



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Vecsei Gábor

ARCFELISMERŐ RENDSZER KIALAKÍTÁSA

Iskolai, üzleti környezetben

KONZULENS

Kertész Zsolt

BUDAPEST, 2016

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
2 Felhasználási lehetőségek	8
2.1 Gyakorlati felhasználások	8
2.1.1 Iskolai jelenlét generálás.....	8
2.1.2 Üzletek látogatottsága.....	9
2.1.3 Online párkereső oldalak	10
2.2 Kutatási felhasználási lehetőségek.....	10
2.3 Összefoglalás	11
3 Elméleti alapok.....	12
3.1 Arcok detektálása.....	12
3.1.1 Viola-Jones alapú arc detekció	12
3.2 Arcok felismerése	17
3.2.1 Eigenfaces	18
4 Technológiák	22
4.1 Matlab Image Processing Toolbox	22
4.2 OpenCV	22
4.3 Eszközök.....	25
4.3.1 Kamera.....	25
4.3.2 Raspberry Pi.....	26
4.3.3 Laptopom (Windows OS).....	27
5 Módszerek kiválasztása	28
5.1 Módszerek összehasonlítása	28
5.1.1 Arcdetektálás	28
5.1.2 Arcfelismerés	32
5.1.3 Összefoglalás	35
6 Megvalósítás	36
6.1 Python	36
6.2 A rendszer terve	37
6.3 Project felépítése.....	37

6.3.1 Mappák	38
6.3.2 Scriptek	38
6.3.3 Konfigurációs fájl	39
6.4 Arcfelismerő tanítása	39
6.4.1 Arcok kivágása a képről	40
6.4.2 Arcok előkészítése	40
6.4.3 Felismerő betanítása	41
6.5 Arcfelismerő alkalmazása	42
6.6 Jelenlét generálása	43
7 Tesztelés	45
7.1 Arcdetektálás tesztelése	46
7.1.1 Pontosság	46
7.1.2 Futási idő.....	47
7.2 Arcfelismerés tesztelése.....	49
7.2.1 Pontosság	50
7.2.2 Futási idő.....	53
Értékelés	54
Köszönetnyilvánítás	55
Irodalomjegyzék.....	56
Függelék.....	57
F.1. Paraméterezés	57
F.1.1. Arcdetektálás	57
F.1.2. Arcfelismerés.....	57
F.2. Környezet.....	57
F.3. Mellékletek	58
F.3.1. Github	58
F.3.2. CD-ROM-on.....	59
F.4. Programból képek.....	59

HALLGATÓI NYILATKOZAT

Alulírott **Vecsei Gábor**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzé tegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 12. 08.

.....
Vecsei Gábor

Összefoglaló

A szakdolgozat megírása és a hozzá kötődő software elkészítése nagyon sokat segített a téma mélyebb megértésében. A project végeztével egy olyan programot sikerült létrehoznom, ami valós időben robusztusan detektálni tud arcokat és felismeri őket, ezzel segítve különböző feladatok automatizálását.

Megnéztem és kipróbáltam több különböző arcdetektálásra használt módszert és ezek közül a *Viola-Jones* objektum detektoros megoldást találtam a legjobbnak. Ugyan így teszteltem több fajta arcfelismerő algoritmust, majd a tesztek végeztével az *Eigenfaces*-re esett a választásom mivel kevés tanító adattal is jó eredményt tudtam elérni és a felismerés pontossága 90% feletti volt.

A két külön modult azaz a detektálást és a felismerést egy olyan rendszerben kötöttem össze, ami egy jelenléti ív generálásában segít ezzel időt nyerve sok helyzetben. A generált Excel táblában látszik, hogy kinek mikor ismerte fel az arcát így mikor bemegyünk egy ajtón akkor tudni fogjuk mikor lépett be az illető a terembe.

Miután elkészítettem a rendszert, le is kellett tesztelni annak érdekében, hogy biztosan azt mondhassam, hogy ez egy használatba helyezhető software lett. Így külön-külön a komplexebb algoritmusokat futtattam és megnéztem az adatokon, hogy hogy teljesít pontosság és futási idő tekintetében és nagyon jó eredményeket kaptam amik arra adnak következtetést, hogy valós körülmények közt is használható az elkészített program.

Abstract

As I wrote my thesis and created the software for it I had a chance to look deeper in the theory of face detection and recognition. After I completed the project and accomplished my goals, I had an application which can run in real-time and detect and recognize faces robustly and that way it can help to automate tasks.

I looked for different approaches in face detection and I tested them but at the end I chose the *Viola-Jones* object detector algorithm because that was the best in all of them. The same way, I started to search for an accurate face recognition method and I ran tests to see the results. After the tests, I selected the so called *Eigenfaces* algorithm because with that I reached up to 90% accuracy.

With this two, separate module (namely detection and recognition) I created a system which can generate attendance sheets. With that we can save a lot of time in different cases. In the generated Excel sheet, we can find who stepped in the door at a specific time.

After the system was created I had to make tests on it in order to determine if I can use it in real life situations. So, I made test scripts for the more complex algorithms and tested those for accuracy and running time. After I ran all the tests I can say that the system can be used in real life situations.

1 Bevezetés

Manapság sokat hallhatunk, olvashatunk a különböző biometrikus azonosítási eljárásokról és ezek közül talán az egyik leggyakrabban hallott az arc detektálás és felismerés. Az arcok detektálása és felismerése egy olyan összetett feladat, amihez nem csak jól kell kódot írni, hanem meg kell érteni alacsonyabb szinten az egyes képfeldolgozási algoritmusokat, technikákat. A rendszert először meg kell tervezni, összeállítani, majd le lehet implementálni. Ennek a rendszernek/programnak a megalkotása egy olyan mérnöki munka, ami tökéletessé teszi ezt a témát, hogy a szakdolgozatomban megvalósítsam.

A mai világban majdnem minden sarkon lapul egy kamera, így tudunk nélkül több felvétel is készülhet rólunk egy séta alkalmával és ugyan az a helyzet az iskolákkal, cégekkel, és különböző intézményekkel. Az arcunk olyan információt tartalmaz, amit fel lehet használni számos területen. Mi emberek, is általában az arca alapján ismerünk fel valakit így, ha gépekbe átültetnénk ezt, akkor bizonyos tevékenységeket nagyobb hatásfokkal és gyorsabban tudnánk csinálni. Ami az egyik másik fontos dolog, hogy ez nem egy drága technika, amihez kell valami különleges hardware. A felismerő eljárásokhoz elég egy kamera és egy server gép, ami elvégzi a bejövő információ feldolgozását és továbbítja azt a kellő helyre.

2 Felhasználási lehetőségek

Mint említettem arcunk olyan információt hordoz, ami egyedi, viszont, ha belegondolunk akkor rájövünk, hogy nem csak mi tudunk róla és így könnyű lemásolni. Vegyünk az, hogy ha készül egy kép rólunk akkor ott van rajta az arcunk, amit onnantól mindenki meg tud mutatni egy kamerának és a software fel fogja ismerni az arcot. Ezért az arcunk inkább egy felhasználónév mintsem jelszó. De azzal, hogy felhasználónévként használjuk felgyorsíthatunk dolgokat vagy pedig még biztonságosabbá tehetünk folyamatokat. Esetlegesen még összefüggéseket is kereshetünk az alapján, hogy az egyik ember mennyire hasonlít a másikra.

2.1 Gyakorlati felhasználások

2.1.1 Iskolai jelenlét generálás

Az általános, középiskolákban nagyon sok idő elmegy azzal, amíg a tanár le ellenőrzi, hogy ki ment be órára és felolvassa a neveket, kitölti a naplót. A hatékonyságot lehetne növelni a képfeldolgozás és az arcfelismerés segítségével. Ha felismernénk a tanulót már akkor, amikor belépett az iskolába vagy terembe akkor már tudunk bizonyos információkat (mint például a detekció ideje, helye) és ezekkel az információkkal legenerálhatunk automatikusan dokumentumokat, mint például a jelenléti ívet. Így nem menne el hosszas idő azzal, hogy a tanár felolvassa a neveket és jobban ki lehetne használni az időt tanulásra.

Az egyetemeken pedig elég lenne előadónként minden bejáratnál egy kamera, ami felismeré a diákokat és így jelenléteket és statisztikákat tudna generálni.

Amikor a tanuló belép az ajtón akkor bele néz egy kamerába pár pillanatra és azután a rendszer már tudni fogja, hogy ki ő és mikor lépett be az ajtón, így kész is a jelenlét, amit a tanár vagy előadó megkap a nap végén, vagy a hónap végén. Az összegyűjtött adatokkal amit a szerveren tárolunk statisztikákat lehet generálni és így

javítani esetleg 1-1 óra időpontján mivel lehet azért nem látogatott az óra mert túl korán vagy túl későn van.

