εξαγωγή των αντικειμενικών χαρακτηριστικών κάθε εικόνας, χρησιμοποιείται το προεκπαιδευμένο νευρωνικό δίκτυο **ResNet50** από το **torchvision.models**. Το δίκτυο αυτό έχει εκπαιδευτεί σε ένα μεγάλο σύνολο δεδομένων (ImageNet), και εξάγει διανυσματικές αναπαραστάσεις εικόνων μέσω των κρυφών επιπέδων του. Συγκεκριμένα, αφαιρείται το τελευταίο επίπεδο ταξινόμησης (**fully connected layer, fc**) ώστε να διατηρηθεί το επίπεδο εξαγωγής χαρακτηριστικών από το προτελευταίο επίπεδο **(embedding layer)**. Η κίνηση αυτή εξασφαλίζει ότι αντί να έχουμε έξοδο μια ταξινόμηση σε κλάσεις, παίρνουμε ένα διανυσματικό χαρακτηριστικό που περιγράφει την εικόνα. Να σημειωθεί πως αρχικά, δοκιμάστηκε με **ResNet18** αλλά έδειξε χειρότερη απόδοση, προφανώς λόγω των λιγότερων κρυφών επιπέδων που διαθέτει (παρόλα αυτά η διαφορά δεν ήταν πολύ μεγάλη πράγμα που μπορεί να σημαίνει πως ίσως σε ορισμένες περιπτώσεις είναι προτιμότερο η χρήση του αφού τα χαρακτηριστικά που εξάγονται από αυτό είναι μικρότερο σε μέγεθος, με παρόμοια απόδοση).

```
# path to 'list.txt' file
list_file = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))), "data", "annotations",
"list.txt")

image_info = []

with open(list_file, 'r') as file:
    for line in file:
        # skip the header lines
        if line.startswith("#"):
            continue

        # split the line into parts (separated by spaces)
        parts = line.strip().split()

        # ensure that lines are consistent
        if len(parts) == 4:
            image_id = parts[0]  # image id (e.g., Abyssinian_100)
```

```
            class_id = int(parts[1])  # class id (1: Cat, 2: Dog)
            species = 1 if image_id[0].isupper() else 2  # 1 for cat, 2 for dog
            breed_id = int(parts[3])  # breed id

            image_info.append({
                'image_id': image_id,
                'class_id': class_id,
                'species': species,
                'breed_id': breed_id
            })

# convert the list of dictionaries into a DataFrame
image_info_df = pd.DataFrame(image_info)
print(image_info_df.head())

# load the pre-trained model
model = resnet50(pretrained=True)

# evaluate the model
model.eval()

# remove the last layer (the classification layer, fc)
model = torch.nn.Sequential(*(list(model.children())[:-1]))
model.eval()

# apply the same transformations as the ones used during training
# https://pytorch.org/vision/0.18/models/generated/torchvision.models.resnet50.html

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])


def extract_features(image_path, model, transform):
    try:
        img = Image.open(image_path).convert('RGB') # ensure RGB format
        image_tensor = transform(img).unsqueeze(0)
```

```
        with torch.no_grad():
            features = model(image_tensor)
        return features.squeeze(0).numpy()
    except Exception as e:
        print(f"Error processing image {image_path}: {e}")
        return None

# training images
image_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))), "data", "images")

all_features = []

# for each entry in the data frame
for _, row in image_info_df.iterrows(): # iterrows() allow to iterate over the rows of a df
    image_name = row['image_id'] + '.jpg'
    image_path = os.path.join(image_dir, image_name)

    # extract features
    features = extract_features(image_path, model, transform)

    if features is not None:
        all_features.append({
            'features': features,
            'image_id': row['image_id'],
            'class_id': row['class_id'],
            'species': row['species'],
            'breed_id': row['breed_id']
        })
    else:
        print(f'Image {image_name} has no extracted features')

# convert list to numpy array
features_array = np.array([item['features'] for item in all_features])

print("Number of all feature entries: ", len(all_features))
print("Features shape: ", features_array.shape)
print("Number of feature entries: ", len(all_features))
if len(all_features) > 0:
    print("Shape of a single feature vector: ", all_features[0]['features'].shape)

for feature in all_features:
```

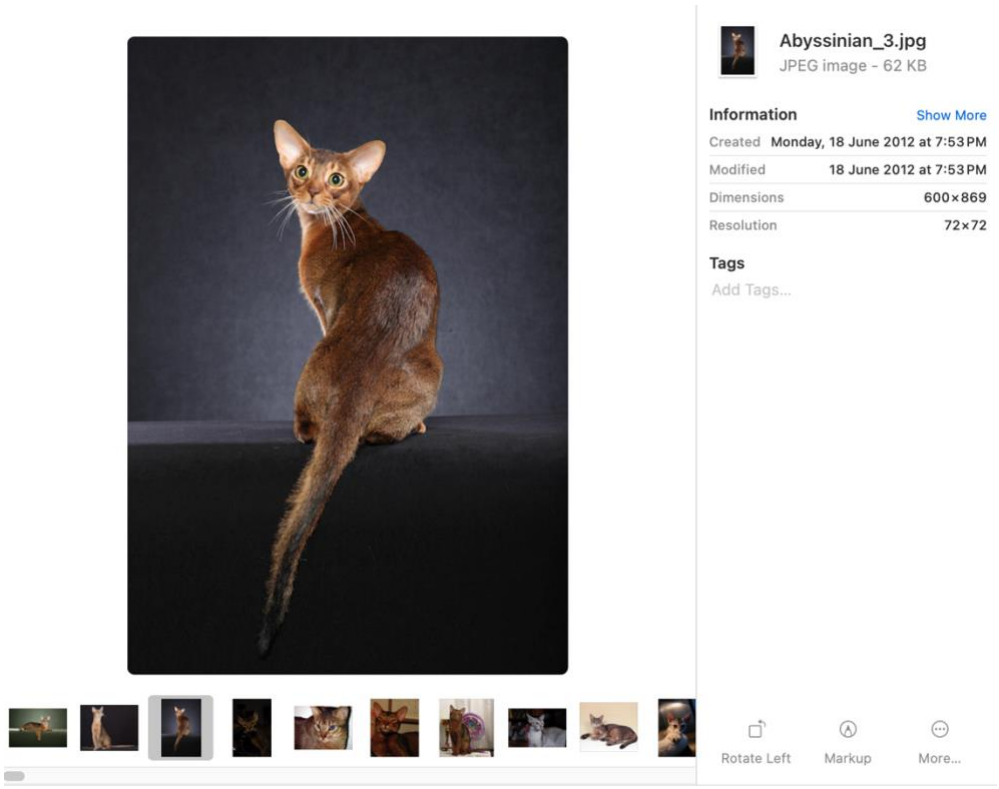
```
    feature['features'] = feature['features'].reshape(-1)  # flatten to 1d dictionary


print("Sample metadata with reshaped features:", all_features[0])
print(len(all_features))
print("Shape of reshaped feature vector:", all_features[0]['features'].shape)
```

