UNIVERSITATEA NAȚIONALĂ DE ȘTIINȚĂ ȘI TEHNOLOGIE POLITEHNICA BUCUREȘTI
FACULTATEA DE ȘTIINȚE APLICATE
Programul de studii:
Teoria Codării şi Stocării Informației

Algoritmi de Recunoaștere Facială

STUDENT

Becheru Alexandru

DISCIPLINĂ

Proiectarea traficului informațional în rețelele de calculatoare

București
2025

Cuprins

Introducere

Recunoașterea facială este o tehnologie inovatoare care transformă modul în care interacționăm cu lumea digitală. Această tehnologie este utilizată într-o multitudine de industrii, inclusiv securitatea, sănătatea, marketingul și divertismentul, și utilizează algoritmi puternici pentru a identifica și autentifica identitatea unei persoane prin evaluarea trăsăturilor feței.

Acest proiect intenționează să exploreze principalele abordări și tehnici din domeniul recunoașterii faciale, punând accentul atât pe aplicațiile practice, cât și pe obstacole. Algoritmii de recunoaștere facială sunt o combinație fascinantă de inteligență artificială, procesare a imaginilor și biometrie care permite sistemelor informatice să evalueze și să identifice trăsăturile distincte ale unei fețe. De la scenarii science fiction la aplicații zilnice, această tehnologie a devenit omniprezentă.

Recunoașterea facială presupune preluarea unei imagini, prelucrarea acesteia și compararea datelor obținute cu o bază de date existentă. Algoritmii examinează caracteristici specifice precum distanța dintre ochi, forma nasului, conturul maxilarului și curbele faciale. Rețelele neuronale profunde sunt utilizate pentru a învăța și a recunoaște modelele unice ale fiecărui individ. Etapele acestei proceduri includ identificarea unei fețe într-o imagine sau într-un videoclip, extragerea și analiza atributelor faciale și compararea rezultatelor cu datele înregistrate. Sunt utilizați algoritmi tradiționali precum Viola-Jones, precum și sisteme mai noi bazate pe rețele neuronale precum FaceNet sau DeepFace, ambele oferind o precizie excepțională.

Una dintre cele mai răspândite aplicații pentru recunoașterea facială este securitatea. Această tehnologie este utilizată pentru deblocarea dispozitivelor mobile, restricționarea accesului în clădiri și monitorizarea zonelor publice. În industria financiară, aceasta ajută la autentificarea tranzacțiilor și la prevenirea fraudelor, în timp ce în domeniul medical, este utilizată pentru a îmbunătăți îngrijirea pacienților și pentru a identifica persoanele dispărute. De asemenea, companiile pot utiliza marketingul pentru a studia reacțiile emoționale ale clienților lor la produse și servicii.

Cu toate avantajele sale, recunoașterea facială generează preocupări etice și juridice. Una dintre principalele preocupări este siguranța datelor cu caracter personal și potențialul de supraveghere intruzivă, așa cum au demonstrat incidentele în care au fost implicate firme precum Hik-Vision și Dahua. Utilizarea abuzivă a acestor tehnologii poate pune în pericol libertatea individuală, iar anumite sisteme sunt mai puțin precise pentru anumite grupuri etnice, ceea ce duce la discriminare involuntară.

Astfel, deși recunoașterea facială are numeroase avantaje, este esențial să se ia în considerare preocupările legate de viața privată, etică și corectitudine pentru a asigura o utilizare etică și echilibrată a acestei tehnologii.

I. ELEMENTE TEORETICE

Biometria este un termen derivat din cuvintele grecești „bios” (viață) și „metrikos” (măsură). Se referă la un set de metode și tehnici automatizate utilizate pentru a identifica o persoană folosind diverse caracteristici biometrice, cum ar fi geometria palmei, amprenta digitală, irisul, retina, geometria facială sau caracteristici comportamentale, cum ar fi timbrul vocal, configurația ADN-ului, structura scrisului de mână și dinamica tastaturii. Este larg recunoscut faptul că unele dintre aceste trăsături biometrice, cum ar fi amprentele digitale sau irisul, pot identifica în mod unic o persoană. Dată fiind unicitatea lor, datele biometrice pot fi valorificate pentru a construi și implementa metode, echipamente și sisteme care oferă performanțe de identificare mult mai bune decât cele existente.

.

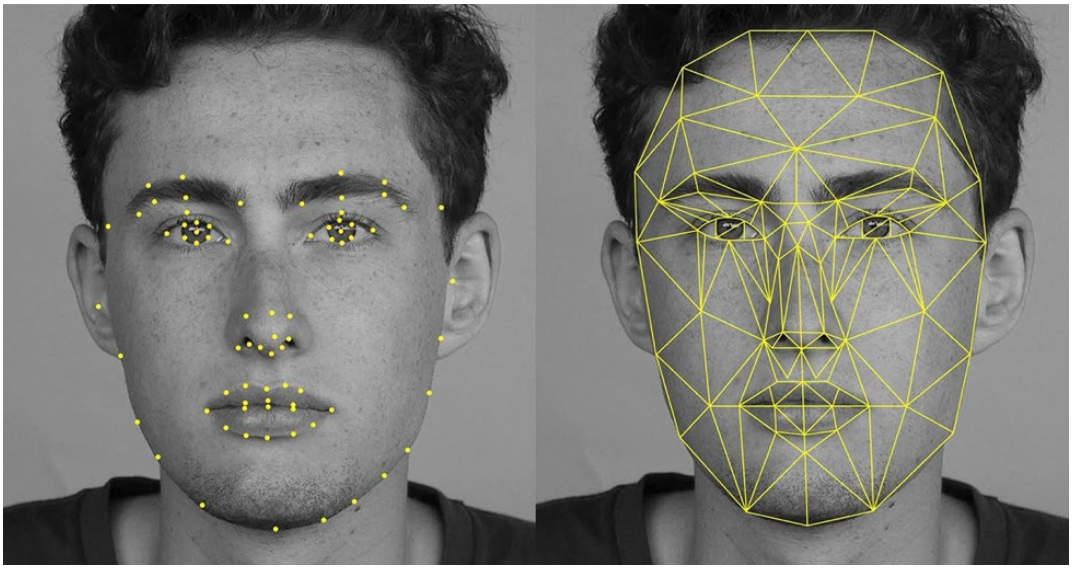


Fig. 1 – Algoritm de recunoaștere facială

Instalarea eficientă a unui sistem de recunoaștere depinde de selectarea celui mai adecvat sistem de recunoaștere pentru a oferi o soluție unică adaptată la nivelul de securitate dorit. Tehnologia biometrică are capacitatea de a lega persoana de acțiunea sau tranzacția efectuată, limitând utilizarea neautorizată și reducând potențialele fraude în sistem.

Un sistem de verificare este mult mai ușor de implementat decât un sistem de recunoaștere a identității. Într-un sistem de verificare, identitatea declarată a unei persoane este certificată prin compararea trăsăturilor biologice precise furnizate de persoană la un moment dat cu trăsăturile stocate anterior în sistem și asociate cu identitatea declarată a persoanei.

Spre deosebire de sistemul de verificare, sistemul de recunoaștere utilizează o abordare mai complexă în care caracteristicile biometrice furnizate de o persoană sunt comparate cu măsurile tuturor caracteristicilor biometrice similare păstrate într-o bază de date sau într-o rețea neuronală.

Recunoașterea irisului este o abordare avansată și automată de identificare biometrică care utilizează algoritmi matematici pentru a recunoaște și analiza modelele din fotografiile sau înregistrările video ale irisului unei persoane. Aceste modele complexe ale irisului sunt unice pentru fiecare individ, rămân stabile în timp și pot fi văzute de la distanță. Această metodă utilizează algoritmi specializați pentru a extrage și a compara caracteristicile specifice ale irisului, permițând identificarea precisă și sigură a persoanelor implicate.

II. PREZENTAREA TEORETICĂ A FRAMEWORK-URILOR DE RECUNOAȘTERE FACIALĂ

Recunoașterea facială este una dintre cele mai avansate aplicații ale inteligenței artificiale (AI) și ale învățării automată (deep learning), cu utilizări care variază de la securitate și verificare biometrică la marketing și interacțiuni digitale personalizate.

Acest capitol intenționează să evalueze în detaliu trei cadre proeminente utilizate pentru recunoașterea facială:

- **FaceNet** – este un model dezvoltat de Google care convertește fețele în vectori numerici pentru o comparație eficientă.
- **DeepFace** – o soluție dezvoltată de Facebook care îmbunătățește acuratețea folosind o arhitectură neuronală profundă și transformări 3D.
- **MediaPipe** – un framework optimizat pentru eficiență și viteză, dezvoltat de Google pentru utilizare în aplicații mobile și web.

În acest capitol vom explora **principiile de funcționare, arhitectura fiecărui model** și vom realiza o **comparație detaliată** între acestea.

II.1 FACENET – PRINCIPII DE FUNCȚIONARE ȘI ARHITECTURA MODELULUI

Principii de funcționare

FaceNet este un model de recunoaștere facială dezvoltat de Google care abordează provocarea identificării persoanelor utilizând o metodă inovatoare numită învățare metrică.

FaceNet, spre deosebire de abordările anterioare care clasifică fețele pe baza unui set predefinit de etichete, convertește fețele în vectori numerici într-un spațiu multidimensional cunoscut sub numele de spațiu de încorporare.

Această metodă compară rapid și precis fețele prin măsurarea distanței euclidiene dintre vectorii rezultați. Fețele similare sunt apropiate, în timp ce fețele diferite sunt îndepărtate. FaceNet este deosebit de eficient datorită acestei metode, care elimină necesitatea de a reantrena modelul pentru fiecare nouă identitate.

Arhitectura modelului

FaceNet utilizează o arhitectură bazată pe Inception (GoogLeNet) care este adaptată pentru extragerea caracteristicilor faciale. Modelul este antrenat prin pierderea tripletului, o funcție de pierdere care optimizează distanța dintre fețe.

Arhitectura FaceNet constă din următorii pași de bază:

1. **Detectarea feței** – localizarea feței într-o imagine folosind un model CNN avansat.
2. **Normalizarea feței** – alinierea feței pentru eliminarea variațiilor de poziție și iluminare.
3. **Extragerea caracteristicilor** – transformarea feței într-un vector numeric de **128 de dimensiuni**.
4. **Compararea vectorilor** – măsurarea similarității faciale utilizând distanța euclidiană.

Aplicații principale:

- **Autentificare biometrică** – utilizată în sistemele de securitate.
- **Identificarea și urmărirea suspecților** – aplicată în supravegherea video.
- **Recunoașterea fețelor în rețele sociale** – Facebook și alte platforme folosesc această tehnologie.

Limitări

- Necesită resurse computaționale mari pentru antrenare și inferență.

- Sensibil la variațiile de iluminare și unghiuri extreme ale feței.

II.2 DEEPFACE – PRINCIPII DE FUNCȚIONARE ȘI ARHITECTURA MODELULUI

Principii de funcționare

DeepFace este un model dezvoltat de Facebook care aspiră la o precizie de recunoaștere facială la nivel uman.

Acest model utilizează o rețea neuronală convoluțională profundă (CNN) și include o caracteristică nouă: normalizarea facială 3D.

Sistemul DeepFace este antrenat prin învățarea metrică profundă, ceea ce presupune că modelul învață să grupeze fețele similare mai aproape într-un spațiu de caracteristici, similar cu abordarea FaceNet.

Arhitectura modelului

DeepFace utilizează o arhitectură neuronală **cu 9 straturi** și este optimizat pentru extragerea caracteristicilor faciale relevante.

Structura modelului include:

1. **Normalizarea feței** – utilizarea unei **transformări 3D** pentru eliminarea variațiilor de poziție.
2. **Extragerea caracteristicilor** – folosirea CNN-urilor pentru generarea vectorilor caracteristici ai feței.
3. **Comparația fețelor** – utilizarea metricii de similaritate pentru recunoaștere.