2.1.2 Üzletek látogatottsága

Az üzleteknél az egyik nagyon fontos dolog, hogy mennyi az új és a visszatérő vásárlók aránya, mivel abból látszik az, hogy mennyire marad valaki hű ahhoz az üzlethez miután elmegy onnan. És ha valaki már nem csak egy új nézelődő, hanem visszatérő vásárló akkor valószínűleg nagyobb összeget fog otthagyni mivel nem csak nézelődni jött. Ezt is sok minden befolyásolhatja, amit le tudunk szűkíteni azzal, hogy egy kamerával folyton figyeljük az érkező vendégeket, akkor aki egyszer betért az üzletbe azt hozzáadjuk a már felismerhető emberekhez és ha még egyszer visszajön akkor már tudni fogja a rendszer, hogy járt itt. Így könnyedén meg tudjuk mondani az arányát az új és a visszatérő vásárlóknak, és következtetéseket vonhatunk le, hogy milyen összefüggésben van ez azzal, hogy éppen ki dolgozik, milyen zene szól, milyen nap van, és még sorolhatnám.

Egy olcsó és egy szinte a vevő számára észrevehetetlen rendszerrel meg is oldottunk egy problémát mivel kamerát így is használnak az üzletek és csak egy olyan software kell, ami képes a fentebb soroltak megvalósításában.

Több kamerát alkalmazva még trackelhetjük is a vásárlókat így további adatot gyűjtve, amit felhasználhatunk különböző statisztikák generálására, az üzlet fejlesztésére, marketing célokra és ami még nagyon fontos, a biztonság növelésére. Külön megtaníthatjuk azokat az arcokat, akik esetleg már loptak a boltból vagy összetűzésbe kerültek más vásárlókkal és ha ők jönnek be az üzletbe akkor a biztonsági őrköt rögtön értesítené a rendszer.



2.1. ábra. Üzletben vásárlók detekciója és felismerése

Forrás: <http://www.consumerreports.org/privacy/facial-recognition-who-is-tracking-you-in-public1>

2.1.3 Online párkereső oldalak

Ezt a példát csak azért hozom fel, hogy lehessen látni, hogy milyen széles körben fel lehet használni a módszert. Ez nem egy mindennapi eset de szerintem hasznos lehetne egy két ilyen oldal számára.

Aki pár keres az interneten, ott először a kinézet a legfontosabb a legtöbb ember számára (sajnálatos módon) és lehet, hogy van, aki az alapján szeretne keresni, hogy az illető hasonlítson valakire. Ilyenkor feltölthetne egy képet és a program visszaadna egy listát azokról a regisztrált felhasználókról, akiknek az arca a legjobban hasonlít a feltöltött képen lévőre.

2.2 Kutatási felhasználási lehetőségek

Kutatási területeken is igen hasznos lehet, az, hogy ha meg tudjuk mondani egy arcról, hogy az kicsoda. Vagyis, ha már meg tudjuk mondai, hogy hasonló valakihez akkor tudunk mondani egy bizonyosságot is, hogy mennyire hasonló az illető. Így párba

állítva embereket meg lehet állapítani, hogy az egyik arc mennyire hasonlít a másikhoz. Ezt különböző demográfiával foglalkozó cégek hasznosítani tudják vagy pedig kutatások történhetnek azzal kapcsolatban, hogy mi az optimális hasonlóság egy kapcsolatban (már ha létezik ilyen).

2.3 Összefoglalás

Látszik, hogy nagyon sokféleképpen fel lehet használni a kapott eredményeket, amiket én megpróbáltam kicsit összefoglalni de még így is csak a felszínét érintettem a használati lehetőségeknek. Mindehhez egy olyan robosztus rendszer elkészítése a szükséges, ami arcokat nagy bizonyossággal detektálni tud és fel is tud ismerni esetlegesen csoportokba rendezni azaz klaszterezni.

3 Elméleti alapok

3.1 Arcok detektálása

Az arc detekció nem egy öreg, régen kitalált terület. Így kis idő alatt elég sok algoritmus és technika került kifejlesztésre annak érdekében, hogy az arcokat fel tudják ismerni a képen. Itt még nem arról van szó, hogy egy arcról megmondjuk, hogy ki az, hanem az arc detekció csak az a folyamat, amivel egy képen meg tudjuk mondani, hogy van-e arc és annak a helyzetét is be tudjuk azonosítani.

Az egyik még máig versenyképes algoritmus vagyis inkább framework az úgynevezett *Viola-Jones object detection* [1]. Ez egy olyan algoritmus még 2001-ben fejlesztettek ki és igaz, hogy object detection-ről van szó benne de leginkább arcok detektálására használják és a megalkotásának ez is volt elsődleges motivációja. Ezzel a módszerrel valós időben lehetőségünk van arcok detektálására képeken, és mivel ilyen gyors így akár követni is tudunk arcokat videókon.

A másik lehetőség, amivel mostanában arcokat detektálnak azt mély neurális hálókkal (deep learning) valósítják meg és az a tapasztalat, hogy pontosabb, mint az előbb említett algoritmus de kicsivel lassabb is. Ekkor konvolúciós hálókat tanítanak, amik megtanulják az arc egyes jellemzőit. Ezzel a módszerrel könnyedén lehet kategorizálni nőket és férfiakat is.

Összefoglalva, amit a Viola-Jones-al tudunk detektálni azt tudjuk deep learninges megközelítéssel is viszont előbbit használva lesz sok false pozitív eredményünk, ami azt jelenti, hogy detektált arcot de mégsem arc van a képen. Viszont ennek a pontosságát lehet növelni úgy, hogy nem veszít sokat a futási idő gyorsaságából. Nagyon sok gyártó kamerákban és mobilokban alkalmazza arcok, és azon belül is mosoly felismerésére a metódust.

Elterjedtsége és robusztussága miatt választottam és is ezt a módszert.

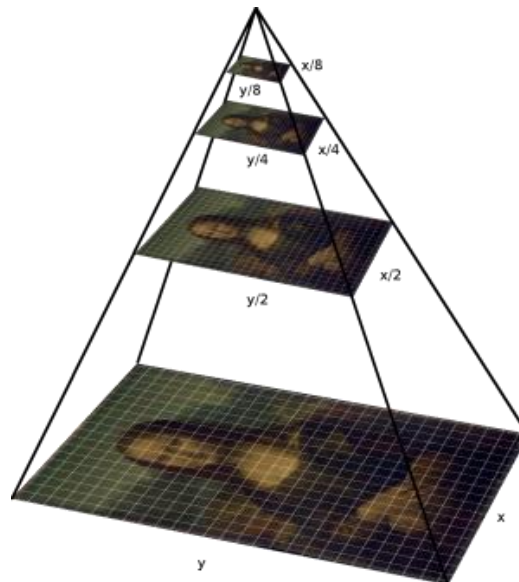
3.1.1 Viola-Jones alapú arc detekció

Paul Viola és Michael J. Jones 2001-ben robosztus, valós idejű objektum detektort fejlesztettek, amit széles körben használnak arcok detektálására. A technika robosztussága [2] abban rejlik, hogy a kép minőségére érzéketlen. Így ugyan azt az eredményt kapjuk egy nagy felbontású képnél, mint egy kisebb felbontásúnál.

Szürkeárnyaltos képeken keres jellemzőket az algoritmus és 640x480-as felbontású képeknél 30 kép / másodperces arányt tudunk elérni, amiből látszik, hogy szépen fog működni akkor is, ha videón akarunk detektálni.

Más algoritmusok sok mindent felhasználnak, azért, hogy elérjék a kellő pontosságot, mint például két kép különbséget nézni videóknál, így egy pontosabb becslést kaphatunk arról, hogy hogy lehet az arc vagy pedig a bőrszín szerinte szegmentálni, de ezeket a Viola-Jones-al is egybe lehet kötni és így növelni tudjuk a pontosságát a detektornak.