Αρχικά, το σύνολο των δεδομένων διαβάζεται από το αρχείο **list.txt** και αποθηκεύεται σε ένα DataFrame, το οποίο περιέχει πληροφορίες όπως το **image_id**, το **class_id** (γάτα ή σκύλος), το **species** (τύπος) και το **breed_id** (φυλή). Για κάθε εικόνα που διαβάζεται, εφαρμόζονται μετασχηματισμοί, ώστε να ταιριάζουν με τη μορφή που απαιτεί το προεκπαιδευμένο δίκτυο **ResNet50**. Οι μετασχηματισμοί αυτοί υλοποιούνται με τη χρήση της *torchvision.transforms.Compose.*

Η διαδικασία εξαγωγής των χαρακτηριστικών πραγματοποιείται μέσω της συνάρτησης *extract_features(image_path, model, transform)*, η οποία δέχεται την εικόνα, την εφαρμόζει τους κατάλληλους μετασχηματισμούς και παράγει τα χαρακτηριστικά χρησιμοποιώντας το προεκπαιδευμένο δίκτυο. Κάθε εικόνα μετατρέπεται σε διανυσματική αναπαράσταση και αποθηκεύεται σε έναν πίνακα **features_array**, ο οποίος έχει διαστάσεις **(# εικόνων, # χαρακτηριστικών)**. Επιπλέον, η λίστα **all_features** περιέχει εκτός από τα διανυσματικά χαρακτηριστικά, και metadata που σχετίζονται με κάθε εικόνα, όπως το **image_id**, το **class_id** κλπ. Άρα, η λίστα **all_features** λειτουργεί ως μια πλήρης αποθήκη δεδομένων, συνδυάζοντας τόσο τα αντικειμενικά χαρακτηριστικά όσο και τις απαραίτητες πληροφορίες για κάθε εικόνα.

2.3 Σύνολο Δεδομένων

Χρησιμοποιήθηκε το **Oxford-IIIT Pet Dataset**, που είναι ένα σύνολο δεδομένων που περιλαμβάνει εικόνες κατοικίδιων από 37 κατηγορίες (ράτσες), με περίπου 200 εικόνες ανά κατηγορία. Οι εικόνες παρουσιάζουν μεγάλη ποικιλία σε κλίμακα, στάση και φωτισμό, προσφέροντας μια πλούσια βάση για πειραματισμό σε προβλήματα υπολογιστικής όρασης. Κάθε εικόνα συνοδεύεται από σχολιασμούς (annotations) που περιλαμβάνουν την ταυτότητα της ράτσας κ.ά. Το αρχείο **list.txt** περιέχει τη λίστα όλων των εικόνων με πληροφορίες για το είδος (γάτα ή σκύλος), την κατηγορία και τη φυλή. Τα ονόματα των αρχείων που ξεκινούν με κεφαλαίο γράμμα υποδηλώνουν εικόνες γάτας, ενώ τα ονόματα με πεζά γράμματα υποδηλώνουν εικόνες σκύλου.



Εικόνα 1. Στιγμιότυπο από το σύνολο δεδομένων

2.4 Διαφορές στην παρούσα Υλοποίηση σε σχέση με αυτή του Άρθρο

- **Χρήση συνημιτονικής ομοιότητας (Cosine Similarity):** Αντί της μεθόδου ομοιότητας που προτείνεται στο άρθρο, η συνημιτονική ομοιότητα έδειξε σημαντικά καλύτερα αποτελέσματα στην ανάκτηση εικόνων, βελτιώνοντας την ακρίβεια της κατάταξης.
- **Αναπροσαρμογή της Κανονικοποίησης Κατάταξης (Ranking Normalization):** Η μέθοδος κανονικοποίησης κατάταξης που περιγράφεται στο άρθρο δεν απέδωσε καλά στην παρούσα υλοποίηση. Μετά από έρευνα, εφαρμόστηκε μια διαφορετική μέθοδος που παρείχε καλύτερα αποτελέσματα.
- **Μη χρήση μεθόδου Επαναληπτικής Κατάταξης (Iterative Ranking):** Η επαναληπτική μέθοδος που βασίζεται στην αναδόμηση του υπεργράφου και την ενημέρωση του W είχε ως αποτέλεσμα τη μείωση της ακρίβειας στο μισό σε σύγκριση με την απλούστερη υλοποίηση. Ως εκ τούτου, η επιλογή της μη επαναληπτικής προσέγγισης αποδείχθηκε πιο αποτελεσματική για την επίτευξη καλύτερων αποτελεσμάτων ανάκτησης.

Αυτές οι παρατηρήσεις και αλλαγές αφορούν την παρούσα υλοποίηση και το συγκεκριμένο σύνολο δεδομένων και δεν αποτελεί γενίκευση.