Aplicații principale

- **Identificarea fețelor în rețele sociale** – folosit de Facebook pentru recunoașterea automată.
- **Securitate și autentificare** – utilizat în sisteme de monitorizare video.
- **Personalizarea conținutului** – recomandări bazate pe recunoașterea facială.

Limitări

- Necesită **un volum mare de date** pentru antrenare.
- Consumă **resurse computaționale** semnificative.

II.3 MEDIAPIPE – PRINCIPII DE FUNCȚIONARE ȘI ARHITECTURA MODELULUI

Principii de funcționare

Google a dezvoltat cadrul de recunoaștere facială MediaPipe, care este conceput pentru identificarea facială în timp real pe dispozitive mobile și aplicații web. Modelul este optimizat pentru eficiență și viteză, sacrificându-se o anumită acuratețe pentru performanțe ridicate.

Arhitectura modelului

MediaPipe folosește un **CNN lightweight**, optimizat pentru funcționare pe CPU și dispozitive mobile. Modelul include:

1. **Detectarea feței** – detectarea și localizarea feței cu un model CNN rapid.
2. **Extragerea caracteristicilor faciale** – extragerea unui set de bază de puncte de caracteristici faciale.
3. **Procesarea în timp real** – optimizări pentru rularea eficientă pe dispozitive cu resurse limitate.

Aplicații principale

- Recunoaștere facială pe dispozitive mobile.

- Realitate augmentată și filtre faciale.
- Aplicații web interactive.

Limitări

- Precizie mai redusă comparativ cu FaceNet și DeepFace.
- Sensibil la calitatea imaginii și la iluminare.

Comparație teoretică între cele trei framework-uri

Caracteristică	FaceNet	DeepFace	MediaPipe
Precizie	Foarte mare	Foarte mare	Medie
Viteză	Medie	Medie	Foarte mare
Scalabilitate	Necesită GPU	Necesită GPU	Funcționează pe CPU
Aplicații	Securitate, biometrie	Rețele sociale, AI	Mobile, realitate augmentată

Astfel, pe baza aspectelor prezentate anterior, putem trage următoarele concluzii:

- **FaceNet și DeepFace** sunt potrivite pentru securitate și biometrie, dar necesită **resurse mari**.
- **MediaPipe** este optimizat pentru **viteză și eficiență**, fiind ideal pentru **aplicații mobile și AR**.

III SETUL DE DATE UTILIZAT

În domeniul recunoașterii faciale, sunt necesare seturi de date mari și variate pentru testarea și formarea modelului. Setul de date Celebrity Face Image Dataset, o bază de date cu imagini ale unor celebrități din întreaga lume, care este accesibilă pe platforma Kaggle, a fost utilizat în această lucrare. Această bază de date este adecvată pentru aplicații precum recunoașterea automată a persoanelor, verificarea facială și categorizarea identității.

3.1. O explicație a bazei de date publice care a fost utilizată

O colecție de imagini ale unor persoane cunoscute aranjate în funcție de identificare se numește Celebrity Face Image Dataset. Cercetarea în domeniul viziunii computerizate, și anume recunoașterea facială, este ținta acestui set de date. Rețelele neuronale convoluționale (CNN), modelele bazate pe învățarea profundă și alte metode de inteligență artificială pentru recunoașterea facială pot fi toate antrenate folosind această bază de date.

Caracteristicile principale ale setului de date includ imagini ale unor persoane faimoase dintr-o varietate de industrii, inclusiv actori, muzicieni, sportivi și altele. Deoarece imaginile provin din surse diferite, există variații în ceea ce privește fundalurile, iluminarea, unghiurile și expresiile feței.

Colecția este aranjată în funcție de numele celebrităților, fiecare dintre acestea având un director cu fotografii unice. Calitatea ridicată a fotografiilor face posibilă utilizarea unei varietăți de metode de prelucrare și recunoaștere.

Această bază de date, care este ușor disponibilă și bine structurată, este adesea utilizată pentru crearea de modele de recunoaștere facială.

3.2. Structura setului de date (numărul de fotografii, distribuția imaginilor)

Setul de date cu imagini ale fețelor celebrităților are următoarea structură organizatorică:

- Cantitatea totală de imagini: Mii de imagini ale diferitelor celebrități sunt incluse în setul de date, în timp ce cantitatea reală de imagini nu este specificată.

- Distribuția imaginilor: Fiecare celebritate are un director în care sunt păstrate mai multe imagini ale sale. Fiecare celebritate poate avea de la câteva zeci la câteva sute de fotografii.
- Dimensiunile imaginilor: Deoarece dimensiunile fotografiilor variază, redimensionarea imaginilor la o dimensiune standard necesită o etapă de preprocesare.

Diversitatea acestei baze de date este una dintre caracteristicile sale esențiale. Deoarece fotografiile au fost colectate din diverse surse, o serie de variabile pot afecta recunoașterea facială, inclusiv:

- Imaginile luate din față, din profil sau din alte perspective sunt cunoscute ca unghiuri de captare.
- Expresiile feței: Serios, șocat, zâmbitor etc.
- Fundalul și iluminarea: Fonduri simple sau complicate, iluminare slabă sau puternică.
- Accesoriile includ pălării, ochelari, diverse produse cosmetice etc.

Setul de date este perfect pentru antrenarea algoritmilor de recunoaștere facială fiabili care pot face față situațiilor din lumea reală datorită imprevizibilității sale.

3.3. Metodologia de preprocesare a imaginilor

Înainte de antrenarea unui model de recunoaștere facială, preprocesarea imaginilor este un pas crucial. Acesta garantează că datele sunt uniforme, curate și optimizate pentru algoritmul de învățare automată.

Am utilizat următoarele metode de preprocesare în această investigație:

1. Redimensionarea imaginilor

Fotografiile din setul de date trebuie să fie reduse la o dimensiune fixă pentru ca algoritmii de recunoaștere facială să poată lucra cu ele, deoarece dimensiunile lor variază. Dimensiunea standard pentru mai multe arhitecturi de rețele neuronale convoluționale (CNN), inclusiv VGG16, ResNet și EfficientNet, este de 224x224 pixeli.

2. Normalizarea pixelilor

Valorile pixelilor sunt normalizate la un interval specificat, adesea între [0, 1] sau [-1, 1], pentru a crește stabilitatea formării modelului. Acest lucru se realizează fie prin aplicarea unei tehnici de normalizare bazate pe media și abaterea standard a setului de date, fie prin simpla împărțire a valorilor RGB la 255.

3. Alinierea și detectarea fețelor

Chipurile pot apărea în numeroase unghiuri și în locuri variate într-o mulțime de imagini. Ne adaptăm la acest lucru prin identificarea și extragerea zonei feței din fiecare imagine utilizând metode de identificare a feței precum cascadele Haar sau MTCNN (Multi-task Cascaded Convolutional Networks).

După detectare, pentru alinierea fețelor se utilizează marcaje faciale, cum ar fi poziția gurii, a nasului și a ochilor. Prin standardizarea pozițiilor fețelor, această tehnică îmbunătățește performanța algoritmului de recunoaștere.

4. Augmentarea datelor

Utilizăm metode de augmentare a imaginilor, precum acestea, pentru a extinde diversitatea setului de date și pentru a evita supraînvățarea modelului:

- Modificarea ușoară a poziției feței prin rotire și translație.
- Oglindirea orizontală, sau răsturnarea, este utilizată pentru a imita variațiile naturale.
- Modificați contrastul și luminozitatea pentru a reproduce diverse scenarii de iluminare.
- Zgomot și încețoșare: pentru a consolida rezistența modelului la zgomot.

5. Conversia scării de gri (opțional)

În unele aplicații, fotografiile alb-negru pot fi utilizate pentru a face recunoaștere facială. Eliminarea informațiilor de culoare permite modelului să se concentreze asupra elementelor structurale ale feței.

Concluzie

Preprocesarea setului de date cu imagini ale fețelor celebre este esențială pentru obținerea unor rezultate fiabile în materie de recunoaștere facială. Utilizăm abordări de scalare, normalizare, detectare a fețelor, aliniere și augmentare pentru a construi un set de date omogen și robust, ideal pentru antrenarea modelelor sofisticate de recunoaștere facială.

Această abordare garantează că algoritmii de recunoaștere pot face față impredictibilității datelor și obstacolelor din lumea reală, sporind astfel precizia și fiabilitatea sistemului de recunoaștere facială.

IV. IMPLEMENTAREA PRACTICĂ A EXPERIMENTULUI

Acest capitol descrie implementarea practică a experimentului de recunoaștere facială utilizând trei modele populare: FaceNet, DeepFace și MediaPipe. Acești algoritmi sunt testați în ceea ce privește acuratețea, viteza de procesare și performanța într-o varietate de scenarii.

4.1. Configurarea mediului de testare

Am utilizat Python pentru a proiecta și testa algoritmii de recunoaștere facială. Mediul de testare este configurat pentru a rula eficient rețele neuronale convoluționale și pentru a gestiona colecții uriașe de imagini.

Parametrii hardware și software de testare:

- Procesor: Intel Core i7-10850H 4.4 GHz.
- GPU: nVIDIA GeForce GTX 1650 Ti.
- RAM: 16 GB DDR4.
- Sisteme de operare: Windows 11 Professional.

4.2 IMPLEMENTAREA SOLUȚIEI FOLOSIND FACENET

Acest cod (regăsit în Anexa 1) implementează un sistem complet de recunoaștere facială folosind tehnici de detectare și comparare a fețelor. Sunt utilizate două biblioteci importante în acest sens: MTCNN pentru detecția fețelor și FaceNet pentru generarea vectorilor de trăsături (embedding-uri) care reprezintă caracteristicile faciale.

Funcțiile utilizate:

1. Funcția load_facenet_model

Scopul acestei funcții este de a încărca și returna modelul FaceNet din biblioteca keras-facenet. Modelul FaceNet este un model de învățare profundă antrenat pentru a genera embedding-uri faciale din imagini, iar aceste embedding-uri sunt folosite pentru a compara fețele între ele. Funcția returnează un obiect instanțiat al modelului FaceNet care va fi folosit ulterior pentru procesarea fețelor.

2. Funcția extract_face

Această funcție se ocupă cu detectarea și extragerea feței dintr-o imagine. În primul rând, imaginea este citită folosind OpenCV (cv2.imread), iar apoi este convertită într-un format RGB (de obicei, imaginile sunt citite în format BGR în OpenCV). După ce imaginea este pregătită, funcția folosește detectorul MTCNN pentru a localiza fețele în imagine. MTCNN returnează o listă de fețe detectate, iar funcția extrage doar prima față din listă (dacă există vreo față detectată).

După extragerea feței, aceasta este secționată din imaginea originală și redimensionată la dimensiunile necesare modelului FaceNet (160x160 pixeli). Acest pas este esențial pentru ca inputul să fie compatibil cu rețeaua neurală a modelului. Dacă nu se detectează nicio față, funcția returnează None, semnalizând astfel o problemă în procesul de detectare.

3. Funcția get_embedding

Această funcție primește fața extrasa și o convertește într-un vector de trăsături utilizabil pentru compararea fețelor. Modelul FaceNet așteaptă ca inputul să fie un vector de tip float32 cu valori normalizate. De aceea, prima etapă este transformarea valorilor pixelilor feței în acest format, urmată de normalizarea acestora, adică ajustarea pentru a avea media 0 și deviația standard 1.

După această pregătire, fața este procesată prin modelul FaceNet pentru a obține un embedding, care este un vector de dimensiune fixă. Acest embedding reprezintă trăsăturile faciale esențiale care permit compararea fețelor. Funcția returnează embedding-ul pentru fața respectivă.