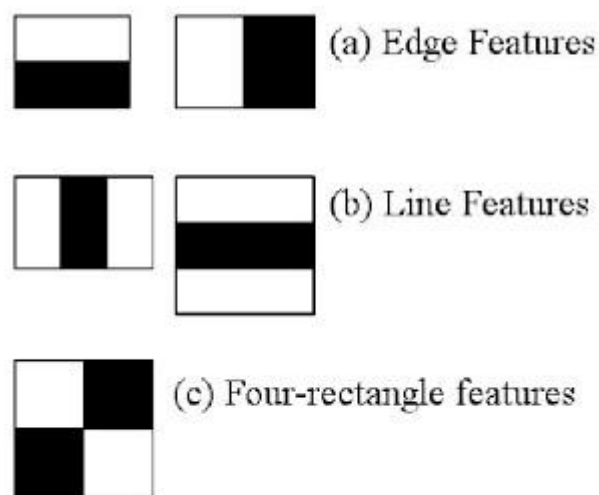
Az algoritmus azért lehet ilyen gyors mert nem az egész képpel foglalkozik egyszerre, hanem csak képrégiókkal és azoknak a jellemzőivel. Jellemzők egy halmazát alkalmazza (Haar féle jellemzők) és azokat nézi meg a szürkeárnyaltos kép régióin több pozícióban és skálában. A skálázás itt fontos, hogy ha valami nagyon nagy vagy nagyon kicsi akkor azt is fel tudja ismerni így Gauss piramist épít, amit az alábbi képen lehet látni.



3.1. ábra. Gauss Piramis építése

Forrás: <http://ipimage.sourceforge.net/documentation/images/>

Olyan osztályozót hozunk létre, ami a fontos jellemzők kis halmazának kiválasztásával AdaBoost-on alapuló tanuló algoritmussal létrehozza azt. Az AdaBoost algoritmus lényege, hogy sok alacsony szintű osztályozót használnak egy kapcsolatot létrehozva egy osztályozó és egy jellemző között és pozitív, illetve negatív példákon megnézzük, hogy mely jellemzők a leghatásosabbak az osztályozás ellátására. A kép egy kis részképében a Haar jellemzők száma sokkal nagyobb, mint a pixelek száma, ami maga után vonja azt, hogy lassú lesz az algoritmus. De pontosan emiatt, hogy gyorsan tudjuk osztályozni, a tanuló folyamatnak csak egy kritikus halmazra kell figyelnie a jellemzőkből és nagy részét ki kell zárnia.

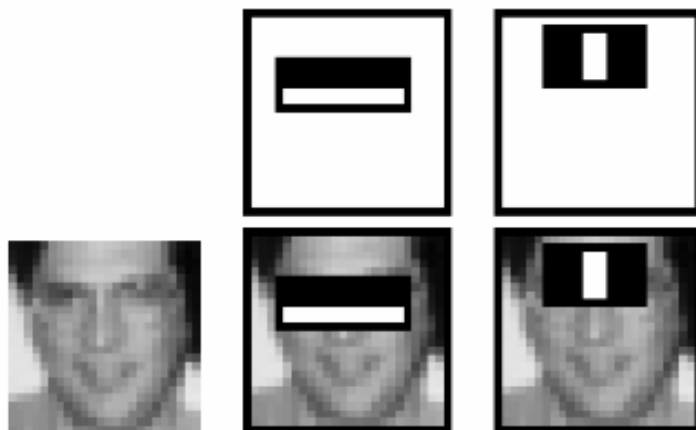


3.2. ábra. Haar Jellemzők

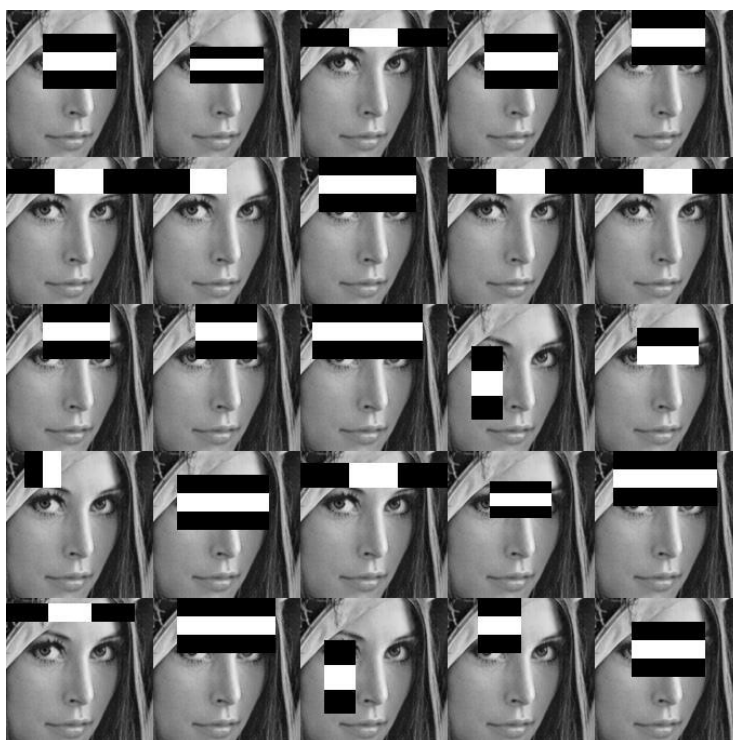
Egy másik fontos tulajdonság az az, hogy több osztályozót kaszkád struktúrába kell rendezni (sorba rendezni). Ez azért jó és gyorsítja az algoritmust, mert csak azokra a területekre kell koncentrálnunk, ahol a fontos jellemzők vannak. Mikor az egyik osztályozó azt detektálja egy régióra, hogy biztosan nincs benne arc, akkor a többi le se fut hanem tovább megyünk a következő régióra és ez így folytatódik amíg az összes jellemzővel le nem ellenőriztük. Ez nagyban hozzájárul a gyorsasághoz mivel, ha már egy jellemző szerint nincs arc a képen akkor nincs is értelme a többivel végig nézni. Így csak a kellően fontos régiókra kell összpontosítani az algoritmusnak.

Ezeket a jellemzőket és az illesztésüket a kép régióira úgy lehet elképzelni, hogy például vesszük azt a régiót, amiben a szemünk és az orrunk van benne. Ekkor szinte biztos, hogy az orrunk fényesebb lesz, mint a két szemünk mivel azok árnyékoltabban

vannak, mint az orrunk amire pont rásüt a fény. Így az alább látható jellemző illeszkedni fog a szemeink és az orrunkra. Ezt láthatjuk az alábbi kép leutolsó képkockájánál.



3.3. ábra. Haar jellemző illeszkedése az orr-ra és a szemekre

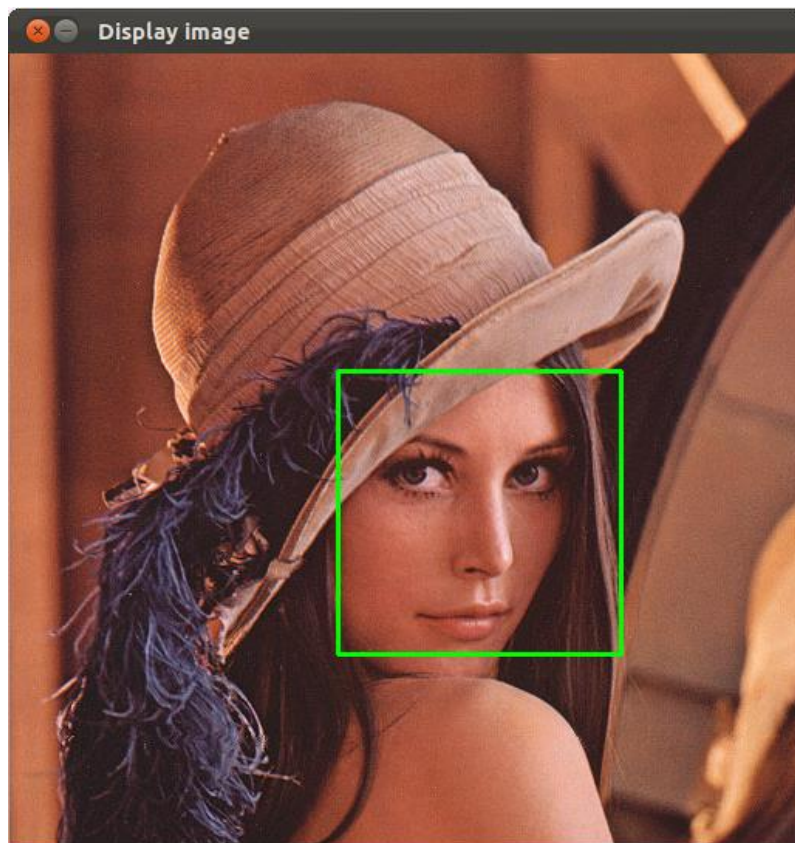


3.4. ábra. Haar jellemzők illesztése képre

Egy ilyen osztályozót már betanítottak a közkezdelt és sokak által használt Intel által fejlesztett (aztán nyílt forráskódú) OpenCV library-ben [3]. Én is ezt a Haar-

Cascade-ot használom a munkámban mivel ezt direkt arra készítették, hogy a lehető legnagyobb pontossággal ismerjen fel arcokat képen és határozza meg azok helyzetét. Az ebben megvalósított detektornak szinte bármit meg lehet adni de mivel valaki már betanított egy arc osztályozót, így nekem nem kell azzal bajlódnom és a detektorba csak betöltök egy már előkészített .xml fájlt.