2.5 Μετρικές Μέτρησης Ακρίβειας Αλγορίθμου

```
# --- step 8: print results ---

# print all the results
top_k = 5
for i in range(W.shape[0]):
    # exclude the image itself and get top similar indices
    top_indices = [idx for idx in np.argsort(W[i])[-top_k*2:][::-1] if idx != i][:top_k]
    query_image_metadata = combined_df.iloc[i]
    print(f"Top {top_k} similar images for image {query_image_metadata['image_id']}:")
    print(f"  Class ID: {query_image_metadata['class_id']}, Species: {query_image_metadata['species']}, Breed ID:
{query_image_metadata['breed_id']}\n")
    for rank, idx in enumerate(top_indices):
        score = W[i, idx]
        similar_image_metadata = combined_df.iloc[idx]
        print(f"  {rank + 1}. Image ID: {similar_image_metadata['image_id']}, "
              f"Class ID: {similar_image_metadata['class_id']}, "
              f"Species: {similar_image_metadata['species']}, "
              f"Breed ID: {similar_image_metadata['breed_id']}, "
              f"Score: {score:.4f}")
    print("=" * 50)


# query a specific image
query_image_index = 371  # by index
# or
query_image_id = "Abyssinian_100"  # by id

# retrieve the row corresponding to the query image
if query_image_index is not None:
    query_image_metadata = combined_df.iloc[query_image_index]
elif query_image_id is not None:
    query_image_metadata = combined_df[combined_df['image_id'] == query_image_id].iloc[0]
else:
    raise ValueError("Please provide either query_image_index or query_image_id.")

# get top-k most similar images
```

```
top_k = 5
query_image_index = query_image_metadata.name
# exclude itself from the similar images
top_indices = [idx for idx in np.argsort(W[query_image_index])[-top_k*2:][::-1] if idx != query_image_index][:top_k]

print(f"Top {top_k} similar images for query image {query_image_metadata['image_id']}:")
print(f"  Class ID: {query_image_metadata['class_id']}, Species: {query_image_metadata['species']}, Breed ID:
{query_image_metadata['breed_id']}\n")
for rank, idx in enumerate(top_indices):
    score = W[query_image_index, idx]
    similar_image_metadata = combined_df.iloc[idx]
    print(f"  {rank + 1}. Image ID: {similar_image_metadata['image_id']}, "
          f"Class ID: {similar_image_metadata['class_id']}, "
          f"Species: {similar_image_metadata['species']}, "
          f"Breed ID: {similar_image_metadata['breed_id']}, "
          f"Score: {score:.4f}")
```

Για κάθε εικόνα στο σύνολο δεδομένων, ο κώδικας εντοπίζει τις **top-k πιο παρόμοιες εικόνες** χρησιμοποιώντας τον πίνακα συσχέτισης **W**. Για την ανάκτηση, εξαιρείται η ίδια η εικόνα και ταξινομούνται οι υπόλοιπες με βάση το βαθμό ομοιότητας. Τα αποτελέσματα εμφανίζουν τα **ID των εικόνων**, την **κατηγορία (class ID)**, το **είδος (species)**, τη **φυλή (breed ID)** και το **σκορ ομοιότητας**. Ο χρήστης μπορεί επίσης να κάνει αναζήτηση εισάγοντας είτε το **index** είτε το **ID** της εικόνας που θέλει να εξετάσει, και το πρόγραμμα επιστρέφει τις πιο σχετικές εικόνες. Επιπλέον, ο αλγόριθμος από μόνος του εμφανίζει ορισμένα ζευγάρια σχετικών εικόνων.

```
# --- step 9: evaluation ---

# split the dataset into train and test sets.
train_indices, test_indices = train_test_split(range(len(combined_df)), test_size=0.2, random_state=42)

# build ground truth mapping:
# for each test image, ground_truth[idx] is the set of indices of images with the same class and breed.
ground_truth = {}
for idx in test_indices:
    row = combined_df.iloc[idx]
    class_id = row['class_id']
    breed_id = row['breed_id']
    # Get indices of all images with the same class_id and breed_id.
    ground_truth[idx] = set(combined_df[(combined_df['class_id'] == class_id) & (combined_df['breed_id'] ==
breed_id)].index.tolist())

# helper function to compute Discounted Cumulative Gain.
```

```
def dcg(relevances):
    return sum(rel / np.log2(rank + 1) for rank, rel in enumerate(relevances, start=1))

k_values = [5]  # top-k value for evaluation
results = {}

for k in k_values:
    precision_scores = []
    ap_scores = []
    ndcg_scores = []

    for idx in test_indices:
        query_image_index = idx
        # get full ranking for the query image (excluding the query image itself).
        ranking = np.argsort(W[query_image_index])[::-1]
        ranking = [i for i in ranking if i != query_image_index]
        top_k_indices = ranking[:k]

        relevant_set = ground_truth[idx]
        num_relevant = len(relevant_set)
        retrieved_relevant = [i for i in top_k_indices if i in relevant_set]
        num_retrieved_relevant = len(retrieved_relevant)

        # precision
        precision = num_retrieved_relevant / k
        # Average Precision (AP) for the top-k list.
        ap = 0.0
        hit_count = 0
        for rank, i in enumerate(top_k_indices, start=1):
            if i in relevant_set:
                hit_count += 1
                ap += hit_count / rank
        if hit_count > 0:
            ap /= hit_count

        # NDCG for the top-k list.
        actual_relevances = [1 if i in relevant_set else 0 for i in top_k_indices]
        ideal_relevances = sorted(actual_relevances, reverse=True)
        actual_dcg = dcg(actual_relevances)
        ideal_dcg = dcg(ideal_relevances) if ideal_relevances else 0
        ndcg = actual_dcg / ideal_dcg if ideal_dcg > 0 else 0
```

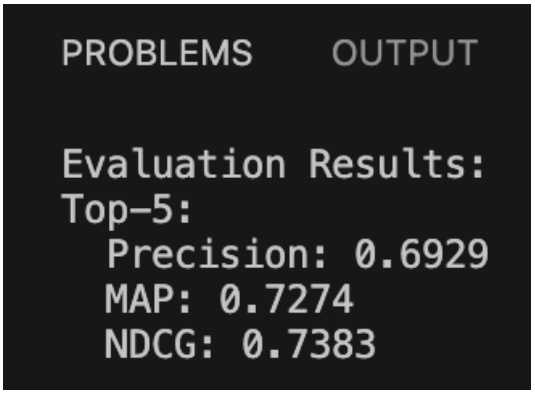
```
        precision_scores.append(precision)
        ap_scores.append(ap)
        ndcg_scores.append(ndcg)

    results[k] = {
        'Precision': np.mean(precision_scores),
        'MAP': np.mean(ap_scores),
        'NDCG': np.mean(ndcg_scores)
    }

# results.
print("Evaluation Results:")
for k_val, metrics in results.items():
    print(f"Top-{k_val}:")
    for metric, value in metrics.items():
        print(f"  {metric}: {value:.4f}")
    print()
```

Επιπλέον έγινε προσπάθεια να μετρηθεί η ακρίβεια χρησιμοποιώντας τρεις βασικούς δείκτες: **Precision@k (ακρίβεια στις top-k εικόνες)**, **Mean Average Precision (MAP)** και **Normalized Discounted Cumulative Gain (NDCG)**. Αρχικά, το σύνολο δεδομένων διαχωρίζεται σε **training** και **test** (80 - 20).Για κάθε εικόνα test, δημιουργείται ένας χάρτης αναφοράς (**ground truth**), ο οποίος περιλαμβάνει όλες τις εικόνες με την ίδια κατηγορία και φυλή. Αυτός ο χάρτης χρησιμοποιείται για να ελέγξει αν οι ανακτηθείσες εικόνες είναι σχετικές.