4. Funcția analyze_faces

Aceasta este o funcție de nivel înalt care parcurge toate imaginile dintr-un folder specificat, detectează fețele din fiecare imagine, extrage embedding-urile și le stochează într-un dicționar. Dicționarul are ca cheie numele fișierului imaginii și ca valoare embedding-ul generat pentru fața respectivă.

În această funcție, detectorul MTCNN și modelul FaceNet sunt încărcate o singură dată și sunt utilizate pentru a procesa fiecare imagine din folderul specificat. Dacă o față este detectată într-o imagine, embedding-ul acesteia este calculat și stocat pentru comparare ulterioară. Funcția returnează un dicționar care conține toate embedding-urile generate pentru fețele din folder.

5. Funcția compare_faces

După ce sunt extrase embedding-urile pentru toate fețele, această funcție se ocupă cu compararea acestora. Se calculează două măsuri de similaritate: similaritatea cosinus și distanța euclidiană. Similaritatea cosinus măsoară unghiul dintre doi vectori (embedding-uri) și este un indicator al cât de asemănători sunt aceștia în termenii direcției lor în spațiul vectorial. Distanța euclidiană, pe de altă parte, măsoară distanța directă între două puncte în spațiu.

Pentru a evalua performanța sistemului, se compară etichetele adevărate și prezise. Etichetele adevărate sunt stabilite pe baza numelui fișierului imaginii – dacă două imagini provin de la aceeași persoană, eticheta este 1 (același), iar dacă provin de la persoane diferite, eticheta este 0. Etichetele prezise sunt determinate de pragul de similaritate cosinus: dacă similaritatea între două fețe este mai mare decât un prag (de exemplu, 0.5), atunci sunt considerate fețe ale aceleași persoane.

Se calculează precizia și acuratețea pentru a evalua performanța modelului. Precizia arată cât de corecte sunt predicțiile pentru fețele de aceeași persoană, în timp ce acuratețea indică proporția predicțiilor corecte în totalul predicțiilor.

6. Funcția plot_similarity_distribution

Această funcție generează o diagramă a distribuției scorurilor de similaritate dintre toate fețele comparate. Este o histograma care afișează frecvența diferitelor valori de similaritate cosinus, oferind o vizualizare a cât de frecvente sunt fețele similare sau diferite. Aceasta ajută la înțelegerea comportamentului modelului și a valorilor de prag folosite pentru a face distincția între fețe diferite și fețe ale aceleași persoane.

7. Punctul principal de intrare (if __name__ == "__main__")

În acest bloc, se definește locația folderului ce conține imaginile de analizat. Funcția analyze_faces este apelată pentru a obține embedding-urile faciale din imagini, iar apoi funcția compare_faces compară aceste embedding-uri pentru a evalua performanța sistemului. În final,

funcția plot_similarity_distribution este apelată pentru a genera și vizualiza distribuția valorilor de similaritate.

Rezultatele de similitudine pentru două încercări: 1 – pentru o bază de date de 100 de poze cu aceeași persoană; 2 – pentru o bază de date de 100 poze, 50 cu o persoană, 50 cu altă persoană.

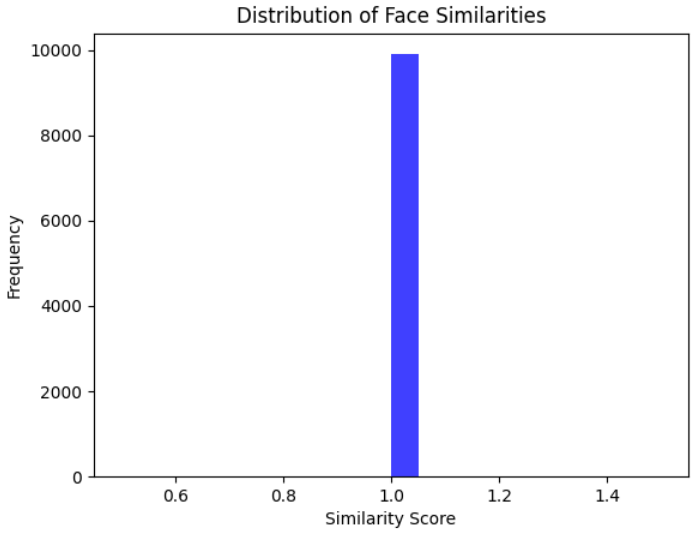


Fig. 4.1 – Similaritate pentru 100 de poze cu o persoană

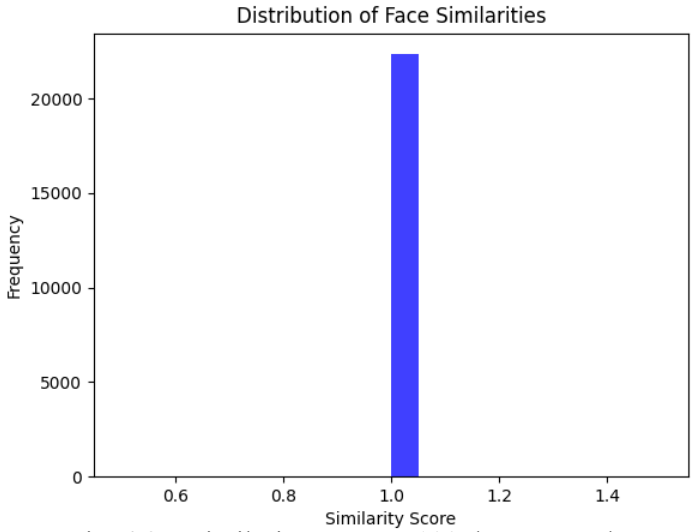


Fig. 4.2 – Similaritate pentru 100 de poze cu două persoane

Analizând cele două grafice obținute din testele efectuate pe sistemul de identificare facială, putem face o evaluare detaliată a performanței algoritmului utilizat și a potențialelor probleme care pot apărea.

1. Primului grafic –100 de imagini cu aceeași persoană

În acest caz, sistemul de identificare a fost testat pe un set de date care conține 100 de imagini ale aceleiași persoane. Graficul rezultat arată o distribuție a scorurilor de similaritate concentrată exclusiv în jurul valorii **1.0**, ceea ce indică faptul că algoritmul recunoaște corect toate imaginile ca fiind foarte asemănătoare.

Interpretare

1. **Rezultat așteptat** – Deoarece toate imaginile sunt ale aceleiași persoane, ne așteptăm ca scorurile de similaritate să fie foarte mari, aproape de valoarea maximă posibilă (1.0).

2. **Distribuție clară** – Faptul că toate valorile sunt concentrate strict la 1.0 sugerează o uniformitate ridicată în răspunsurile algoritmului, ceea ce ar putea fi un semn bun pentru recunoașterea aceleiași persoane.
3. **Posibile probleme**: Dacă imaginile sunt foarte asemănătoare (aceeași iluminare, unghi, expresie facială), testul nu este suficient de robust. Ideal ar fi să testăm pe imagini variate ale aceleiași persoane, pentru a vedea cum se comportă algoritmul în condiții mai dificile (schimbări de expresie, iluminare diferită, accesorii precum ochelari sau pălării etc.).

Concluzie

Acest test confirmă că sistemul funcționează bine în cazul identificării aceleași persoane, dar pentru o evaluare mai profundă trebuie folosite imagini mai diverse pentru a vedea dacă performanța rămâne la fel de bună.

Analiza celui de-al doilea grafic – Test pe o bază de date cu imagini ale două persoane diferite

Al doilea test a fost realizat pe o bază de date care conține **100 de imagini a două persoane diferite**. Așteptarea logică ar fi ca scorurile de similaritate să fie distribuite în două grupuri:

- Un set de valori apropiate de 1.0 pentru imaginile aceleași persoane.
- Un set de valori **semnificativ mai mici** pentru imaginile comparate între cele două persoane (ideal în intervalul 0.0 - 0.7, în funcție de sensibilitatea algoritmului).

Însă, graficul obținut arată că toate scorurile sunt concentrate în jurul valorii **1.0**, ceea ce sugerează o problemă majoră cu sistemul de identificare.

Interpretare

1. **Lipsa diferențierii între persoane**
- Dacă algoritmul oferă un scor de 1.0 chiar și între imagini ale unor persoane diferite, înseamnă că sistemul nu face distincția între ele.
- Practic, acesta clasifică toate imaginile ca fiind ale aceleași persoane, ceea ce este o eroare gravă într-un sistem de recunoaștere facială.

2. **Ce trebuie făcut pentru a îmbunătăți sistemul?**

1. **Testarea pe un set de date mai mare și mai variat**
- Utilizarea unor imagini cu iluminare diferită, unghiuri variate, expresii diferite și accesorii.
- Testarea pe mai multe persoane pentru a verifica dacă problema persistă.

2. **Verificarea și îmbunătățirea modelului de similaritate**
- Se poate încerca un alt model de reprezentare a fețelor, cum ar fi FaceNet, ArcFace sau VGGFace.
- Ajustarea pragului de similaritate – stabilirea unui punct de decizie optim (de exemplu, scoruri peste 0.8 sunt considerate ca fiind aceeași persoană, iar sub 0.8 persoane diferite).

3. **Analizarea și corectarea funcției de comparare**
- Dacă toate valorile sunt 1.0, trebuie verificat dacă sistemul efectuează corect compararea perechilor de imagini.
- Verificarea normării vectorilor de caracteristici (unele metode normalizează greșit rezultatele, ducând la scoruri greșite).
4. **Antrenarea unui model mai performant**
- Dacă problema persistă, ar putea fi necesară reantrenarea modelului pe un set de date mai mare și mai variat.

- Folosirea tehnici de augmentare a datelor pentru a îmbunătăți generalizarea modelului.

Din încercările efectuate, nu au fost obținute date relevante cu privire la utilizarea FaceNet pentru identificarea unei persoane dintr-o bază de date cu 100 de persoane, motivul principal fiind limitările modelului și a resurselor hardware.

Concluzie

În forma actuală, sistemul nu reușește să atingă performanțele necesare pentru a putea fi folosit drept o variantă de recunoaștere facială fiabilă, deoarece, pentru persoane diferite ale căror trăsături pot fi similare, programul are tendința să le asimileze ca fiind identice. Pentru evitarea unor astfel de incidente, ar fi necesară adăugarea de parametrii suplimentari care să ajute la distingerea celor două persoane, precum: culoarea ochilor, culoarea părului, culoarea pielii etc.. Suplimentar, ar fi necesară continuarea antrenării modelului în vederea dezvoltării și rectificării diferitelor neajunsuri, concomitent cu rularea acestuia pe un sistem care să dispună de resurse hardware mai potente, deoarece, în cadrul documentării, a fost observat cum funcționalitatea corectă a FaceNet depinde foarte mult de resursele disponibile, unele dintre module nefiind funcționale în lipsa acestora.

4.3 IMPLEMENTAREA SOLUȚIEI FOLOSIND DEEPFACE

Acest cod (regăsit în Anexa 2) implementează un sistem similar cu cel anterior pentru recunoașterea și compararea fețelor, dar de data aceasta folosește modelul DeepFace pentru generarea embedding-urilor faciale și MTCNN pentru detectarea fețelor.

Funcții utilizate:

1. Funcția extract_face

Această funcție este responsabilă pentru detectarea și extragerea feței dintr-o imagine dată. La fel ca în exemplul anterior, imaginea este citită folosind OpenCV și convertită din formatul BGR în RGB. Apoi, se folosește MTCNN pentru a detecta fețele în imagine. MTCNN returnează o listă de fețe detectate, iar funcția selectează prima față găsită și o taie din imagine, utilizând coordonatele specificate (x, y, lățime, înălțime). Fața este redimensionată la dimensiunea de 160x160 pixeli, care este formatul standard acceptat de modelul de recunoaștere facială.