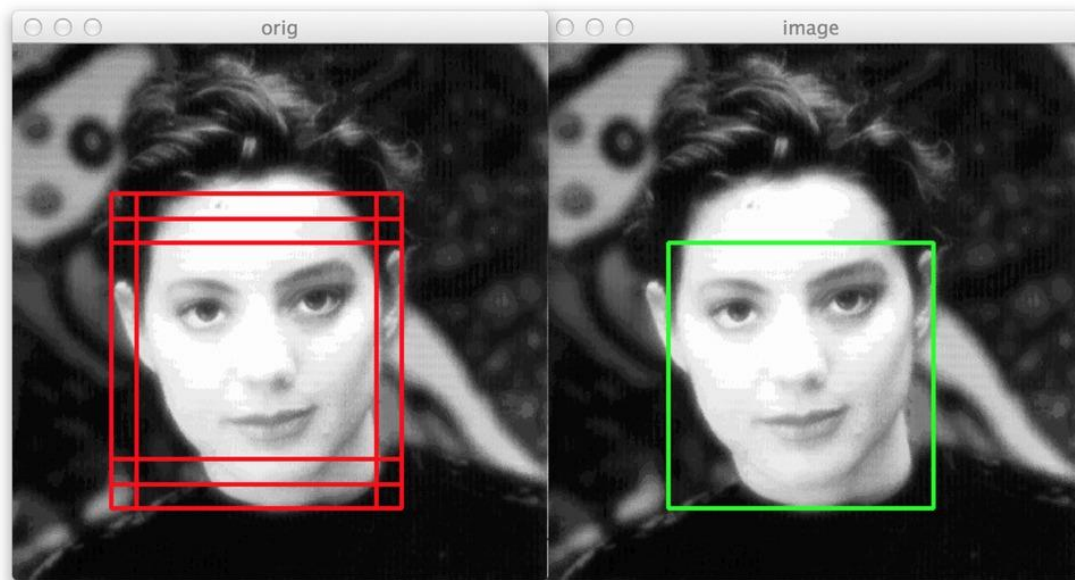
Ezt az arc felismerőt viszont olyan arcokon tanították (nagy részben) amikor az arc teljesen a kamerába nézett, esetleg kicsit elfordult, ezért is hívják *frontal face cascade*-nak. Ennek az a hátránya, hogy csak ugyan ilyen arcokat ismer fel a képeket. Tehát nem rotáció invariáns. Ha eldöntöm a fejemet nagy mértékbe balra vagy jobbra akkor már nem tudja felismerni az egyébként könnyen kivehető arcot a képen. De ez nem is probléma, mivel az ép programomban is úgy működik a felismerés, hogy a felhasználónak pár másodperc erejéig bele kell néznie a kamerába.



3.5. ábra. Detekció eredménye

Forrás: <https://pdincau.wordpress.com/2012/11/15/face-detection-using-python-and-opencv/>

Mivel csúszó ablakkal vizsgáljuk meg a képet (a képet kisebb képekre osztjuk) így általában az történik, hogy többször sikerül detektálni egy arcot. Viszont ez nekünk nem jó mert az algoritmus lehet azt adná vissza, hogy 5 arc van a képen, pedig az az arc ugyan az, csak többször lett felismerve. Ennek kiküszöbölésére használják a Non-Maximum Supression-t (NMS algoritmus) ami a többszörös detekcióból kiválasztja az optimálisat.



3.6. ábra. NMS eredménye többszörös detekcióra

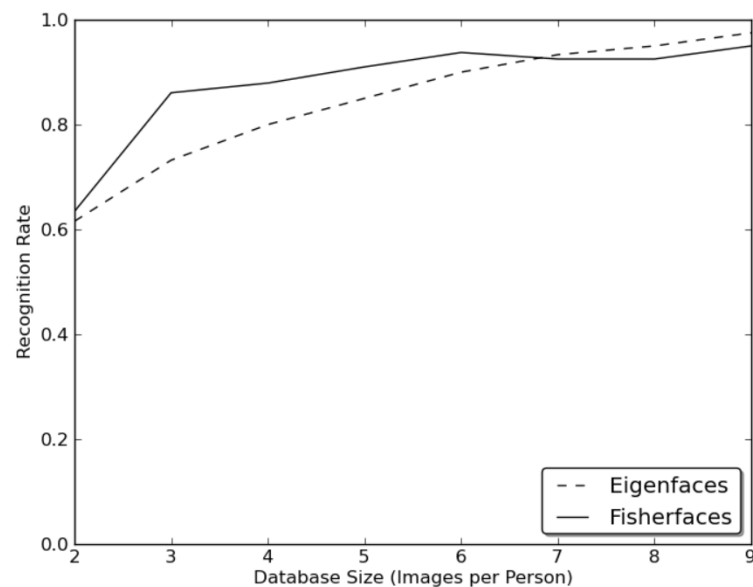
Forrás: <http://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python>

Ez az algoritmus a dobozok (bounding box-ok) átlapolódó területét nézi meg és az alapján kiválasztja azt amelyik optimális a detekcióra.

3.2 Arcok felismerése

Az arcok felismerése, már más feladat, mint a detektálást, bár erősen egymáshoz kapcsolódnak egy olyan rendszerben, mint amit én is készítettem. Először detektálni kell az arcot a képen aztán pedig mikor előkészítettük a képet, hogy csak egy arc legyen

rajta akkor fel lehet ismerni, és megmondani, hogy melyik már előzőleg előkészített arc hasonlít rá, és hogy mennyire. Ennek is több technikája van. Az OpenCV-ben 3 fajta található meg: Eigenfaces, Fisherfaces, Local Binary patterns Histograms (LBP) [4]. Ezek közül én az elsőt használom mivel az szolgálja a legjobb találati eredményt kevés bemenő adatra, amivel a rendszert betanítom, hogy tudja, hogy ki kicsoda. Az alábbi ábrán is jól látszik, hogy kevés tanító adattal jobban járok, ha az Eigenfaces-t választom.



3.7. ábra. *Eigenfaces és Fisherfaces arcfelismerők pontosságának összevetése*

Ezeknél a módszereknél kell egy adatbázissal rendelkezni, amiben megtalálhatók a képek az arcokról és kellene még *label*-nek nevezett adatok, amik azt mondják meg, hogy ki van a képen. A label-ek, avagy címkék számok szoktak lenni, amiket aztán string-ként át tudunk alakítani és így visszakaphatjuk a nevét valakinek. Viszont nem muszáj már meglévő dataset-et alkalmazni, mivel akár valós időben adhatunk hozzá új adatokat és csak újra tanítjuk majd az osztályozónkat.

3.2.1 Eigenfaces

Az Eigenfaces [5] módszert sajátarc módszernek szokták magyarul nevezni. Ennek az arcfelismerő megoldásnak a lényege, hogy az arcokat vektorokként ábrázolja.

A bekerülő képekből készít egy „átlagarcot” és ezután az átlagarcról való eltérést nézzük. Az arcok lényeges részei a képek kovariancia mátrixának saját-vektoraiként határozhatók meg. A saját-vektorok képviselik az egyes arcok különbözőségét és bizonyos jellemzőit de ezek közül is csak a legnagyobb saját-értékűeket alkalmazzák mivel abban vannak benne a fő jellemzői az arcnak. Új kép felvételénél kivonják azt az átlagarcból és így kapunk egy különbség képet, mint vektort. Ezután a saját-vektorok szorzatát képzik, és így megkapjuk az arc helyét a saját-vektor térben.

[6] $N \times N$ (azért négyzetes mivel a detektorunk négyzetekben találja meg az arcokat) a kép pixeleinek a száma melyet egy oszlop-vektorral reprezentálunk:

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ \vdots \\ \vdots \\ a_{N^2} \end{pmatrix} \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ \vdots \\ \vdots \\ b_{N^2} \end{pmatrix} \quad \vec{c} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ \vdots \\ \vdots \\ c_{N^2} \end{pmatrix} \quad \dots \quad \vec{h} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ \vdots \\ \vdots \\ \vdots \\ h_{N^2} \end{pmatrix}$$

Ezután kiszámoljuk a képek átlagát ezt fogom m -el jelölni:

Error! Objects cannot be created from editing field codes.