Η διαδικασία αξιολόγησης ξεκινά με την ταξινόμηση των εικόνων με βάση τον πίνακα συσχέτισης **W**. Για κάθε εικόνα αναφοράς, επιλέγονται οι **top-k πιο παρόμοιες εικόνες**. Ο δείκτης **Precision@k** υπολογίζει το ποσοστό των εικόνων αυτών που είναι σχετικές, δηλαδή ανήκουν στην ίδια κατηγορία και φυλή. Η μέση ακρίβεια (**MAP**) υπολογίζεται ως ο μέσος όρος των ακρίβειών σε διαφορετικές θέσεις κατάταξης σχετικών εικόνων, λαμβάνοντας υπόψη τη θέση στην οποία εμφανίζονται. Αν μια σχετική εικόνα εμφανιστεί νωρίς στην κατάταξη, η συμβολή της στο συνολικό αποτέλεσμα είναι μεγαλύτερη.

Για την αξιολόγηση της συνολικής ποιότητας της κατάταξης, χρησιμοποιείται ο δείκτης **NDCG**, ο οποίος δίνει μεγαλύτερη βαρύτητα στις σχετικές εικόνες που βρίσκονται στις υψηλότερες θέσεις της κατάταξης. Ο **DCG (Discounted Cumulative Gain)** υπολογίζει το κέρδος που προκύπτει από τη θέση κάθε σχετικής εικόνας, ενώ ο **IDCG (Ideal DCG)** υπολογίζει το ιδανικό κέρδος με όλες τις σχετικές εικόνες στις πρώτες θέσεις. Ο **NDCG** είναι ο λόγος των δύο τιμών και μετρά πόσο κοντά είναι η πραγματική κατάταξη στην ιδανική (βλέπε documentation).

Τα αποτελέσματα φαίνονται παρακάτω, **Precision: 0.6929, MAP: 0.7274, NDCG: 0.7383**.



Εικόνα 2. Τιμές μετρικών

Τα παραπάνω αποτελέσματα ήταν πολύ χειρότερα χρησιμοποιώντας την μέθοδο iterative ranking που προτάθηκε στο άρθρο και που αναλύθηκε στην διάλεξη, αφού όλα ήταν περίπου στο ½ της απόδοσης.

Παράγοντες που επηρεάζουν τις μετρικές, φαίνονται παρακάτω

1. **Πολυπλοκότητα και παραλλαγές του dataset:**

   Το **Oxford-IIIT Pet Dataset** περιλαμβάνει εικόνες με μεγάλες παραλλαγές σε φωτισμό, στάση, κλίμακα και περιβάλλον, γεγονός που δυσκολεύει την αναγνώριση των σωστών συσχετισμών. Επίσης, ορισμένα ζώα μπορεί να μοιάζουν μεταξύ διαφορετικών φυλών, προκαλώντας λανθασμένες ανακτήσεις.

2. **Μικρός αριθμός top-k:**

   Για παράδειγμα, αν το **k = 5**, το σύστημα μπορεί να αποδώσει καλύτερα για μεγαλύτερα **k-values**.

3. **Η μέθοδος που χρησιμοποιείται για iterative ranking:**

   Για παράδειγμα, η προηγούμενη μέθοδος που παρουσιάστηκε στο βήμα 6 (η πρώτη), είχε το ½ της απόδοσης αυτής που τελικά χρησιμοποιήθηκε

Τελευταία, και πιο απλή συστημική διαδικασία για την μέτρηση της ακρίβειας αποτελεί η παρακάτω μέθοδος.

```
# evaluate precision with ground truth
def evaluate_precision(W, combined_df, query_index, k=5):
    # get ground truth: images of same breed/class as query
    query_meta = combined_df.iloc[query_index]
    ground_truth = combined_df[
        (combined_df['breed_id'] == query_meta['breed_id']) &
        (combined_df.index != query_index)
    ].index.tolist()
```

```
# get top k retrieved images
sorted_indices = np.argsort(W[query_index])[-k*2:][::-1]
retrieved = [idx for idx in sorted_indices if idx != query_index][:k]

# calculate precision
relevant = sum(1 for idx in retrieved if idx in ground_truth)
precision = relevant / k

return precision
```

Η συνάρτηση **evaluate_precision** υπολογίζει την ακρίβεια **(Precision)** του συστήματος ανάκτησης εικόνων για μία συγκεκριμένη εικόνα. Συγκεκριμένα, εντοπίζει τις σχετικές εικόνες **(ground truth)**, δηλαδή αυτές που ανήκουν στην ίδια φυλή (**breed_id**) με την εικόνα-ερώτημα, εξαιρώντας την ίδια την εικόνα από το σύνολο των σχετικών. Στη συνέχεια, ταξινομεί τις εικόνες με βάση τον πίνακα ομοιότητας **W**, όπου οι πιο όμοιες εικόνες κατατάσσονται πρώτες. Από αυτή την ταξινόμηση, λαμβάνει τις **top-k** πιο όμοιες εικόνες και υπολογίζει πόσες από αυτές είναι πράγματι σχετικές με βάση το ground truth. Τέλος, η ακρίβεια προκύπτει ως το πηλίκο των σχετικών εικόνων που ανακτήθηκαν προς το σύνολο των **k εικόνων** που εξετάστηκαν.

3. ΕΠΙΔΕΙΞΗ ΤΗΣ ΛΥΣΗΣ

Για να αναδείξουμε την λειτουργία του αλγορίθμου θα τρέξει για μία συγκεκριμένη εικόνα **(query image)**, όπως φαίνεται παρακάτω.

```
==================================================
Top 5 similar images for query image Bombay_120:
  Class ID: 8, Species: 1, Breed ID: 4

  1. Image ID: Bombay_132, Class ID: 8, Species: 1, Breed ID: 4, Score: 7.1517
  2. Image ID: Bombay_107, Class ID: 8, Species: 1, Breed ID: 4, Score: 4.2887
  3. Image ID: Bombay_19, Class ID: 8, Species: 1, Breed ID: 4, Score: 1.9111
  4. Image ID: Bombay_46, Class ID: 8, Species: 1, Breed ID: 4, Score: 0.0001
  5. Image ID: Bombay_34, Class ID: 8, Species: 1, Breed ID: 4, Score: 0.0001
```