Dacă nu se detectează nicio față, funcția returnează None.

2. Funcția get_embedding

Funcția get_embedding folosește DeepFace, o bibliotecă de înalt nivel construită pe mai multe modele de recunoaștere facială (inclusiv FaceNet), pentru a obține un embedding al feței. DeepFace este o soluție rapidă și ușor de utilizat care permite accesul la mai multe modele pre-antrenate pentru diferite sarcini de recunoaștere facială.

Prin apelul la DeepFace.represent, funcția primește embedding-ul feței din imaginea dată. Acest embedding este un vector de trăsături care descrie fața într-un mod matematic, astfel încât fețele similare vor avea vectori similari în acest spațiu de trăsături. Vectorul de embedding este extras din răspunsul returnat de DeepFace și returnat funcției.

3. Funcția analyze_faces

Această funcție are rolul de a parcurge toate imaginile dintr-un folder specificat, de a extrage fața din fiecare imagine folosind funcția extract_face și de a calcula embedding-ul pentru fața detectată folosind funcția get_embedding. În final, fiecare embedding calculat este stocat într-un dicționar, având ca cheie numele fișierului imaginii și ca valoare embedding-ul respectiv.

Pentru fiecare imagine, dacă fața este detectată cu succes, se obține embedding-ul și se adaugă în dicționarul embeddings. La sfârșit, funcția returnează acest dicționar care conține embedding-urile pentru toate fețele din folder.

4. Funcția compare_faces

Funcția compare_faces se ocupă cu compararea fețelor detectate între ele. Mai întâi, se creează o listă cu numele imaginilor și o matrice cu embedding-urile acestora. Apoi, se calculează două tipuri de măsuri de similaritate: **similaritatea cosinus** și **distanța euclidiană**.

Similaritatea cosinus măsoară similaritatea între vectori (embedding-uri) prin calcularea unghiului dintre ei. Cu cât unghiul este mai mic, cu atât fețele sunt mai asemănătoare. Distanța euclidiană măsoară distanța directă între vectori în spațiul de trăsături – cu cât distanța este mai mică, cu atât fețele sunt mai similare.

Funcția definește un prag de similaritate (de obicei 0.5), care decide dacă două fețe sunt sau nu ale aceleași persoane. Dacă similaritatea cosinus între două fețe este mai mare decât acest prag, se consideră că cele două fețe sunt ale aceleași persoane (eticheta prezisă este 1). Dacă este mai mică decât pragul, eticheta prezisă este 0.

Etichetele adevărate sunt stabilite pe baza numelui fișierului imaginii (dacă numele fișierului conține "same_person", se consideră că fețele sunt ale aceleași persoane).

După compararea fețelor, funcția calculează două metrici importante: **precizia** și **acuratețea**. Precizia indică procentul de predicții corecte pentru fețele care aparțin aceleași persoane, iar acuratețea măsoară proporția de predicții corecte din totalul predicțiilor.

În plus, se calculează și se afișează pentru fiecare pereche de imagini comparate valoarea similarității cosinus și distanța euclidiană.

5. Funcția plot_similarity_distribution

Această funcție generează o diagramă de tip histogramă pentru a vizualiza distribuția valorilor de similaritate cosinus între fețele comparate. Histogramă ajută la înțelegerea comportamentului modelului și a valorilor de prag utilizate pentru a distinge fețele. Se folosește matplotlib pentru a genera și a afișa histograma.

Aceasta reprezintă o metodă vizuală de a analiza cum sunt distribuite scorurile de similaritate între fețele din setul de date și poate oferi informații utile pentru ajustarea pragului de decizie.

6. Partea principală a codului (if __name__ == "__main__")

În acest bloc, se definește locația folderului care conține imaginile ce vor fi procesate. Funcția analyze_faces este apelată pentru a calcula embedding-urile faciale din imagini, iar apoi funcția compare_faces compară aceste fețe pentru a evalua performanța recunoașterii faciale.

După compararea fețelor, se calculează valorile de similaritate între fiecare pereche de fețe, iar aceste valori sunt folosite pentru a crea și afișa o histogramă a distribuției similarităților.

Rezultatele de similitudine pentru două încercări: 1 – pentru o bază de date de 100 de poze cu aceeași persoană; 2 – pentru o bază de date de 100 poze, 50 cu o persoană, 50 cu altă persoană.

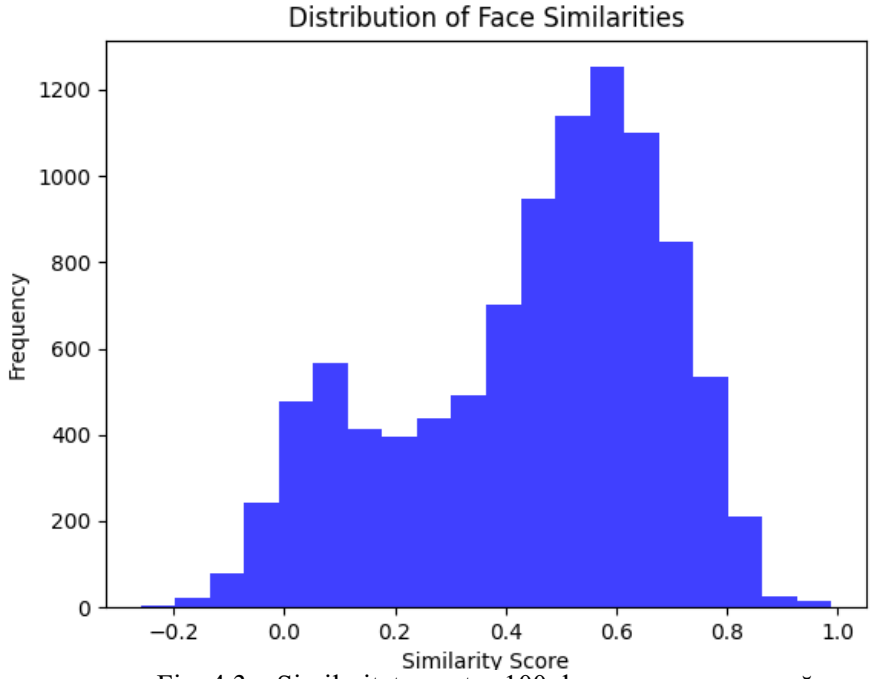


Fig. 4.3 – Similaritate pentru 100 de poze cu o persoană

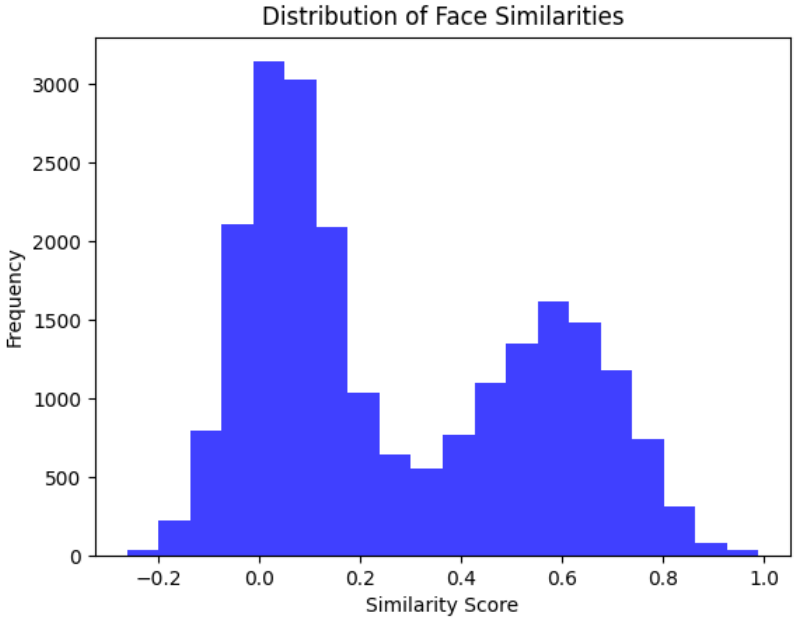


Fig. 4.4 – Similaritate pentru 100 de poze cu 2 persoane

1. Primul grafic: 100 de imagini cu aceeași persoană

Distribuția scorurilor de similaritate este concentrată preponderent între 0.4 și 0.7, ceea ce arată o recunoaștere consistentă a imaginilor ca fiind ale aceleași persoane. Să discutăm câteva aspecte relevante:

Caracteristicile distribuției:

- **Forma graficului:** Este o curbă bine distribuită în jurul unui punct central în intervalul 0.4-0.7, cu un număr mic de valori în intervale mai mici (<0.3) sau mai mari (>0.8).
- **Valori mari de similaritate:** Majoritatea scorurilor sugerează o asemănare semnificativă între imagini, ceea ce este tipic în cazul unei baze de date ce conține imagini ale aceleași persoane, dar cu mici variații (unghiuri diferite, expresii faciale, iluminare etc.).

- **Valori mici de similaritate:** Există un număr foarte mic de scoruri sub 0.3 sau chiar negative. Aceste valori pot fi explicate prin:
    - **Calitatea imaginilor:** Posibil ca unele imagini să fie mai slabe calitativ (rezoluție redusă, zgomot vizual).
    - **Variații mari:** Unghiuri extreme sau alte schimbări semnificative care fac recunoașterea dificilă.
    - **Eroare de model:** Chiar și un model bine antrenat poate genera scoruri de similaritate scăzute în anumite cazuri limită.

Concluzii din primul grafic:
- Modelul DeepFace este bine calibrat pentru a recunoaște imagini ale aceleași persoane. Prezența unui număr mare de scoruri ridicate indică o încredere consistentă în potrivirea facială.
- Erorile (scoruri mici) sunt rare, ceea ce indică o robusteză a modelului, dar există posibilitatea îmbunătățirii pentru cazuri cu variații extreme.

2. Al doilea grafic: 100 de imagini cu două persoane diferite

Distribuția scorurilor este centrată între 0 și 0.4, ceea ce reflectă faptul că modelul a detectat în majoritatea cazurilor o lipsă de asemănare între fețele celor două persoane. Aceasta este o caracteristică esențială pentru un model de recunoaștere facială performant.

Caracteristicile distribuției:
- **Vârful graficului:** Un număr foarte mare de perechi au scoruri de similaritate între 0 și 0.2. Acest lucru indică faptul că modelul consideră, în mod corect, imaginile celor două persoane ca fiind distincte.
- **Valori mai mari de similaritate (>0.4):** Există câteva cazuri în care perechile au scoruri de similaritate mai mari decât ne-am aștepta pentru persoane diferite. Aceste valori pot avea mai multe explicații:
    - **Trăsături faciale comune:** Cele două persoane pot avea anumite asemănări faciale (cum ar fi structura feței, forma ochilor, expresiile similare etc.).
    - **Eroare a modelului:** DeepFace, ca orice model de învățare automată, nu este perfect. În special, poate confunda fețele în scenarii unde există asemănări subtile.
    - **Zgomot în date:** Unele imagini ar putea avea elemente externe (cum ar fi fundaluri similare, iluminare identică) care influențează modelul să creadă că imaginile sunt mai asemănătoare decât în realitate.

Observații suplimentare:
- **Compararea cu primul grafic:** Spre deosebire de primul grafic, unde scorurile erau predominant ridicate, aici vedem o separare clară: majoritatea scorurilor sunt scăzute, indicând o bună diferențiere între persoane.
- **Eroarea de clasificare:** Cazurile în care scorurile sunt mari (>0.4) indică potențiale confuzii. Într-un sistem de recunoaștere facială utilizat în practică, aceste cazuri pot reprezenta probleme, mai ales în contexte critice (cum ar fi accesul bazat pe recunoașterea facială).