(M = beolvasott képek száma)



3.8. ábra. Bal oldalon látható az átlagarc, a többi 3 arc pedig Eigenface

Forrás: <http://www.0xc0de.net/2011/07/eigenfaces-with-opencv-2.html>

Miután megkaptuk az átlagarcot, ki kell vonnunk az egyes oszlopvektorokat ebből az m vektorból.

$$\vec{a}_m = \begin{pmatrix} \vec{a}_1 - \vec{m}_1 \\ \vec{a}_2 - \vec{m}_2 \\ \vec{a}_3 - \vec{m}_3 \\ \vdots \\ \vdots \\ \vec{a}_{N^2} - \vec{m}_{N^2} \end{pmatrix} \quad \dots \quad \vec{h}_m = \begin{pmatrix} \vec{h}_1 - \vec{m}_1 \\ \vec{h}_2 - \vec{m}_2 \\ \vec{h}_3 - \vec{m}_3 \\ \vdots \\ \vdots \\ \vec{h}_{N^2} - \vec{m}_{N^2} \end{pmatrix}$$

Létre kell hoznunk egy A mátrixot, ami a pixelek számával azonos sorból áll és M oszlopból (beolvasott képek száma).

$$A = \begin{bmatrix} \vec{a}_m & \vec{b}_m & \vec{c}_m & \vec{d}_m & \vec{e}_m & \vec{f}_m & \vec{g}_m & \vec{h}_m \end{bmatrix}$$

Ezután kiszámítjuk a kovariancia mátrixot úgy, hogy az A mátrix transzponáltját beszorozzuk önmagával, így egy $M \times M$ -es mátrixot kapunk. Mikor fordítva járunk el akkor egy $N^2 \times N^2$ mátrixot kapnánk, aminek túl nagy a számítás igénye.

$$L = A^T A$$

$$C = AA^T$$

L mátrix sajátvektorainak lineáris kombinációjával előállíthatók C sajátvektorai. Jelöljük L mátrix sajátvektorát V -vel. Ekkor $U=AV$, ami a C mátrix sajátvektorai. Ez a vektor fejezi ki az eltéréseket az arcok között. Úgy is nevezik, mint arctér.

Ezután, ha fel akarunk ismeri egy arcot a detektálás után akkor abból is oszlopvektort képzünk, az átlagarcot kivonjuk belőle, és rávetítjük az összes arcot és a keresett arcot is az arctérre. Ezután ki kell számítani az arcterek távolságát és megkapjuk, hogy az arcok „milyen messze vannak”. Itt egy távolságmértéket kapunk vissza és innen a legkisebbet vesszük annak, mint a beolvasott arc megfelelője. Természetesen ez még nem azt jelenti, hogy biztosan van egy találatunk, mivel, ha egy képet beadunk a programnak az mindig visszatér egy számmal mert mindig lesz egy

legkisebb távolság. Ezért nekünk meg kell határozni egy thresholdot ami felett nem számít felismerésnek, és ugyan úgy fogja kezelni, mint egy ismeretlen arcot.

4 Technológiák

A szakdolgozatom készítése során többféle hardwares és softwarés technológiával kellett megismerkednem. Ebben a fejezetben be szeretném mutatni azokat a könyvtárakat, amiket használtam és a miérteket megmagyarázni, hogy mi alapján döntöttem egyes könyvtárak mellett.

4.1 Matlab Image Processing Toolbox

Először a matlab-ot választottam kiindulás pontnak, mivel az egyetemen laboratóriumok során ezt használtuk legtöbbször. Még a fent említett toolbox-al is volt laborunk így tudtam, hogy nagyjából mit hol keressek. Nagyon jónak találtam ezt a toolboxot mert minden, ami kellett megtalálható volt benne, de nagy gond az volt vele, hogy nagyon lassan futottak le az egyes képfeldolgozási algoritmusok és így nem lehetett volna real-time arcfelismerőt készíteni vele, illetve ebben a toolboxban nem találhatók meg bizonyos modulok, amik benne vannak a lent említett OpenCV könyvtárban.

4.2 OpenCV

Az OpenCV egy open source azaz nyílt forráskódú gépi látással foglalkozó könyvtár. BSD license-el rendelkezik, ami azt jelenti, hogy szabad felhasználású. Sajnos vannak olyan metódusok benne, mint például a *SIFT* és *SURF*, amik más licenszeléssel rendelkeznek és azok már nem szabad felhasználásúak.

A könyvtár nagyban segíti a gyors prototípus tervezést, mivel szinte minden alap algoritmus implementálva van benne, amiből komplexebb rendszereket létre lehet hozni. Elsődleges célja a valós-idejű képfeldolgozási alkalmazások fejlesztése.

Az Intel kezdte el fejleszteni még 1999-ben [7] a könyvtárat és 2000-ben lett bemutatva a nagyközönségnek az *IEEE Conference on Computer Vision and Pattern Recognition* konferencián. Ekkor még nem volt hivatalos kiadás mert csak béta verziók léteztek de 2006-ban megtört a jég és bemutatták az 1.0-ás hivatalos változatot. Ebben a változatban még csak C nyelven volt interfész.

2009 második felében (októberben) megjelent a 2.0-ás változat és ezzel a verzióval nagy fejlődésen esett át a projekt. Ennél a verziónál jelent meg a C++ és a Python interfész (A C mellett) és így már 3 nyelvet támogatott. Mivel ez idő alatt fejlődött a hardware is így többmagos rendszerekben már jobb implementációval gyorsabb futást biztosított a felhasználónak.

2012 augusztusában létrejött egy nonprofit alapítvány, *OpenCV.org* mely átvette a fejlesztést. Több nem hivatalos támogatással több wrapper is elkészült, így C# és Ruby nyelven is elérhető lett többek között. Ekkor már széles körben alkalmazták, mivel a támogatott nyelvek listája egyre csak nőtt.

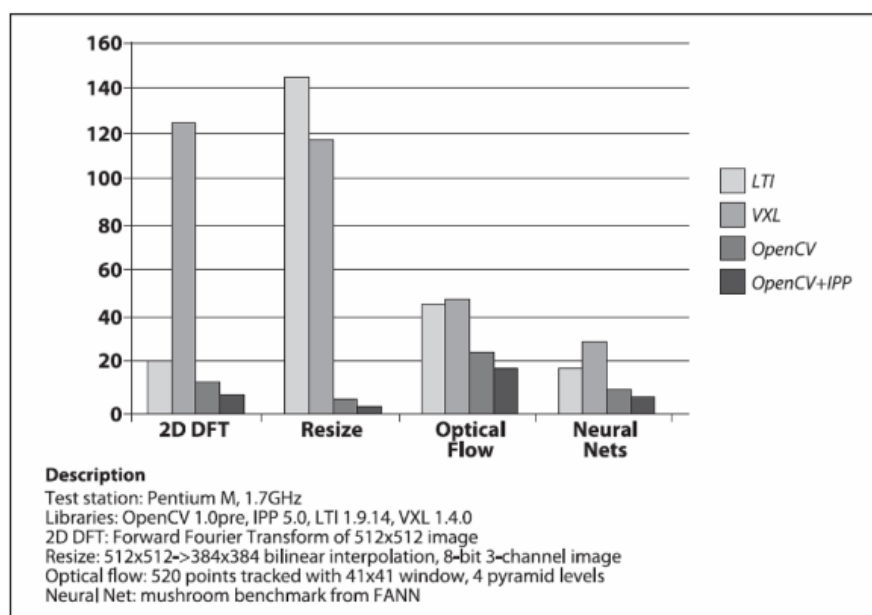
Mivel alapvetően C nyelvre épült ezért sokféle rendszeren lehet működtetni. A fejlesztés ezért cross platform módon, különböző platformokra párhuzamosan történik. A rendszerek közt megtalálható mind asztali és mind mobil operációs rendszerek. Támogatja a Window, Linux, OS X, FreeBSD, OpenBSD, Android, iOS, BlackBerry 10 rendszereket.

A felsorolt pozitívumok miatt használja így több mint 47.000 fő, és ez egyre csak növekszik. Munkám során ezzel a könyvtárral dolgozok és még sose csalódtam benne én sem. A lenti ábrán látszik az összehasonlítása más képfeldolgozással foglalkozó könyvtárakkal, és az is, hogy jobb minden területen a többiekkel szemben. Tehát ha nagyon gyorsan szeretnék valamit futtatni akkor biztosan az OpenCV-t választom. Mivel megtalálható benne minden így nem kell újra feltalálni a kereket és foglalkozhatok a lényeges részekkel és nem megy el az idő az alap algoritmusok implementálására.