Εικόνα 3. Top-5 εικόνες βάση του αλγορίθμου

Παρατηρείται πως με βάση την εικόνα *Bombay_120*, ο αλγόριθμος ανακτά σωστά 5 εικόνες που είναι ακριβώς ίδια ράτσα.

```
Top 5 similar images for image yorkshire_terrier_92:
  Class ID: 37, Species: 2, Breed ID: 25

  1. Image ID: yorkshire_terrier_107, Class ID: 37, Species: 2, Breed ID: 25, Score: 3.5278
  2. Image ID: yorkshire_terrier_55, Class ID: 37, Species: 2, Breed ID: 25, Score: 1.0645
  3. Image ID: yorkshire_terrier_104, Class ID: 37, Species: 2, Breed ID: 25, Score: 0.3314
  4. Image ID: yorkshire_terrier_17, Class ID: 37, Species: 2, Breed ID: 25, Score: 0.1002
  5. Image ID: yorkshire_terrier_58, Class ID: 37, Species: 2, Breed ID: 25, Score: 0.0180
```

Εικόνα 4. Top-5 εικόνες βάση του αλγορίθμου

Παρατηρείται πως με βάση την εικόνα *yorkshire_terrier_92*, ο αλγόριθμος ανακτά σωστά 5 εικόνες που είναι ακριβώς ίδια ράτσα.

```
==========================================
Top 5 similar images for image wheaten_terrier_93:
  Class ID: 36, Species: 2, Breed ID: 24

  1. Image ID: wheaten_terrier_6, Class ID: 36, Species: 2, Breed ID: 24, Score: 0.7308
  2. Image ID: wheaten_terrier_8, Class ID: 36, Species: 2, Breed ID: 24, Score: 0.2100
  3. Image ID: wheaten_terrier_7, Class ID: 36, Species: 2, Breed ID: 24, Score: 0.1189
  4. Image ID: wheaten_terrier_133, Class ID: 36, Species: 2, Breed ID: 24, Score: 0.0739
  5. Image ID: wheaten_terrier_58, Class ID: 36, Species: 2, Breed ID: 24, Score: 0.0696
```

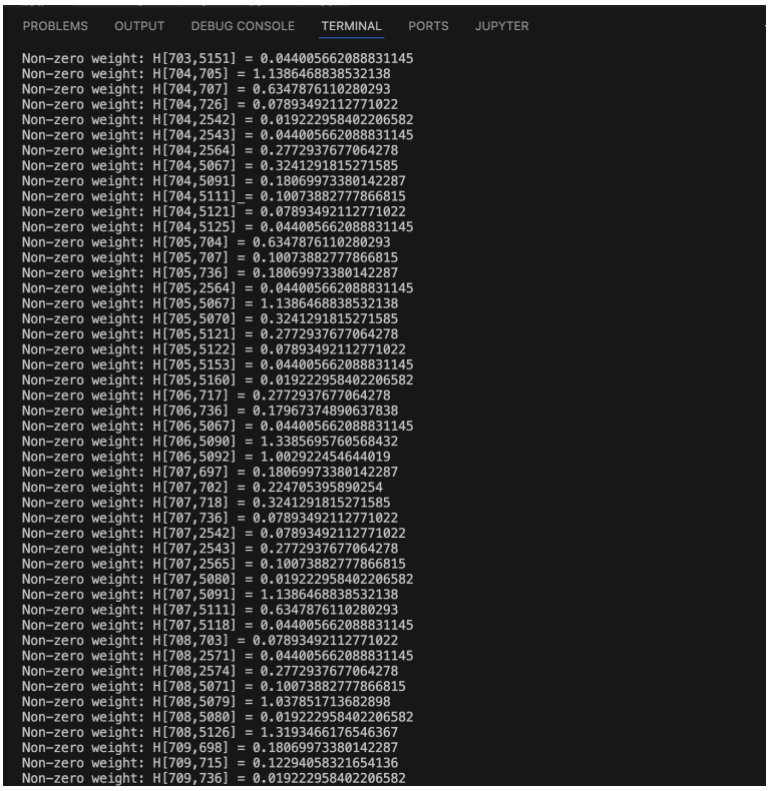
Εικόνα 5. Top-5 εικόνες βάση του αλγορίθμου

Παρατηρείται πως με βάση την εικόνα *wheaten_terrier_93*, ο αλγόριθμος ανακτά σωστά 5 εικόνες που είναι ακριβώς ίδια ράτσα.

Επιπλέον παρακάτω μπορούν να παρατηρηθούν στιγμιότυπα από τον αλγόριθμο.

```
ioannis@4 assignment % /usr/local/bin/python3 "/Users/ioannis/Library/CloudStorage/OneDrive-unipi.gr
/Desktop/ioannis/university/7th semester/image analysis/assignment/scripts/final_working_scripts/man
ifold_ranking.py"
          image_id  class_id  species  breed_id
0  Abyssinian_100         1        1         1
1  Abyssinian_101         1        1         1
2  Abyssinian_102         1        1         1
3  Abyssinian_103         1        1         1
4  Abyssinian_104         1        1         1
(7349, 2048)
Distance Matrix (first 10 rows):
[[0.         0.16169261 0.16801011 ... 0.39620133 0.43145609 0.37732539]
 [0.16169261 0.         0.10309354 ... 0.41215095 0.41056258 0.39751829]
 [0.16801011 0.10309354 0.         ... 0.38621546 0.40075803 0.38434794]
 ...
 [0.14478142 0.15212219 0.11131591 ... 0.34522607 0.38448358 0.35300217]
 [0.14842945 0.10400904 0.05399436 ... 0.409488   0.42733155 0.39143953]
 [0.36043241 0.3513676  0.33280898 ... 0.37965635 0.41091193 0.38098724]]

Similarity Matrix (first 10 rows):
[[1.         0.85070266 0.84534529 ... 0.67287122 0.64956259 0.68569292]
 [0.85070266 1.         0.90204259 ... 0.6622243  0.663277   0.67198565]
 [0.84534529 0.90204259 1.         ... 0.67962408 0.66981212 0.68089447]
 ...
 [0.86521139 0.85888333 0.89465607 ... 0.70806026 0.68080213 0.70257567]
 [0.86206082 0.90121715 0.94743745 ... 0.66399013 0.65224727 0.67608293]
 [0.69737471 0.70372502 0.71690712 ... 0.68409646 0.66304533 0.68318661]]

Initial Ranks (first 10 rows):
[[   0 2310   38 ... 6720 5948 1506]
 [   1 2122 1883 ... 7153 5948 1506]
 [   2 3775    8 ... 7153 5948 1506]
 ...
 [   7 1856    2 ... 2479 5948 1506]
 [   8    2 3775 ... 4903 5948 1506]
 [   9 3698 1880 ... 5948 2467 1506]]

Initial Sorted Similarity Values (first 10 rows):
[[1.         0.87969196 0.87772771 ... 0.59277282 0.57252044 0.56887136]
 [1.         0.91207912 0.90358511 ... 0.57228923 0.54729895 0.53731837]
 [1.         0.96822536 0.94743745 ... 0.57413873 0.56211639 0.54780676]
 ...
 [1.         0.89996027 0.89465607 ... 0.5903416  0.58927305 0.58187164]
 [1.         0.94743745 0.94610736 ... 0.57609219 0.55583589 0.54682286]
 [1.         0.8314763  0.82911817 ... 0.59294125 0.5917898  0.56270412]]
```