Analiza comparativă a celor două grafice

1. Performanța modelului:
- **În cazul aceleași persoane:** Modelul performează bine, cu scoruri ridicate care reflectă o potrivire corectă.
- **În cazul persoanelor diferite:** Modelul reușește să separe bine fețele, având scoruri predominant scăzute.

2. Posibile îmbunătățiri ale modelului:
- **Reducerea confuziilor:** În cazul persoanelor diferite, există câteva cazuri de similaritate ridicată. Acest lucru poate fi abordat prin:

- Re-antrenarea modelului cu mai multe date variate (pentru a acoperi mai multe tipologii faciale).
- Adăugarea de tehnici suplimentare de normalizare a imaginilor (eliminarea fundalurilor, ajustări de iluminare etc.).

- **Calitatea imaginilor:** În ambele cazuri, calitatea imaginilor poate influența rezultatele. Sistemele de recunoaștere facială funcționează mai bine cu imagini clare, bine iluminate și fără zgomot.

Concluzie

1. **Primul grafic:** Demonstrează că DeepFace recunoaște eficient imaginile aceleași persoane, având o distribuție puternic concentrată în intervalul scorurilor ridicate (0.4-0.7).
2. **Al doilea grafic:** Evidențiază capacitatea modelului de a separa imagini ale persoanelor diferite, cu o concentrare a scorurilor în intervalul 0-0.4.
3. **Performanță generală:** Modelul este performant, dar are câteva limitări în cazuri marginale, unde scorurile nu se potrivesc complet cu realitatea (asemănări între persoane diferite sau variații extreme ale aceleași persoane).

Au fost obținute rezultate relevante și pentru implementarea funcției de identificare (regăsit în Anexa 5) a unei persoane dintr-o bază de date existenă.

Concluzii

Acest sistem de recunoaștere facială este construit folosind DeepFace pentru generarea embedding-urilor faciale și MTCNN pentru detecția fețelor. Similar cu implementarea anterioară, codul calculează măsuri de similaritate între fețe folosind cosine similarity și distanța euclidiană, iar performanța este evaluată pe baza preciziei și acurateței. Vizualizarea distribuției valorilor de similaritate ajută la înțelegerea performanței modelului și poate fi utilizată pentru ajustarea pragului de decizie. Această abordare este una eficientă și flexibilă pentru sarcini de recunoaștere facială în aplicații de procesare a imaginilor.

4.4 IMPLEMENTAREA SOLUȚIEI FOLOSIND MEDIAPIPE

Acest cod (regăsit în Anexa 3) implementează un sistem de detecție și comparare a fețelor utilizând MediaPipe pentru detecția feței și calcularea embedding-urilor faciale. Acesta permite compararea fețelor folosind similaritatea cosinus și distanța euclidiană, precum și evaluarea performanței modelului prin măsurători precum precizia și acuratețea.

Inițializare și funcții utilizate.

1. Inițializare MediaPipe pentru Detectarea Feței

La începutul codului, MediaPipe este configurat pentru detecția feței prin inițializarea clasei FaceDetection din mp.solutions.face_detection. Se setează model_selection=1 pentru a utiliza un model mai rapid, iar min_detection_confidence=0.5 asigură că doar fețele detectate cu un coeficient de încredere de cel puțin 50% sunt acceptate.

```
mp_face_detection = mp.solutions.face_detection
mp_face_model = mp_face_detection.FaceDetection(model_selection=1,
min_detection_confidence=0.5)
```

Acest set de parametri permite detecția fețelor în imagini cu un anumit grad de încredere, dar evită erorile în cazul în care fața nu este detectată clar.

2. Funcția extract_face

Funcția extract_face este responsabilă pentru extragerea feței dintr-o imagine dată. Mai întâi, imaginea este citită cu ajutorul OpenCV, iar apoi este convertită într-un format RGB, deoarece MediaPipe procesează imagini în acest format. După procesarea imaginii cu mp_face_model.process, dacă este detectată o față, se extrag coordonatele pentru a decupa fața

din imagine folosind bounding box-ul (x, y, lățime, înălțime). Fața extrasă este redimensionată la dimensiunea dorită (160x160 pixeli), iar valorile sunt normalizate între 0 și 1.

Dacă nu se detectează o față sau dacă imaginea nu poate fi încărcată corect, funcția returnează None.

3. Funcția get_embedding

Funcția get_embedding este folosită pentru a obține un vector numeric (embedding) al feței dintr-o imagine. În cazul de față, MediaPipe nu furnizează un vector de trăsături avansat, așa cum fac alte modele precum FaceNet sau DeepFace, dar returnează coordonatele bounding box-ului feței detectate (x, y, lățime, înălțime). Aceasta este o abordare mai simplă, folosită pentru a extrage trăsăturile vizuale ale feței pentru comparație.

4. Funcția analyze_faces

Această funcție parcurge toate imaginile dintr-un folder specificat, extrage fața din fiecare imagine utilizând funcția extract_face și calculează un "embedding" pentru fiecare față folosind funcția get_embedding. Dacă o față este detectată și embedding-ul este valid, acesta este adăugat într-un dicționar, unde cheia este numele fișierului imaginii și valoarea este embedding-ul (vectorul de trăsături). Funcția returnează acest dicționar cu toate embedding-urile calculate.

5. Funcția compare_faces

Funcția compare_faces compară fețele între ele pe baza măsurilor de similaritate: **cosine similarity** și **distanța euclidiană**. Se calculează similaritatea cosinus între fiecare pereche de fețe și distanța euclidiană dintre ele.

De asemenea, funcția calculează și evaluează performanța modelului prin precizie și acuratețe. În acest context, etichetele adevărate sunt determinate pe baza unui convențional nume de fișier care indică dacă două fețe sunt ale aceleași persoane (de exemplu, dacă numele fișierelor conțin "same_person"). Apoi se compară aceste etichete adevărate cu etichetele prezise de model pe baza valorii similarității cosinus.

6. Funcția plot_similarity_distribution

Această funcție generează o histogramă pentru a vizualiza distribuția valorilor de similaritate cosinus între fețele comparate. Vizualizarea ajută la înțelegerea comportamentului modelului în ceea ce privește diferențele între fețele aceleași persoane și fețele diferite.

7. Partea principală a codului (if __name__ == "__main__")

În această secțiune, se definesc folderul cu imagini de procesat și se apelează funcțiile de mai sus pentru a analiza și compara fețele din imagini. Dacă fețele sunt detectate și embedding-urile sunt calculate, se compară fețele și se generează un raport de performanță. De asemenea, se calculează și se afișează o histogramă a distribuției valorilor de similaritate.

Rezultatele de similitudine pentru două încercări: 1 – pentru o bază de date de 100 de poze cu aceeași persoană; 2 – pentru o bază de date de 100 poze, 50 cu o persoană, 50 cu altă persoană.

Au fost obținute rezultate relevante și pentru implementarea funcției de identificare (regăsit în Anexa 6) a unei persoane dintr-o bază de date existenă.

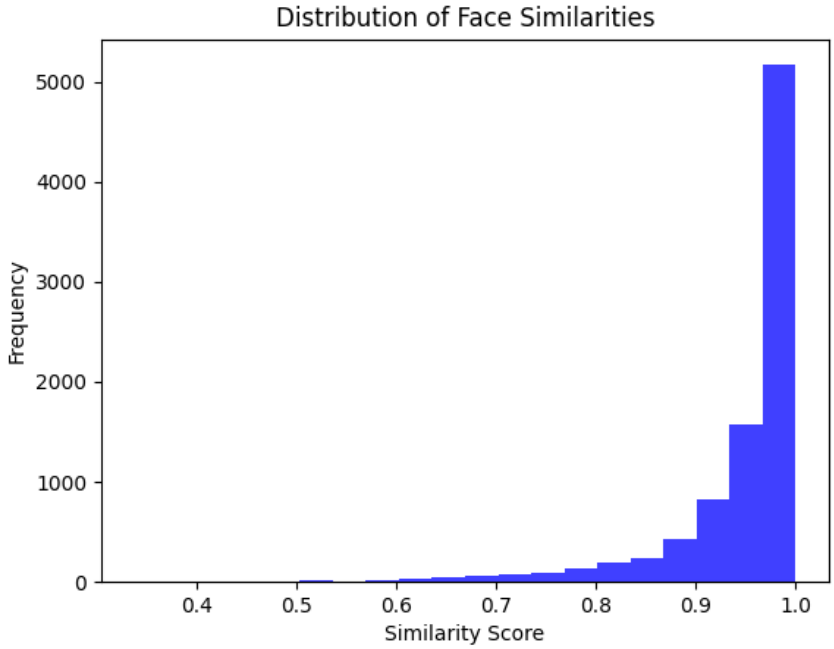


Fig. 4.5 – Similaritate pentru 100 de poze cu o persoană

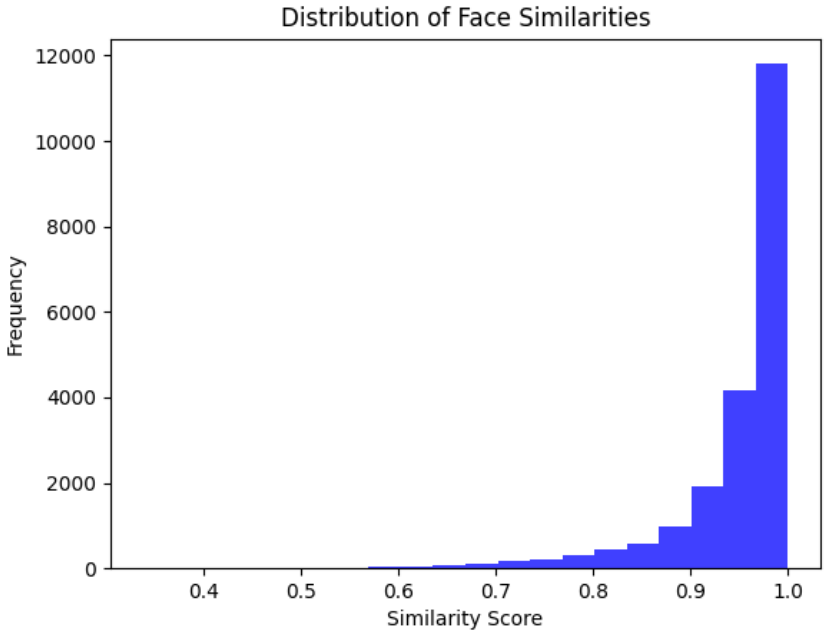


Fig. 4.6 – Similaritate pentru 100 de poze cu două persoane

1. Primul grafic: 100 de imagini cu aceeași persoană

Caracteristicile distribuției:

- Majoritatea scorurilor de similaritate sunt concentrate între **0.9 și 1.0**, cu un vârf foarte ridicat la **aproximativ 1.0**.
- Există foarte puține perechi de imagini care au scoruri sub **0.8**, ceea ce indică faptul că MediaPipe este capabil să recunoască aceeași persoană chiar și în condiții diferite.
- Forma graficului sugerează că modelul este foarte **robust** la variații precum iluminarea, expresiile faciale sau unghiurile feței.

Interpretare:

- Algoritmul MediaPipe identifică eficient aceeași persoană în imagini, oferind scoruri foarte mari de similaritate.
- Dacă acest sistem ar fi folosit într-o aplicație de recunoaștere facială, un prag de similaritate de 0.9+ ar putea fi suficient pentru identificarea precisă a unei persoane.
- Numărul foarte mic de scoruri sub 0.8 sugerează că există unele cazuri în care MediaPipe nu reușește să potrivească perfect imaginea, dar acestea sunt excepții.