Több könyvet is írtak használatáról (például [8]) így a teljesség igénye nélkül bemutatnám az alapvető jellemzőit a könyvtárnak:

- Megtalálhatóak benne az alap adatstruktúrák mint mátrixok, vektorok, és természetesen az ezekkel végezhető műveletek is
- Dinamikus adatstruktúrák, mint listák, sorok, halmazok, fák, gráfok

- Képeket könnyedén olvashatunk be és írhatunk ki fájlba. Itt csak arra kell figyelni, hogy az OpenCV a színes képeket „fordítva” olvassa be és tárolja a mátrixban mivel RGB (piros, zöld, kék) komponenseket felcseréli és BGR struktúrában tárolja, ami könyvtárak közötti együttműködést kicsit megnehezíti.
- Képek elő feldolgozásához használt él és sarok detektálás, szín konvertálás, morfológiai operátorok
- Hough transzformáció, kontúrok keresése és feldolgozása, mintaillesztés, momentumok
- Objektumfelismerés: Eigenfaces, Local Binary Pattern Histogram, Fisherfaces, Haar-Cascades, rejtett Markov modellek
- Gépi tanulással kapcsolatos algoritmusok, függvények is megtalálhatók benne: Support Vector Machine (SVM), Döntési fa, Bayes osztályozó
- General User Interface (GUI): képek megjelenítése, video megjelenítése, billentyűzet és egér kezelés, különböző primitív alakzatok rajzolása (egyenes, kör, poligon, szöveg)



4.1. ábra. *OpenCV összehasonlítása LTI, VXL Image Processing könyvtárakkal*

Forrás: [8]

Ezekén kívül még nagyon sok minden más megtalálható a könyvtárban de én a szakdolgozatom készítése során ezeket használtam legtöbbit.

4.3 Eszközök

Mivel képfeldolgozásról van szó, azon belül is webkamerán való arcfelismerésről így kellett egy webkamera ami kielégíti azokat az igényeket, hogy olyan képek kapjak amin jól lehet detektálni az emberek arcát. Itt látni lehet majd, hogy semmilyen speciális felszerelés nem kell ahhoz, hogy felépítsünk egy ilyen rendszert és ami a legjobb benne, hogy ez egy költséghatékony módja a fejlesztésnek. Ez nem csak amiatt előnyös, mert nem kell diákként sokat költeni egy komplexebb berendezésre, hanem mert így olcsón lehet megoldani egy olyan problémát amire lehet sokkal többet költenek intézmények.

4.3.1 Kamera

A legelején mikor elkezdtem a projektet azt gondoltam, hogy kell majd egy jobb felbontású kamera egy sima webkameránál, de hamar rájöttem, hogy ez nem így van és nyugodtan használhatom a laptopomban a beépített kamerát.

Ennek főbb paraméterei:

- érzékelő: $\frac{1}{4}$ CMOS
- Színes, 24 bit
- Látószög: 50°-60° körül van
- 640x480- as felbontású
- 30 fps (frames per second) kép frissítésű

A paramétereiből a kamerának látszik, hogy a mai boltokban megtalálható leolcsóbb webkamera már kielégíti ezeket a „megkötéseket”. A feladatom során rájöttem, hogy nem is érdemes 640x480 felbontásnál jobbat használni, mert a

képpontok növelése csak csökkenti a gyorsaságát a software-nek ami nem túl előnyös és ugyan olyan eredményeket lehet elérni egy jobb minőségű kamerával. Tehát amikor jobb kamerát használtam, akkor is az első lépés az volt, hogy a felbontását lecsökkentettem, hogy így növeljem az algoritmusok hatékonyságát.

Egy ilyen kamera ára, ha internetről rendelem, 500-600 HUF körül van, ami azt bizonyítja, nem kell túlköltekezni egy arcdetektáló rendszer készítésénél.



4.2. ábra. Kamera képe színesben és szürkeárnyaltos módban

4.3.2 Raspberry Pi

A Raspberry Pi [9] egy olyan kis méretű számítógép, amit az Egyesült Királyságban fejlesztettek ki oktatási célokra. Különböző Linux disztribúciókkal működtethető. Több verziója létezik (*A*, *B*, *Zero*). Én a Raspberry Pi 2 *B* verziót használom, mivel azon megtalálható integrált Ethernet csatlakozó, 2db USB port, illetve a rendszer memóriája 512 MB. Hivatalosan kifejlesztett operációs rendszer a *Raspbian* [10] amely a Debian Linux Raspberry Pi-ra kifejlesztett változata. Belső tárolóként MicroSD kártya használható.

Ez is azért jó mert szintén olcsónak számít és mivel Linux disztribúció futtatható rajta így telepíthető rá, az OpenCV könyvtár és könnyedén futtatható a Python kód is amiben a program meg lett írva.



4.3. ábra. *Raspberry Pi egy kamerával összekötve*

Forrás: [11]

4.3.3 Laptopom (Windows OS)

A software fejlesztését csak a laptopomon ami egy (Dell Inspiron 5521 - Windows Operációs rendszerrel) végeztem de mivel nem használlok Windows specifikus kódot így Linuxon is ugyan úgy lefut a program.

5 Módszerek kiválasztása

Miután kiválasztottam a megfelelő softwareket, programozási nyelvet, hardwareket és megismerkedtem az elméleti alapokkal így elkezdhettem megvalósítani az arcfelismerő projektemet. Először el kellett döntenem, hogy majd milyen algoritmusokat szeretnék használni, mivel ez az alapja annak, hogy sikeresen elkészülhessen a program. Így nem egyből a tervezésbe vettem bele magamat, hanem több kisebb Python script segítségével kipróbáltam egyes algoritmusok hatékonyságát és használhatóságát. A legfontosabb választott algoritmusokról már leírtam az elméleti alapokat a 3.1. és 3.2. fejezetben. Ezek bizonyultak a leggyorsabbnak és leghatásosabbnak.

5.1 Módszerek összehasonlítása

Mint említettem, a nagy projekt előtt elkészítettem kisebbeket is. Ilyenek voltak például a különálló modulok, mint arcdetektálás, ahol azt próbáltam felmérni, hogy hogy lehet pontosítani és hogy lehet gyorsabbá tenni az algoritmust.

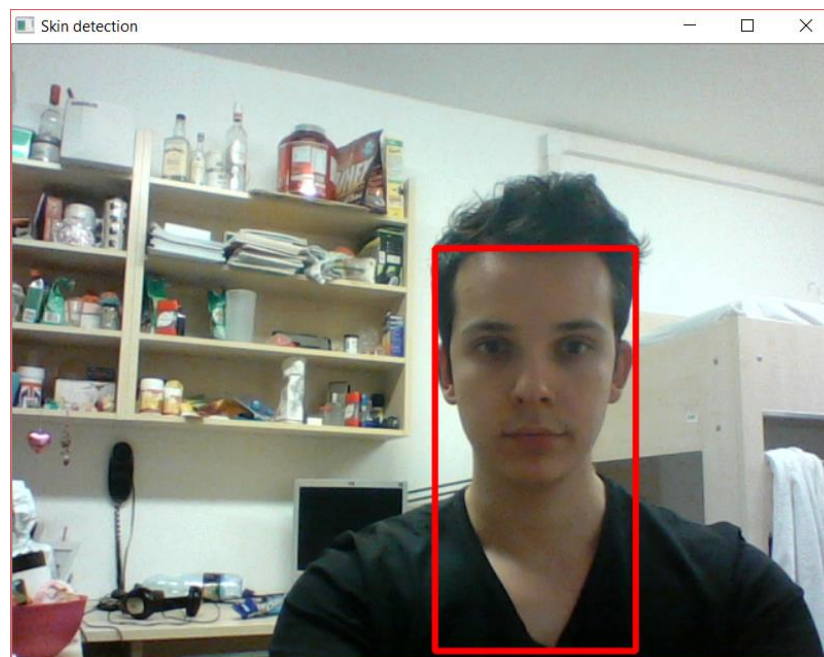
5.1.1 Arcdetektálás

Nagyon fontos, hogy pontosan detektáljon arcot a software mivel, ha rosszak a detekciók és nincs GUI akkor lehet azt hiszi majd a felhasználó, hogy detektálta az arcát a program, miközben lehet, hogy egy másik tárgyat vélt arcnak és így rossz működést fogunk jónak vélni.