Εικόνα 6. Αρχικοί πίνακες αποστάσεων και ομοιότητας



Εικόνα 7. Υπολογισμός τιμών του πίνακα συνάφειας H

4. ΕΠΕΚΤΑΣΕΙΣ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΠΡΟΟΠΤΙΚΕΣ

Μετά το τελευταίο βήμα του αλγορίθμου ακολουθεί ένα επιπλέον βήμα για να αποθηκεύσει τη δομή του υπεργράφου και τους σχετικούς πίνακες, ώστε να μην χρειάζεται κάθε φορά να επαναυπολογίζονται αφού η διαδικάσια διαρκεί κάποια ώρα.

```
# --- step 7: save for later use ---


# combine features and metadata into a DataFrame
combined_data = []
for i in range(len(features_array)):
    row = image_info_df.iloc[i]
    combined_data.append({
        'image_id': row['image_id'],
        'class_id': row['class_id'],
        'species': row['species'],
        'breed_id': row['breed_id'],
        'features': features_array[i],
        'initial_ranks_subset': initial_ranks[i],
        'initial_sorted_values_subset': initial_ranks_values[i],
        'normalized_ranks': normalized_ranks[i]
    })
combined_df = pd.DataFrame(combined_data)
```

```
print("Combined DataFrame head:")
print(combined_df.head())

# get the absolute path of the current script
current_script_path = os.path.abspath(__file__)
artifacts_path = os.path.abspath(os.path.join(os.path.dirname(current_script_path), '..', '..', 'artifacts'))
os.makedirs(artifacts_path, exist_ok=True)

# save hypergraph data
hypergraph_file = os.path.join(artifacts_path, 'hypergraph_data.npz')
np.savez(hypergraph_file, H=H, Sh=Sh, Sv=Sv, W=W) # compressed numpy file
print("Hypergraph data saved to 'hypergraph_data.npz'.")

# save metadata (combined DataFrame)
combined_data_file = os.path.join(artifacts_path, 'combined_data.csv')
combined_df.to_csv(combined_data_file, index=False) # csv file
print("Metadata saved to 'combined_data.csv'.")
```

Έτσι προέκυψε η ιδέα να φτιαχτεί ένα ακόμα script το οποίο θα χρησιμοποιεί τις αποθηκευμένες δομές ώστε να προβάλλει σε ένα GUI τις εικόνες.

```
import numpy as np
import pandas as pd
import os
from PIL import Image
import matplotlib.pyplot as plt
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout,
                             QPushButton, QLabel, QLineEdit, QFrame)
from PyQt5.QtCore import Qt


class ImageRetrievalGUI(QMainWindow):
    def __init__(self, W, combined_df):
        super().__init__()
        self.W = W
        self.combined_df = combined_df
        self.init_ui()

    def init_ui(self):
        self.setWindowTitle('Image Retrieval')
        self.setGeometry(400, 200, 400, 150)
```

```
    self.setStyleSheet('background-color: black;')  # color

    central_widget = QWidget()
    self.setCentralWidget(central_widget)
    layout = QVBoxLayout(central_widget)
    layout.setContentsMargins(40, 20, 40, 20)

    # create a container for centered content
    container = QWidget()
    container_layout = QVBoxLayout(container)

    # input section with styling
    self.input_label = QLabel('Enter image ID or index:')
    self.input_label.setAlignment(Qt.AlignCenter)
    self.input_label.setStyleSheet('font-size: 14px; margin-bottom: 5px; color: white;')

    self.input_field = QLineEdit()
    self.input_field.setFixedHeight(30)
    self.input_field.setStyleSheet('padding: 5px; border: 1px solid #ccc; border-radius: 4px; color: white;')

    self.search_button = QPushButton('Search')
    self.search_button.setFixedHeight(35)
    self.search_button.setStyleSheet("""
        QPushButton {
            background-color: white;
            color: black;
            border: none;
            border-radius: 4px;
            padding: 5px 20px;
            font-size: 14px;
        }
        QPushButton:hover {
            background-color: grey;
        }
    """)
    self.search_button.clicked.connect(self.search_images)

    # add widgets to container
    container_layout.addWidget(self.input_label)
    container_layout.addWidget(self.input_field)
    container_layout.addWidget(self.search_button)
```

```
        container_layout.setSpacing(10)

        # add container to main layout
        layout.addWidget(container)

    def search_images(self):
        query = self.input_field.text()
        try:
            query = int(query)
        except ValueError:
            query = query.strip()

        query_image(self.W, self.combined_df, query, top_k=5, image_folder='data/images')


def load_saved_model(hypergraph_file='hypergraph_data.npz', metadata_file='combined_data.csv'):
    if not os.path.exists(hypergraph_file):
        print(f"Saved hypergraph model '{hypergraph_file}' not found. Please run the model-building script first.")
        exit(1)
    data = np.load(hypergraph_file)
    W = data['W']

    if not os.path.exists(metadata_file):
        print(f"Metadata file '{metadata_file}' not found. Please ensure you have saved the combined metadata.")
        exit(1)
    combined_df = pd.read_csv(metadata_file)

    return W, combined_df

def query_image(W, combined_df, query, top_k=5, image_folder='images'):
    # determine query index: if query is int, use it directly; otherwise, search for the image_id.
    if isinstance(query, int):
        query_index = query
    else:
        matches = combined_df[combined_df['image_id'] == query].index
        if len(matches) == 0:
            print(f"No image with id '{query}' found.")
            return
        query_index = matches[0]

    # retrieve top_k similar images using the saved affinity matrix W.
```

```
# exclude the query image itself.
sorted_indices = np.argsort(W[query_index])[-top_k*2:][::-1]
top_indices = [idx for idx in sorted_indices if idx != query_index][:top_k]

query_meta = combined_df.iloc[query_index]
print(f"Top {top_k} similar images for image {query_meta['image_id']}:")
for rank, idx in enumerate(top_indices):
    score = W[query_index, idx]
    meta = combined_df.iloc[idx]
    print(f"  {rank + 1}. Image ID: {meta['image_id']}, "
          f"Class ID: {meta['class_id']}, "
          f"Species: {meta['species']}, "
          f"Breed ID: {meta['breed_id']}, "
          f"Score: {score:.4f}")

precision = evaluate_precision(W, combined_df, query_index, k=top_k)
print(f"\nPrecision@{top_k}: {precision:.4f}")

# plot the query image and its top similar images.
# we assume each image is named as image_id + ".jpg"
plt.style.use('dark_background')
plt.rcParams['figure.facecolor'] = 'black'
plt.rcParams['savefig.facecolor'] = 'black'
plt.rcParams['axes.edgecolor'] = 'black'
plt.rcParams['figure.edgecolor'] = 'black'
plt.rcParams['axes.titlecolor'] = 'white'
plt.rcParams['axes.labelcolor'] = 'white'
plt.rcParams['lines.color'] = 'white'
num_plots = top_k + 1
fig, axes = plt.subplots(2, num_plots, figsize=(4 * num_plots, 8))
fig.patch.set_facecolor('black')

# query image
query_meta = combined_df.iloc[query_index]
query_image_path = os.path.join(image_folder, f"{query_meta['image_id']}.jpg")
try:
    query_img = Image.open(query_image_path)
    axes[0,0].imshow(query_img)
except Exception as e:
    print(f"Could not load query image: {e}")
axes[0,0].axis('off')
```