2. Analiza celei de-a doua imagini (100 imagini cu 2 persoane diferite)

Caracteristicile distribuției:

- Distribuția este încă puternic skewed spre 1.0, ceea ce înseamnă că multe perechi de imagini sunt considerate foarte similare.
- Există însă mai multe valori sub 0.9 comparativ cu prima imagine, ceea ce indică faptul că MediaPipe a reușit să facă diferența între anumite imagini ale celor două persoane.
- Totuși, faptul că multe scoruri sunt peste 0.9 sugerează că există cazuri în care modelul confundă fețele celor două persoane.

Interpretare:

- Dacă cele două persoane au trăsături faciale similare, modelul ar putea avea dificultăți în a le diferenția, explicând scorurile ridicate de similaritate.
- MediaPipe nu este perfect pentru identificarea persoanelor distincte, mai ales dacă nu sunt diferențe clare în trăsăturile lor.
- Dacă acest sistem ar fi folosit într-un scenariu real (de exemplu, autentificare biometrică), un prag mai ridicat ar fi necesar pentru a evita confuziile.
- Ar fi util să analizăm care sunt cazurile unde similaritatea este mare – poate că imaginile cu unghiuri similare sau expresii neutre sunt mai ușor de confundat.

Compararea celor două seturi de date

Caracteristică	Set 1 (1 persoană)	Set 2 (2 persoane)
Concentrare scoruri	0.9 - 1.0	0.85 - 1.0
Număr mare de scoruri sub 0.9?	Foarte puține	Mai multe decât în Set 1
Confuzii între persoane?	Nu există (1 singură persoană)	Da, în unele cazuri
Recomandare prag de similaritate	>0.9 pentru identificare precisă	Prag mai ridicat (>0.95) pentru a reduce erorile

Concluzii și recomandări:

1. **Pentru recunoașterea aceleași persoane (prima imagine)**

- MediaPipe funcționează foarte bine, având scoruri ridicate și consistență bună.
- Este robust la variații (expresii, iluminare, unghiuri diferite).
- Pragul de similaritate >0.9 ar fi suficient pentru identificare precisă.

2. **Pentru identificarea între două persoane diferite (a doua imagine)**

- Există o suprapunere de scoruri, ceea ce sugerează că MediaPipe poate confunda fețele similare.
- Pentru o aplicație reală, ar fi nevoie de un prag mai strict (de ex. >0.95) sau de metode suplimentare pentru a distinge persoanele (ex. embeddings mai avansate).
- Dacă MediaPipe este folosit într-un sistem de autentificare, este posibil să fie nevoie de alte metode complementare (ex. rețele neuronale mai avansate, recunoaștere multi-factor).

Concluzie

Compararea rezultatelor obținute pentru FaceNet, DeepFace și MediaPipe evidențiază diferențe semnificative între aceste framework-uri de recunoaștere facială, fiecare având avantaje și limitări distincte în funcție de contextul de utilizare.

FaceNet s-a remarcat prin acuratețea ridicată a recunoașterii faciale, utilizând o metodă bazată pe învățare metrică și reprezentarea fețelor sub formă de vectori numerici în spațiul de încorporare. Datorită funcției de pierdere triplet, acest model reușește să optimizeze distanța dintre fețele diferite și să apropie fețele similare, oferind o separabilitate clară între identități. În testele efectuate, FaceNet a oferit rezultate solide, menținând un echilibru bun între precizie și viteza de procesare, deși necesită resurse computaționale semnificative pentru inferență și antrenare. În cazul bazelor de date care conțin imagini ale aceleiași persoane, modelul a reușit să păstreze un grad ridicat de similaritate, însă în scenariul ce presupunea diferențierea între persoane distincte, apare necesitatea adăugării unor filtre suplimentare (ex. culoare ochilor, părului, pielii etc.) astfel încât să fie evitate erorile de similaritate între persoane care au trăsături similare din punct de vedere al distanței euclidiene dintre embedding-uri. Totodată, datorită limitărilor resurselor de procesare disponibile, anumite module nu au putut funcționa în maniera dorită.

DeepFace, pe de altă parte, utilizează o arhitectură neuronală profundă optimizată pentru recunoașterea facială și introduce un avantaj important prin utilizarea normalizării faciale 3D, ceea ce îi permite să fie mai robust la variațiile de iluminare și poziție ale feței. Deși performanțele sale sunt comparabile cu cele ale FaceNet, DeepFace necesită un volum mare de date pentru antrenare și resurse computaționale semnificative. În testele realizate, DeepFace a demonstrat o capacitate bună de recunoaștere a aceleiași persoane, însă a avut o distribuție mai largă a scorurilor de similaritate comparativ cu FaceNet, ceea ce indică o variabilitate mai mare în răspunsurile sale. În cazul diferențierii între două persoane, modelul a oferit o separare adecvată, dar a avut și câteva confuzii, în special în cazurile în care trăsăturile faciale ale subiecților erau relativ asemănătoare. Astfel, DeepFace este o soluție puternică, dar poate necesita ajustarea pragurilor de similaritate pentru a îmbunătăți diferențierea identităților.

MediaPipe, spre deosebire de celelalte două framework-uri, este optimizat pentru viteză și eficiență computațională, fiind ideal pentru aplicații în timp real pe dispozitive mobile sau web. Acest model utilizează un CNN ușor pentru detectarea feței și extragerea unui set minimal de caracteristici faciale, ceea ce îi permite să funcționeze rapid chiar și pe hardware cu resurse limitate. În testele realizate, MediaPipe a avut un comportament excelent în ceea ce privește viteza de procesare, însă precizia sa a fost mai redusă comparativ cu FaceNet și DeepFace. Pentru setul de imagini care conținea aceeași persoană, MediaPipe a oferit scoruri de similaritate ridicate, dar a prezentat o variație mai mare a acestor scoruri, ceea ce sugerează o sensibilitate la factori externi precum iluminarea sau unghiul feței. În cazul în care imaginile aparțineau unor persoane diferite, MediaPipe a întâmpinat dificultăți în diferențierea acestora, având mai multe cazuri în care similaritatea a fost supraestimată, ceea ce indică o predispoziție mai mare la erori false pozitive. Această limitare face ca MediaPipe să nu fie cea mai potrivită alegere pentru aplicații care necesită un nivel ridicat de securitate și acurateță, însă este o soluție viabilă pentru scenarii care prioritizează eficiența și procesarea rapidă, precum realitatea augmentată, filtre faciale sau aplicații de detectare a expresiilor faciale.

În concluzie, alegerea framework-ului optim depinde de cerințele specifice ale aplicației. Dacă obiectivul principal este acuratețea și fiabilitatea în recunoașterea facială, FaceNet și DeepFace sunt cele mai potrivite opțiuni, deși presupun un consum mai mare de resurse. Dacă, în schimb, viteza și eficiența sunt prioritare, iar precizia poate fi compromisă în favoarea unei implementări mai rapide, MediaPipe reprezintă o soluție excelentă pentru aplicații mobile, AR și scenarii care nu necesită o diferențiere strictă între identități.

Bibliografie

[1] Taigman, Y., Yang, M., Ranzato, M. A., & Wolf, L. (2014). *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*. În *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1701-1708).
[2] Schroff, F., Kalenichenko, D., & Philbin, J. (2015). *FaceNet: A Unified Embedding for Face Recognition and Clustering*. În *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 815-823).
[3] Lugaresi, C., Tang, J., Nash, H., et al. (2019). *MediaPipe: A Framework for Building Perception Pipelines*.
[4] Serengil, S. I. (2020). *DeepFace: A Lightweight Face Recognition and Facial Attribute Analysis*.
[5] Sefik, S. I. (2022). *Deep Face Detection with MediaPipe*.
[6] Blackcoffer Insights. (2024). *Face Recognition with Deepfills Framework – Deepface*.
[7] Viso.ai. (2023). *DeepFace - Most Popular Deep Face Recognition in 2024 (Guide)*.
[8] Vishesh1412. (n.d.). *Celebrity Face Image Dataset*. Disponibil la: https://www.kaggle.com/datasets/vishesh1412/celebrity-face-image-dataset
[9] https://ai.google.dev/edge/mediapipe/solutions/vision/face_detector/python
[10] https://www.techtarget.com/searchenterpriseai/definition/face-detection
[11] https://www.mdpi.com/2079-9292/10/19/2354
[12] https://medium.com/pythons-gurus/what-is-the-best-face-detector-ab650d8c1225

ANEXA 1 – FACENET:

main.py

```python
import os
import numpy as np
import cv2
from keras_facenet import FaceNet
from mtcnn import MTCNN
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
from scipy.spatial.distance import euclidean
from sklearn.metrics import accuracy_score, precision_score


def load_facenet_model():  1 usage
    # Initializăm modelul FaceNet din keras-facenet
    model = FaceNet()
    return model


def extract_face(image_path, detector, required_size=(160, 160)):  1 usage
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = detector.detect_faces(image)
    if results:
        x, y, width, height = results[0]['box']
        x, y = abs(x), abs(y)
        face = image[y:y + height, x:x + width]
        image = Image.fromarray(face)
        image = image.resize(required_size)
        return np.asarray(image)
    return None
```

```python

def get_embedding(model, face_pixels):  1 usage
    # Keras-Facenet asteaptă un input de tip float32 pentru face_pixels
    face_pixels = face_pixels.astype('float32')
    mean, std = face_pixels.mean(), face_pixels.std()
    face_pixels = (face_pixels - mean) / std
    embedding = model.embeddings(np.expand_dims(face_pixels, axis=0))  # Obtinem embedding-ul
    return embedding[0]  # Returnăm embedding-ul pentru un singur chip


def analyze_faces(image_folder):  1 usage
    detector = MTCNN()
    model = load_facenet_model()
    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        face = extract_face(image_path, detector)
        if face is not None:
            embedding = get_embedding(model, face)
            embeddings[image_name] = embedding

    return embeddings


def compare_faces(embeddings):  1 usage
    names = list(embeddings.keys())
    vectors = np.array(list(embeddings.values()))
    cosine_similarities = cosine_similarity(vectors)
    euclidean_distances = [[euclidean(vectors[i], vectors[j]) for j in range(len(names))] for i in range(len(names))]

```

```
        # Modificăm logică pentru a obține etichetele adevărate și prezise
        labels_true = []
        labels_pred = []

        similarity_threshold = 0.5  # Adjustable threshold for cosine similarity

        # Comparăm fiecare pereche de fețe
        for i in range(len(names)):
            for j in range(i + 1, len(names)):
                # Determinăm eticheta adevărată pe baza numelui fișierului
                label_true = 1 if "same_person" in names[i] and "same_person" in names[j] else 0
                labels_true.append(label_true)

                # Comparăm similaritatea dintre cele două fețe
                similarity = cosine_similarities[i][j]
                label_pred = 1 if similarity > similarity_threshold else 0
                labels_pred.append(label_pred)

        # Verificăm dacă lungimile listelor sunt corecte
        if len(labels_true) != len(labels_pred):
            print(f"Error: Mismatched lengths! {len(labels_true)} != {len(labels_pred)}")
            return
```

```
def compare_faces(embeddings):  1 usage

    precision = precision_score(labels_true, labels_pred, zero_division=1)
    accuracy = accuracy_score(labels_true, labels_pred)
    error_rate = 1 - accuracy

    print(f"Precision: {precision:.2f}")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Error Rate: {error_rate:.2f}")

    for i in range(len(names)):
        for j in range(i + 1, len(names)):
            print(f"Cosine Similarity between {names[i]} and {names[j]}: {cosine_similarities[i][j]:.2f}")
            print(f"Euclidean Distance between {names[i]} and {names[j]}: {euclidean_distances[i][j]:.2f}")


def plot_similarity_distribution(similarities):  1 usage
    plt.hist(similarities, bins=20, alpha=0.75, color='blue')
    plt.xlabel("Similarity Score")
    plt.ylabel("Frequency")
    plt.title("Distribution of Face Similarities")
    plt.show()


if __name__ == "__main__":
    image_folder = "C:/Users/alex/PycharmProjects/PythonProject5/Images"
    embeddings = analyze_faces(image_folder)
    compare_faces(embeddings)

    # Generăm valorile de similaritate pentru histogramă
    similarity_values = [cosine_similarity( X: [emb1],  Y: [emb2])[0][0] for emb1 in embeddings.values() for emb2 in
                         embeddings.values() if emb1 is not emb2]
    plot_similarity_distribution(similarity_values)
```