5.1.1.1 Börszín alapú szegmentálás

Gyorsítani szerettem volna úgy, hogy az ember bőrszínét detektálom [12][13] (valójában ez egy szín detekció) először és a legnagyobb ilyen detekciót bekeretezem, azaz készítek neki egy *Bounding Boxot*. Aztán ezen a „dobozon” amit *ROI*-nak nevezünk (*Region Of Interest*) belül keresek majd arcot. És mivel az eredeti képnél kisebb lesz ez a terület, így gyorsabban lefut a detektáló algoritmusom.

Ennek előnye azt volt, hogy tényleg pontosabb lett a detektálás (de csak bizonyos esetekben), mivel biztosan csak olyan helyeket néztem, ahol az ember lehet, az arcunkat úgyse takarjuk le annyira, vagy ha igen akkor felismerni sem lehet.



5.1. ábra. *ROI a bőr szegmentálásával*

Hátránya sajnos több volt, mint előnye, mert mint kiderült a pontosságot más módon is el tudom érni és a bőrszín alapú szegmentálás lassabb volt, mint az egész képen lefuttatva az arcdetektáló algoritmus. Nem csak ez volt a gond, hanem robosztus módon nehezebb szegmentálni bőr alapján mivel nagyon sokféle bőrtípus létezik és azok színét még befolyásolja a megvilágítás is ami folyton változik így mondhatom azt, hogy bizonyos esetekben pontosabb volt, de egyáltalán nem volt robosztus megoldás. Tehát ez egy zsákutca volt.

5.1.1.2 Mozgás alapú szegmentálás

Ez a másik ötletem volt, hogy szegmentálhatnám a kép egy részét mozgás alapján, ami nem is lett volna túl nehéz feladat, mivel itt csak el kell mosni (*Gauss Blur*) a beérkező képet, aztán egy előző képből kivonni, és így keletkezik egy különbségek, amit még binarizálok is. Így azokon a helyeken, ahol nem mozdult el semmi feketék lesznek viszont, ha valami elmozdult akkor ott a kép mátrixban 1-esek jelennek meg. Ha ezeknek az összekapcsolódó 1-eseknek száma meghalad egy thresholdot, azaz küszöbértéket akkor azt vehetjük úgy, hogy az a *ROI*.

Ezzel a módszerrel az volt a gond, hogy például az iskolában ha a kamera az osztályból kifelé néz, és több diák elsétál a kamera előtt, akkor mozgást detektál, de nem a megfelelő régiót választja ki, mivel annak a detekciónak a területe nagyobb lesz mint annak a területe aki éppen a kamerába néz. Így ezt is elvetettem.

5.1.1.3 Szemek keresése

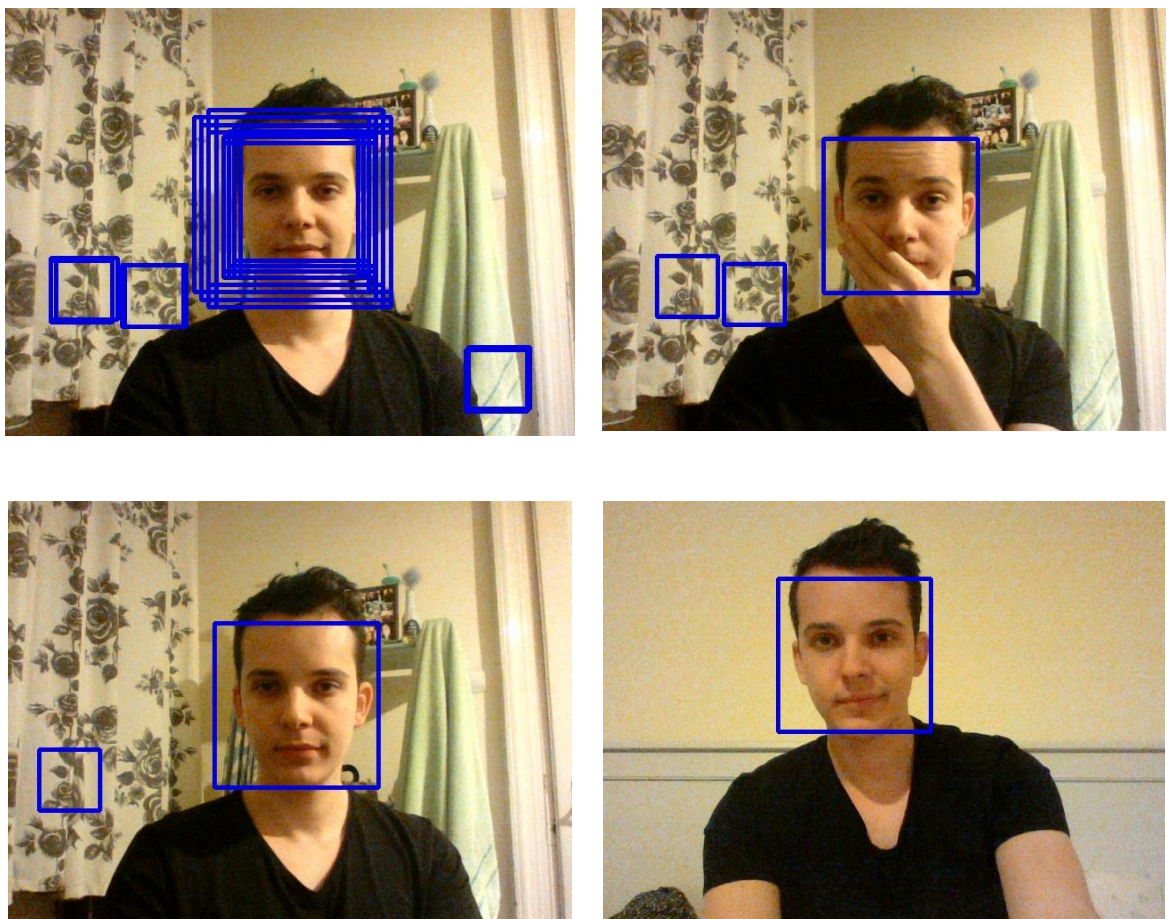
Ennél a módszernél csak a *Viola-Jones* objektum detektort használtam. Az OpenCV-ben megtalálható előre betanított osztályozók és készült nem csak arcra, hanem bal és jobb szemre is, így egy képen azokat is detektálni tudom az arccal együtt.

Itt az volt az ötletem, hogy ne csak arcra futtasunk detektálást, hanem szemekre is. És akkor mondhatjuk egy találatról, hogy az pontos, ha van egy nagy bounding boxunk amiben a fej van, aztán abban, ha van pontosan 2 másik kisebb bounding box amikben a szem van, akkor megtaláltuk pontosan az arcot.

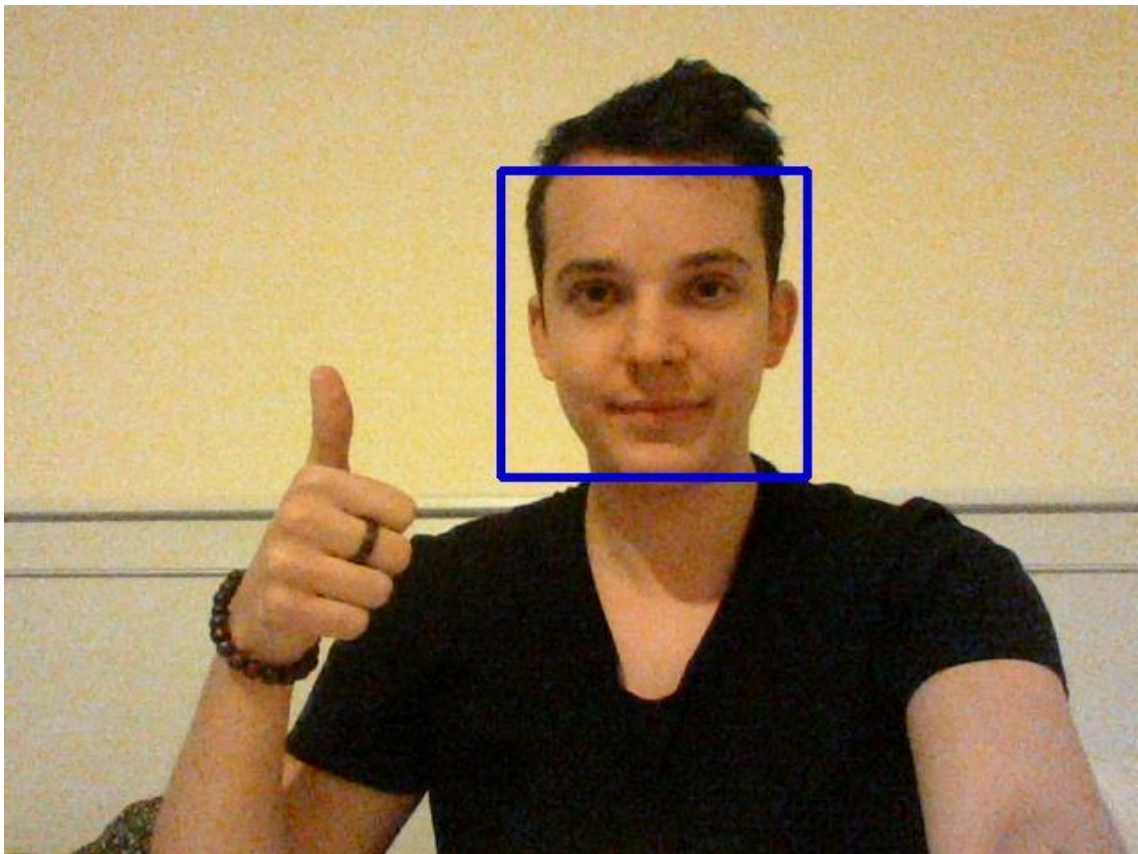
Érezni lehet, hogy ezzel lassulni fog a futása a programnak mivel egyszerre 3-szor futtatjuk ugyan azt az algoritmust, viszont ezzel pontosabban tudott detektálni, mint az előző metodikákkal.