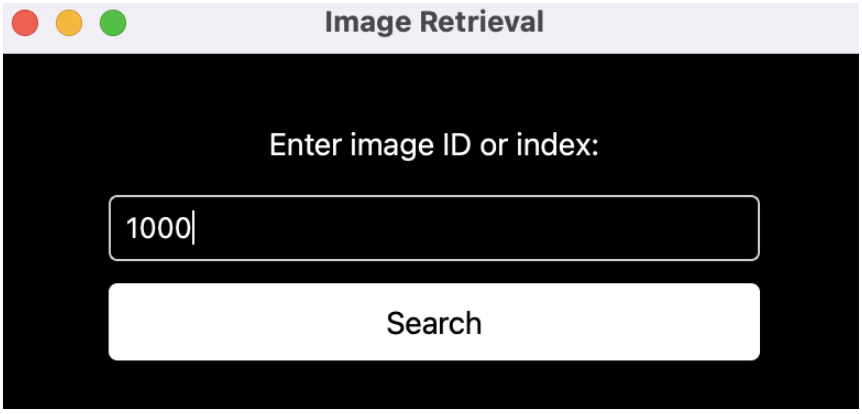
```
    axes[0,0].set_title("Query", color='white')
    axes[0,0].set_facecolor('black')

    # query metadata
    meta_text = f"Image ID: {query_meta['image_id']}\nClass ID: {query_meta['class_id']}\n" \
                f"Species: {query_meta['species']}\nBreed ID: {query_meta['breed_id']}"
    axes[1,0].text(0.5, 0.5, meta_text, ha='center', va='center', color='white')
    axes[1,0].axis('off')
    axes[1,0].set_facecolor('black')

    # similar images
    for i, idx in enumerate(top_indices, start=1):
        meta = combined_df.iloc[idx]
        score = W[query_index, idx]

        # image
        image_path = os.path.join(image_folder, f"{meta['image_id']}.jpg")
        try:
            img = Image.open(image_path)
            axes[0,i].imshow(img)
        except Exception as e:
            print(f"Could not load image: {e}")
        axes[0,i].axis('off')
        axes[0,i].set_title(f"Rank {i}\nScore: {score:.4f}", color='white')
        axes[0,i].set_facecolor('black')

        # metadata
        meta_text = f"Image ID: {meta['image_id']}\nClass ID: {meta['class_id']}\n" \
                    f"Species: {meta['species']}\nBreed ID: {meta['breed_id']}"
        axes[1,i].text(0.5, 0.5, meta_text, ha='center', va='center', color='white')
        axes[1,i].axis('off')
        axes[1,i].set_facecolor('black')

    plt.suptitle(f"Precision@{top_k}: {precision:.4f}", y=0.98, color='white')
    plt.tight_layout()
    plt.show()

# evaluate precision with ground truth
def evaluate_precision(W, combined_df, query_index, k=5):
    # get ground truth: images of same breed/class as query
    query_meta = combined_df.iloc[query_index]
```

```
    ground_truth = combined_df[
        (combined_df['breed_id'] == query_meta['breed_id']) &
        (combined_df.index != query_index)
    ].index.tolist()

    # get top k retrieved images
    sorted_indices = np.argsort(W[query_index])[-k*2:][::-1]
    retrieved = [idx for idx in sorted_indices if idx != query_index][:k]

    # calculate precision
    relevant = sum(1 for idx in retrieved if idx in ground_truth)
    precision = relevant / k

    return precision


if __name__ == '__main__':
    app = QApplication(sys.argv)
    W, combined_df = load_saved_model()
    window = ImageRetrievalGUI(W, combined_df)
    window.show()
    sys.exit(app.exec_())
```