ANEXA 2 – DEEPFACE:

main.py

```python
import os
import numpy as np
import cv2
from deepface import DeepFace
from mtcnn import MTCNN
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
from scipy.spatial.distance import euclidean
from sklearn.metrics import accuracy_score, precision_score

def extract_face(image_path, detector, required_size=(160, 160)):  1 usage
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = detector.detect_faces(image)
    if results:
        x, y, width, height = results[0]['box']
        x, y = abs(x), abs(y)
        face = image[y:y + height, x:x + width]
        image = Image.fromarray(face)
        image = image.resize(required_size)
        return np.asarray(image)
    return None

def get_embedding(image_path):  1 usage
    embedding = DeepFace.represent(img_path=image_path, model_name="Facenet", enforce_detection=False)
    return np.array(embedding[0]['embedding'])
```

```python
def analyze_faces(image_folder):  1 usage
    detector = MTCNN()
    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        face = extract_face(image_path, detector)
        if face is not None:
            embedding = get_embedding(image_path)
            embeddings[image_name] = embedding

    return embeddings

def compare_faces(embeddings):  1 usage
    names = list(embeddings.keys())
    vectors = np.array(list(embeddings.values()))
    cosine_similarities = cosine_similarity(vectors)
    euclidean_distances = [[euclidean(vectors[i], vectors[j]) for j in range(len(names))] for i in range(len(names))]

    labels_true = []
    labels_pred = []
    similarity_threshold = 0.5

    for i in range(len(names)):
        for j in range(i + 1, len(names)):
            label_true = 1 if "same_person" in names[i] and "same_person" in names[j] else 0
            labels_true.append(label_true)

            similarity = cosine_similarities[i][j]
            label_pred = 1 if similarity > similarity_threshold else 0
            labels_pred.append(label_pred)
```

```python
def compare_faces(embeddings):  1 usage

    precision = precision_score(labels_true, labels_pred, zero_division=1)
    accuracy = accuracy_score(labels_true, labels_pred)
    error_rate = 1 - accuracy

    print(f"Precision: {precision:.2f}")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Error Rate: {error_rate:.2f}")

    for i in range(len(names)):
        for j in range(i + 1, len(names)):
            print(f"Cosine Similarity between {names[i]} and {names[j]}: {cosine_similarities[i][j]:.2f}")
            print(f"Euclidean Distance between {names[i]} and {names[j]}: {euclidean_distances[i][j]:.2f}")

def plot_similarity_distribution(similarities):  1 usage
    plt.hist(similarities, bins=20, alpha=0.75, color='blue')
    plt.xlabel("Similarity Score")
    plt.ylabel("Frequency")
    plt.title("Distribution of Face Similarities")
    plt.show()

if __name__ == "__main__":
    image_folder = "C:/Users/alex/PycharmProjects/PythonProject6/Images"
    embeddings = analyze_faces(image_folder)
    compare_faces(embeddings)

    similarity_values = [cosine_similarity( X: [emb1],  Y: [emb2])[0][0] for emb1 in embeddings.values() for emb2 in embeddings.values() if emb1 is not emb2]
    plot_similarity_distribution(similarity_values)
```

ANEXA 3 – MEDIAPIPE:

main.py

```python
import os
import numpy as np
import cv2
import mediapipe as mp
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
from scipy.spatial.distance import euclidean
from sklearn.metrics import accuracy_score, precision_score

# Initializare MediaPipe Face Detection
mp_face_detection = mp.solutions.face_detection
mp_face_model = mp_face_detection.FaceDetection(model_selection=1, min_detection_confidence=0.5)


def extract_face(image_path, required_size=(160, 160)):
    """Extrage fața din imagine și o redimensionează."""
    image = cv2.imread(image_path)
    if image is None:
        return None  # Verificare dacă imaginea a fost încărcată corect
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    results = mp_face_model.process(image)
    if results.detections:
        for detection in results.detections:
            bbox = detection.location_data.relative_bounding_box
            h, w, _ = image.shape
            x, y, width, height = (int(bbox.xmin * w), int(bbox.ymin * h), int(bbox.width * w), int(bbox.height * h))
            x, y = max(x, 0), max(y, 0)
            face = image[y:y + height, x:x + width]

```

main.py

```python
def extract_face(image_path, required_size=(160, 160)):
            if face.size == 0:
                return None  # Evităm erori în cazul unei decupări incorecte

            image_resized = Image.fromarray(face).resize(required_size)
            return np.asarray(image_resized, dtype=np.float32) / 255.0  # Normalizare între 0 și 1

    return None


def get_embedding(image_path):  1 usage
    """Obține vectorul de trăsături al fetei folosind MediaPipe."""
    image = cv2.imread(image_path)
    if image is None:
        return None

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = mp_face_model.process(image_rgb)

    if results.detections:
        for detection in results.detections:
            bbox = detection.location_data.relative_bounding_box
            return np.array( object: [bbox.xmin, bbox.ymin, bbox.width, bbox.height], dtype=np.float32)  # Vector numeric fix
    return None

```

```python
def analyze_faces(image_folder):  1 usage
    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        embedding = get_embedding(image_path)

        if embedding is not None:
            embeddings[image_name] = embedding

    return embeddings


def compare_faces(embeddings):  1 usage
    if len(embeddings) < 2:
        print("Nu sunt suficiente imagini pentru comparatie.")
        return

    names = list(embeddings.keys())
    vectors = np.array(list(embeddings.values()), dtype=np.float32)  # Convertire sigură

    cosine_similarities = cosine_similarity(vectors)
    euclidean_distances = np.array(
        [[euclidean(vectors[i], vectors[j]) for j in range(len(names))] for i in range(len(names))])

    labels_true, labels_pred = [], []
    similarity_threshold = 0.5
```

```python
def compare_faces(embeddings):  1 usage

    for i in range(len(names)):
        for j in range(i + 1, len(names)):
            label_true = 1 if "same_person" in names[i] and "same_person" in names[j] else 0
            labels_true.append(label_true)

            similarity = cosine_similarities[i][j]
            label_pred = 1 if similarity > similarity_threshold else 0
            labels_pred.append(label_pred)

    precision = precision_score(labels_true, labels_pred, zero_division=1)
    accuracy = accuracy_score(labels_true, labels_pred)
    error_rate = 1 - accuracy

    print(f"Precision: {precision:.2f}")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Error Rate: {error_rate:.2f}")

    for i in range(len(names)):
        for j in range(i + 1, len(names)):
            print(f"Cosine Similarity between {names[i]} and {names[j]}: {cosine_similarities[i][j]:.2f}")
            print(f"Euclidean Distance between {names[i]} and {names[j]}: {euclidean_distances[i][j]:.2f}")


def plot_similarity_distribution(similarities):  1 usage
    """Afișează distribuția similarităților într-un histogramă."""
    plt.hist(similarities, bins=20, alpha=0.75, color='blue')
    plt.xlabel("Similarity Score")
    plt.ylabel("Frequency")
    plt.title("Distribution of Face Similarities")
    plt.show()
```

ANEXA 4 – FACENET

```python
import os
import numpy as np
import cv2
from keras_facenet import FaceNet
from mtcnn import MTCNN
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt


def load_facenet_model():
    return FaceNet()


def extract_face(image_path, detector, required_size=(160, 160)):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = detector.detect_faces(image)
    if results:
        x, y, width, height = results[0]['box']
        x, y = abs(x), abs(y)
        face = image[y:y + height, x:x + width]
        image = Image.fromarray(face)
        image = image.resize(required_size)
        return np.asarray(image)
    return None


def get_embedding(model, face_pixels):
    face_pixels = face_pixels.astype('float32')
    mean, std = face_pixels.mean(), face_pixels.std()
    face_pixels = (face_pixels - mean) / std
    embedding = model.embeddings(np.expand_dims(face_pixels, axis=0))
    return embedding[0]

```

```python
def analyze_faces(image_folder):
    detector = MTCNN()
    model = load_facenet_model()
    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        face = extract_face(image_path, detector)
        if face is not None:
            embedding = get_embedding(model, face)
            embeddings[image_name] = embedding

    return embeddings


def find_matching_faces(input_image_path, embeddings, model, threshold=0.5):
    detector = MTCNN()
    face = extract_face(input_image_path, detector)
    if face is None:
        print("No face detected in input image.")
        return []

    input_embedding = get_embedding(model, face)
    matching_faces = []

    for name, emb in embeddings.items():
        similarity = cosine_similarity([input_embedding], [emb])[0][0]
        if similarity > threshold:
            matching_faces.append((name, similarity))

    return sorted(matching_faces, key=lambda x: x[1], reverse=True)


if __name__ == "__main__":
    image_folder = "C:/Users/alex/PycharmProjects/PythonProject5/Images"
    test_image_path = "C:/Users/alex/PycharmProjects/PythonProject5/Test/test_image.jpeg"
```

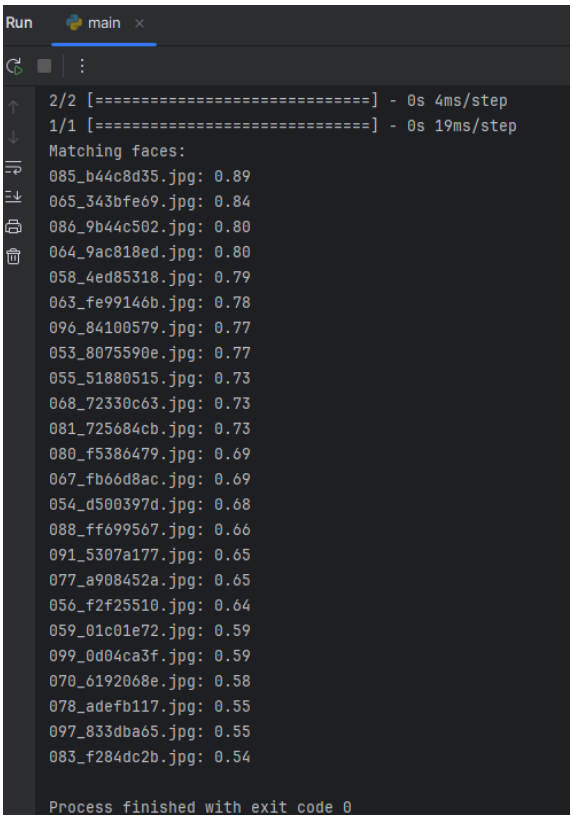
```python

    model = load_facenet_model()
    embeddings = analyze_faces(image_folder)
    matches = find_matching_faces(test_image_path, embeddings, model)

    if matches:
        print("Matching faces:")
        for name, score in matches:
            print(f"{name}: {score:.2f}")
    else:
        print("No matching faces found.")
```

```
C:\Users\alexa\PycharmProjects\PythonProject5\.venv\Scripts\python.exe C:\Users\alexa\PycharmProjects\PythonProject5\main.py
2025-02-09 12:28:25.316655: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2025-02-09 12:28:25.317020: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2025-02-09 12:28:28.081405: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library nvcuda.dll
2025-02-09 12:28:29.047917: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1733] Found device 0 with properties:
pciBusID: 0000:01:00.0 name: NVIDIA GeForce GTX 1650 Ti computeCapability: 7.5
coreClock: 1.485GHz coreCount: 16 deviceMemorySize: 4.00GiB deviceMemoryBandwidth: 178.8461B/s
2025-02-09 12:28:29.048786: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2025-02-09 12:28:29.049403: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cublas64_11.dll'; dlerror: cublas64_11.dll not found
2025-02-09 12:28:29.050003: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cublasLt64_11.dll'; dlerror: cublasLt64_11.dll not found
2025-02-09 12:28:29.050690: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cufft64_10.dll'; dlerror: cufft64_10.dll not found
2025-02-09 12:28:29.051432: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'curand64_10.dll'; dlerror: curand64_10.dll not found
2025-02-09 12:28:29.052048: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cusolver64_11.dll'; dlerror: cusolver64_11.dll not found
2025-02-09 12:28:29.052667: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cusparse64_11.dll'; dlerror: cusparse64_11.dll not found
2025-02-09 12:28:29.053274: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudnn64_8.dll'; dlerror: cudnn64_8.dll not found
2025-02-09 12:28:29.053445: W tensorflow/core/common_runtime/gpu/gpu_device.cc:1766] Cannot dlopen some GPU libraries. Please make sure the missing libraries mentioned above are instal
Skipping registering GPU devices...
2025-02-09 12:28:29.054012: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the follow
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2025-02-09 12:28:29.055197: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1258] Device interconnect StreamExecutor with strength 1 edge matrix:
2025-02-09 12:28:29.055358: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1264]
2025-02-09 12:28:33.359716: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the MLIR Optimization Passes are enabled (registered 2)
WARNING:tensorflow:5 out of the last 1566 calls to <function Model.make_predict_function.<locals>.predict_function at 0x000001BC8199C670> triggered tf.function retracing. Tracing is e
WARNING:tensorflow:6 out of the last 1567 calls to <function Model.make_predict_function.<locals>.predict_function at 0x000001BC81727160> triggered tf.function retracing. Tracing is e
```