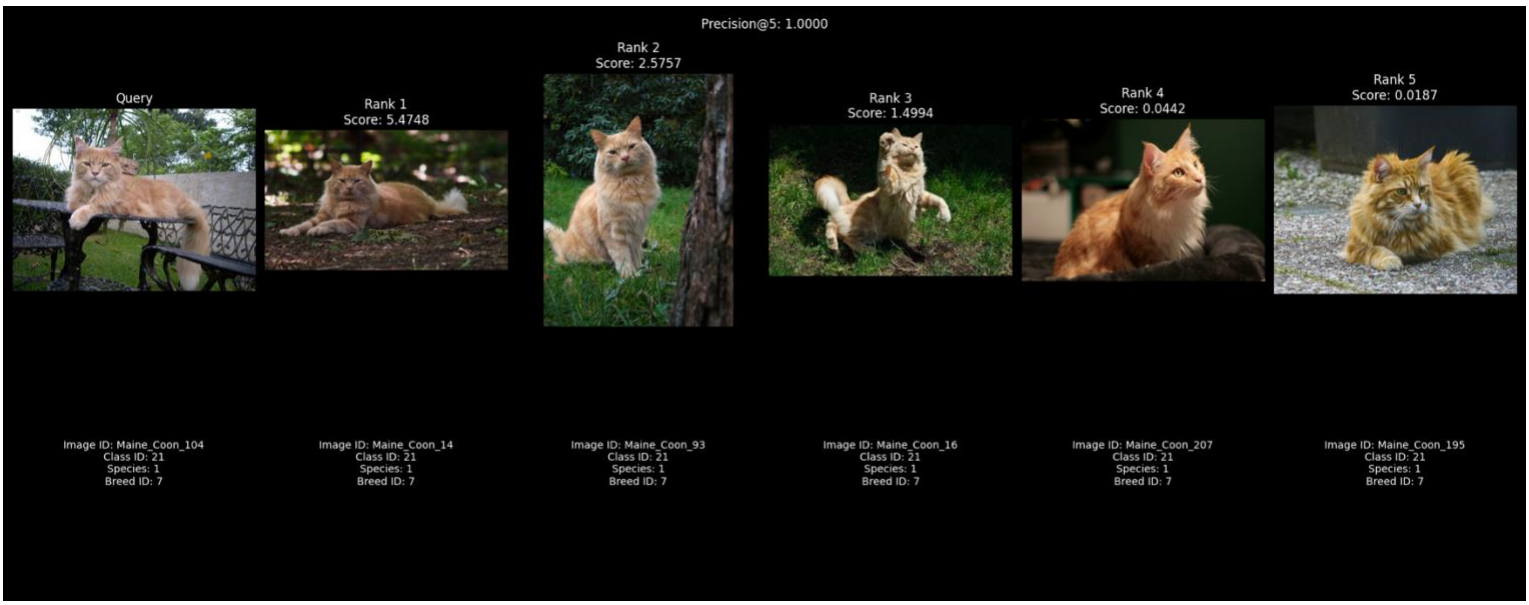
Ο χρήστης εισάγει το **ID ή τον δείκτη της εικόνας** που θέλει να ανακτήσει και το πρόγραμμα εμφανίζει τις top-k παρόμοιες εικόνες. Η διεπαφή κατασκευάζεται με τη χρήση της βιβλιοθήκης **PyQt5** και οι εικόνες προβάλλονται σε έναν καμβά **matplotlib**.

Η βασική διαδικασία ξεκινά με τη φόρτωση του αποθηκευμένου μοντέλου υπεργραφήματος **(hypergraph)** και των μεταδεδομένων από τα αρχεία **hypergraph_data.npz** και **combined_data.csv**. Στη συνέχεια, με τη συνάρτηση **query_image()**, βρίσκονται οι πιο παρόμοιες εικόνες, με βάση τον πίνακα ομοιότητας **W**. Επίσης, υπολογίζεται το **Precision@k** με τη συνάρτηση **evaluate_precision()**, ελέγχοντας πόσες από τις ανακτηθείσες εικόνες είναι σχετικές (ίδιας φυλής).

Παρακάτω φαίνεται η διαδικασία, ενώ στο αρχείο **gui.mp4**, υπάρχει βίντεο με τη διαδικασία.

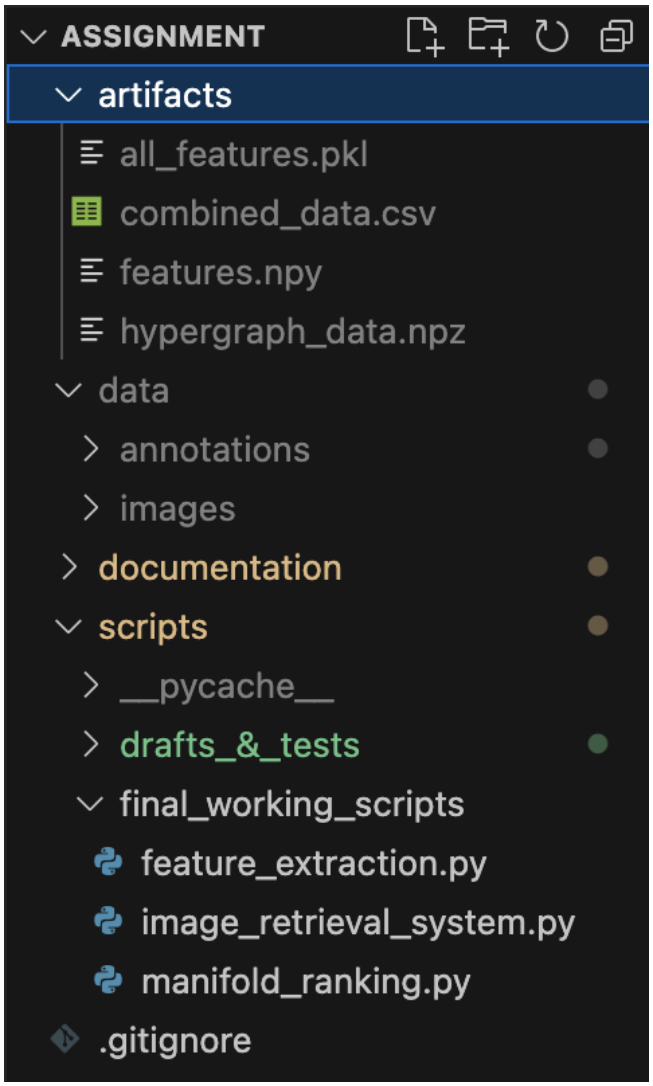


Εικόνα 8. GUI



Εικόνα 9. Figure με τις πιο σχετικές εικόνες και την ακρίβεια (precision 100%)

Παρακάτω φαίνεται η δομή του φακέλου, σε περίπτωση που κάποιος θελήσει να αναπαράγει τον αλγόριθμο.



Εικόνα 10. Figure με τις πιο σχετικές εικόνες και την ακρίβεια (precision 100%)

ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ ΑΝΑΦΟΡΕΣ

[1] https://www.youtube.com/watch?v=WswUVN2fb_4 (Theory)

[2] https://www.youtube.com/watch?v=xfh8iqY-das (Theory)

[3] https://github.com/lopes-leonardo/sgcc (Theory)

[4] https://pytorch.org/vision/0.18/models/generated/torchvision.models.resnet50.html (ResNet50)

[5] https://www.robots.ox.ac.uk/~vgg/data/pets/ (Dataset)

[6] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.dcg_score.html (DCG Score)

[7] https://github.com/dkaterenchuk/ranking_measures/blob/master/measures.py (Implementation of DCG Score)

[8] Διαλέξεις, Σημειώσεις και Διαφάνειες μαθήματος

[9] Σύγγραμμα μαθήματος