ANEXA 5 – DEEPFACE

```python
import os
import numpy as np
import cv2
from deepface import DeepFace
from mtcnn import MTCNN
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt


def extract_face(image_path, detector, required_size=(160, 160)):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = detector.detect_faces(image)
    if results:
        x, y, width, height = results[0]['box']
        x, y = abs(x), abs(y)
        face = image[y:y + height, x:x + width]
        image = Image.fromarray(face)
        image = image.resize(required_size)
        return np.asarray(image)
    return None


def get_embedding(image_path):
    embedding = DeepFace.represent(img_path=image_path, model_name="Facenet", enforce_detection=False)
    return np.array(embedding[0]['embedding'])


def analyze_faces(image_folder):
    detector = MTCNN()
    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        face = extract_face(image_path, detector)
        if face is not None:
            embedding = get_embedding(image_path)
```

```python
            embedding = get_embedding(image_path)
            embeddings[image_name] = embedding

    return embeddings


def find_matching_faces(input_image_path, embeddings, threshold=0.5):
    input_embedding = get_embedding(input_image_path)
    matching_faces = []

    for name, emb in embeddings.items():
        similarity = cosine_similarity([input_embedding], [emb])[0][0]
        if similarity > threshold:
            matching_faces.append((name, similarity))

    return sorted(matching_faces, key=lambda x: x[1], reverse=True)


if __name__ == "__main__":
    image_folder = "C:/Users/alex/PycharmProjects/PythonProjectó/Images"
    test_image_path = "C:/Users/alex/PycharmProjects/PythonProjectó/Test/test_image.jpeg"

    embeddings = analyze_faces(image_folder)
    matches = find_matching_faces(test_image_path, embeddings)

    if matches:
        print("Matching faces:")
        for name, score in matches:
            print(f"{name}: {score:.2f}")
    else:
        print("No matching faces found.")

```

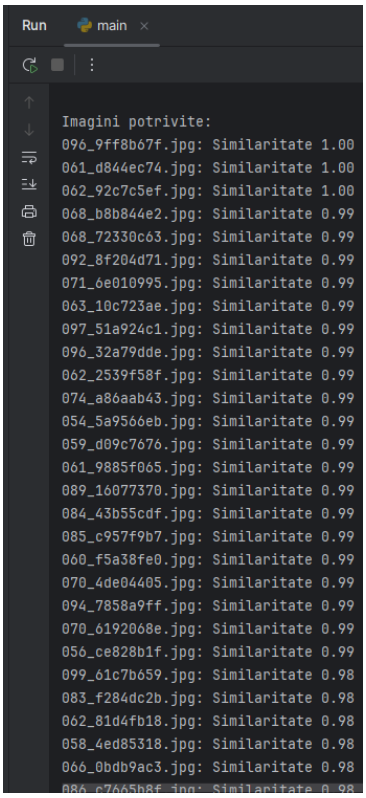

Run
main
2/2 [==============================] - 0s 4ms/step
1/1 [==============================] - 0s 19ms/step
Matching faces:
085_b44c8d35.jpg: 0.89
065_343bfe69.jpg: 0.84
086_9b44c502.jpg: 0.80
064_9ac818ed.jpg: 0.80
058_4ed85318.jpg: 0.79
063_fe99146b.jpg: 0.78
096_84100579.jpg: 0.77
053_8075590e.jpg: 0.77
055_51880515.jpg: 0.73
068_72330c63.jpg: 0.73
081_725684cb.jpg: 0.73
080_f5386479.jpg: 0.69
067_fb66d8ac.jpg: 0.69
054_d500397d.jpg: 0.68
088_ff699567.jpg: 0.66
091_5307a177.jpg: 0.65
077_a908452a.jpg: 0.65
056_f2f25510.jpg: 0.64
059_01c01e72.jpg: 0.59
099_0d04ca3f.jpg: 0.59
070_6192068e.jpg: 0.58
078_adefb117.jpg: 0.55
097_833dba65.jpg: 0.55
083_f284dc2b.jpg: 0.54
Process finished with exit code 0

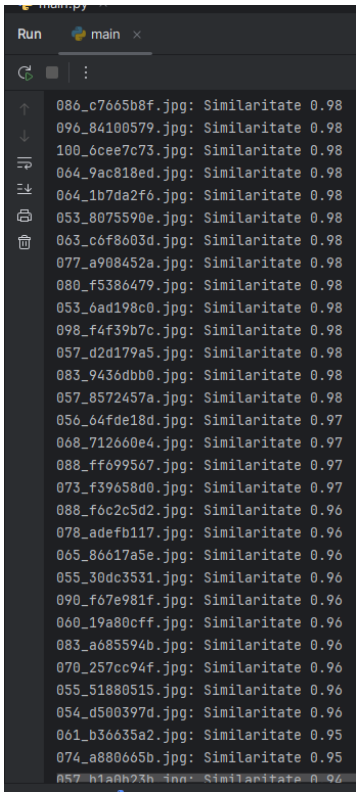
ANEXA 6 – MEDIAPIPE

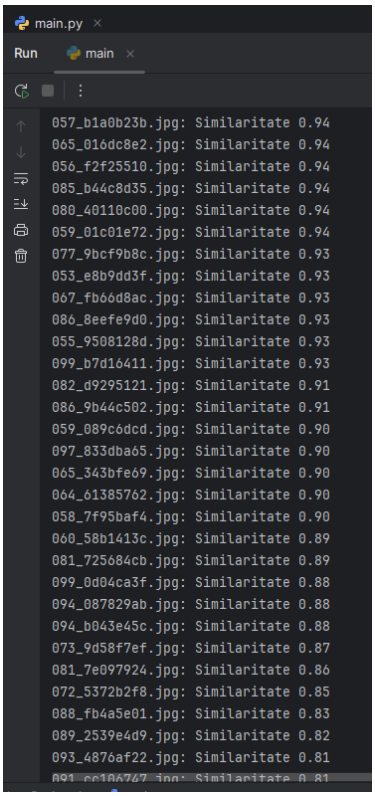
```python
import os
import numpy as np
import cv2
import mediapipe as mp
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity

# Inițializare MediaPipe Face Detection
mp_face_detection = mp.solutions.face_detection
mp_face_model = mp_face_detection.FaceDetection(model_selection=1, min_detection_confidence=0.5)


def extract_face(image_path, required_size=(160, 160)):
    """Extrage fața din imagine și o redimensionează."""
    image = cv2.imread(image_path)
    if image is None:
        return None  # Imaginea nu a fost încărcată corect

    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = mp_face_model.process(image)

    if results.detections:
        for detection in results.detections:
            bbox = detection.location_data.relative_bounding_box
            h, w, _ = image.shape
            x, y, width, height = (int(bbox.xmin * w), int(bbox.ymin * h), int(bbox.width * w), int(bbox.height * h))
            x, y = max(x, 0), max(y, 0)
            face = image[y:y + height, x:x + width]

            if face.size == 0:
                return None  # Evităm erori în cazul unei decupări incorecte

            image_resized = Image.fromarray(face).resize(required_size)
            return np.asarray(image_resized, dtype=np.float32) / 255.0  # Normalizare între 0 și 1

    return None
```

```python
def get_embedding(image_path):
    """Obține vectorul de trăsături al feței folosind MediaPipe."""
    image = cv2.imread(image_path)
    if image is None:
        return None

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = mp_face_model.process(image_rgb)

    if results.detections:
        for detection in results.detections:
            bbox = detection.location_data.relative_bounding_box
            return np.array([bbox.xmin, bbox.ymin, bbox.width, bbox.height], dtype=np.float32)  # Vector numeric fix
    return None


def analyze_faces(image_folder):
    """Analizează toate fețele dintr-un folder și returnează embeddings."""
    if not os.path.exists(image_folder):
        print(f"Folderul specificat nu există: {image_folder}")
        return {}

    embeddings = {}

    for image_name in os.listdir(image_folder):
        image_path = os.path.join(image_folder, image_name)
        embedding = get_embedding(image_path)

        if embedding is not None:
            embeddings[image_name] = embedding

    return embeddings


def match_face(target_image, database_embeddings, threshold=0.5):
    """Compară o imagine de referință cu baza de date și returnează cele mai bune potriviri."""
    target_embedding = get_embedding(target_image)
```

```python
    if target_embedding is None:
        print("Nu s-a putut extrage fața din imaginea de referință.")
        return []

    similarities = {}

    for name, emb in database_embeddings.items():
        similarity = cosine_similarity([target_embedding], [emb])[0][0]
        if similarity >= threshold:  # Doar imaginile peste un anumit prag sunt afișate
            similarities[name] = similarity

    # Sortare descrescătoare după similaritate
    sorted_matches = sorted(similarities.items(), key=lambda x: x[1], reverse=True)

    print("\nImagini potrivite:")
    for name, sim in sorted_matches:
        print(f"{name}: Similaritate {sim:.2f}")

    return sorted_matches


if __name__ == "__main__":
    database_folder = "C:/Users/alex/PycharmProjects/PythonProject5/Images"
    target_image_path = "C:/Users/alex/PycharmProjects/PythonProject5/Test/test_image.jpeg"

    print("Analizăm baza de date...")
    database_embeddings = analyze_faces(database_folder)

    if database_embeddings:
        print(f"Comparăm {target_image_path} cu baza de date...")
        matches = match_face(target_image_path, database_embeddings)

        if not matches:
            print("Nu s-au găsit potriviri peste pragul setat.")
    else:
        print("Nu s-au găsit fețe valide în baza de date.")
```







```
091_cc106747.jpg: Similaritate 0.81
063_fe99146b.jpg: Similaritate 0.78
093_4027b1e2.jpg: Similaritate 0.77
078_d36c2be3.jpg: Similaritate 0.74
091_930fe784.jpg: Similaritate 0.73
058_1b5b5422.jpg: Similaritate 0.68
054_ca0da4ca.jpg: Similaritate 0.65
067_fb02357d.jpg: Similaritate 0.62
091_5307a177.jpg: Similaritate 0.62

Process finished with exit code 0
```