5.1.1.4 Haar-Cascade paraméterek

A fent említett módszereket mind kipróbáltam és mindegyik amennyire segített a pontosság vagy a gyorsaság növelésében annyira le is rontott valamit ezek közül. Így utána néztem az OpenCV dokumentációjában a Haar-Cascade-al való objektum detektálásnak és észrevettem, hogy a `CascadeClassifier::detectMultiScale` metódusában be lehet állítani egy paramétert ami a `minNeighbours`. Ez alapból 3-ra van állítva, viszont, ha ennek növelem az értékét 5-re akkor máris pontosabban detektál az algoritmus. Ez a paraméter és értéke azt jelenti, hogy minimum hány szomszédjának kell lennie a detekciónak, hogy azt arcnak vehessük. Mint ahogy a 3.1.1 fejezetben említettem, egy arcot többször is detektál az algoritmus aztán ezekből egy *Non-Maximum Supression* alapján kiválasztja a legjobbat. Itt csak akkor választja ki, ha teljesül az a paraméter, amit mi beállítottunk a `minNeighbours`-nek.



5.2. ábra. *Különböző minNeighbours értékek hatása a detektálásra
(Fentről le és balról jobbra: 0, 1, 2, 3)*



5.3. ábra. *minNeighbours* érték 5-ösre van állítva

Ennek a paraméternek az értékét változtatva próbálkoztam, hogy mi lehet a legoptimálisabb. És sok próbálkozás után és internetes utána olvasás után rájöttem, hogy ha 5-nek választjuk meg, akkor kaptuk a legjobb eredményt.

Így a detektálónk egyszerre pontos is és gyors is, mert nem használunk semmit pluszba. Ezzel a módszerrel lehetséges a 26 kép/másodperces detekció.

5.1.2 Arcfelismerés

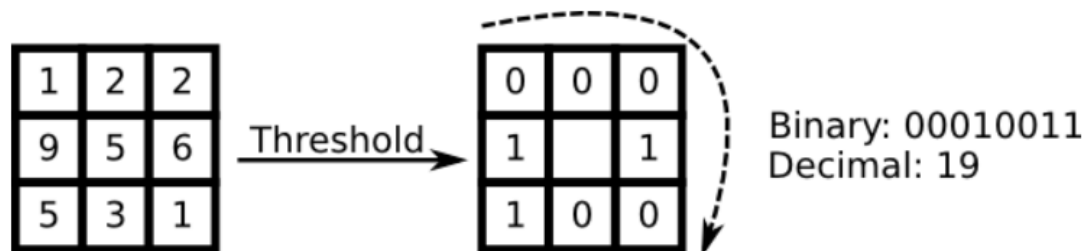
Nem csak a detektálásra, de arcok felismerésére is több fajta módszer létezik, amik közül többet kipróbálva végül a 3.2.1-ben bemutatott Eigenfaces-re esett a

választásom. A megoldásokban vannak tisztán képfeldolgozáson alapuló módszerek de bevethetjük a Neurális Hálózatokat is, jó eredmények eléréséhez.

5.1.2.1 Local Binary Patterns

Az Eigenfaces és Fisherfaces-nél úgy kezeljük az arcokat, mint vektorok, amik elég nagy dimenziójúak tudnak lenni (ami, mint tudjuk nem optimális sok esetben). Viszont a LBP módszernél megpróbálunk lokális jellemzőket kinyerni a képből és aztán az alapján megmondani, hogy kihez tartozik az arc. Ekkor nem kell az egész képre koncentrálni, így nem lesz akkora a dimenzió száma, mint a másik két esetben.

Ennek az algoritmusnak a gyökerei a 2D textúra analízishez visz vissza minket. Az alap lényege az, hogy összegezzük a lokális struktúrákat egy képen, úgy, hogy a szomszédos pixeleket hasonlítjuk össze. Vesszük egy pixel értékét és az alapján thresholdolunk a lokális ablakban. (Ha nagyobb vagy egyenlő a középső pixel értékénél a szomszédos akkor 1-es kerül a helyére, ha kisebb akkor 0. Ezt binarizálásnak hívják)



5.4. ábra. LBP alapja

Forrás: [4]

Ezt kipróbálva egész jó eredményekre jutottam, ami 75%-os találati arányt jelent de mint kiderült ez inkább ismétlődő textúrák felismerésében jó, mintsem arcokéban ezért elvettem ennek az algoritmusnak a használatát és nem foglalkoztam a pontosításával.

5.1.2.2 HOG + SVM

Egy másik technika, amit kipróbáltam az az volt, hogy bevettem a neurális hálókat és az arcok képéről *HOG (Histogram of Oriented Gradients)* jellemzőket [14] nyertem ki. Aztán ezeket a jellemzőket, mint vektorokat egy *Support Vector Machine*-nek (*SVM*) adtam át, ami adatok klasszifikálására jó. A tanítás eléggé gyors volt, csak pár perc legrosszabb esetben mikor sok adatunk volt (20 osztály és osztályonként 3 vektor, tehát 60 bemenő adatunk volt), mivel csak nagyon kevés tanító adatot tudok használni. Emberenként úgy kell számolnom, hogy maximum 5 képem lesz de inkább 3.

Egy ember számított egy osztálynak és ezután, ha kaptam egy képet egy arcról, utána csak ki kellett nyernem a *HOG* leírókat róla és a már előre betanított modellemmel prediktálni tudtam, hogy kire hasonlít.

Mikor sok adattal próbáltam ki, és kevés osztállyal, akkor meglehetősen jól működött és 80%-os találati arányt biztosított. Viszont mikor kevés adatom (képem) volt, viszont annál több osztályom, akkor nem működött jól és 60-65%-os találati arányt tudtam csak elérni, ami megengedhetetlen egy ilyen rendszernél.

Előnye a gyorsaság volt ennek a megoldásnak, mivel miután rendelkezünk már egy betanított *SVM*-mel utána csak egy fájlból vissza kellett tölteni és átlagosan 0.03 másodperc alatt megmondta az eredményt.

5.1.2.3 Eigenfaces

Az elméletét már bemutattam ennek az algoritmusnak, viszont most szeretném ismertetni, hogy miért ezt választottam az arcok felismeréséhez.

Előnye az volt, hogy kevés tanító adattal is nagy pontosságot lehet elérni. Legalább is nagyobb, mint a többi módszerrel. Ekkor elég 2-3 kép és 80-85% felett tudja felismerni az arcokat, ami már jó aránynak számít. Mikor egy arcot felismerünk, akkor nem csak azt kapjuk vissza, hogy kire hasonlít, hanem egy számértéket is arra vonatkozóan, hogy mennyire hasonló az illető. Ez alapján azt tudjuk mondani a kódban, hogy állítunk egy thresholdot és ha azt nem haladja meg ez a szám csak akkor vesszük úgy, hogy felismer valaki és akkor már elfogadjuk, a predikcióját a rendszernek.

5.1.3 Összefoglalás

A módszerek, amikkel próbálkoztam és ismertettem csak néhány a sok meglévő közül. A projektemhez végül arcdetekcióhoz a Haar-Cascade-os megközelítést választottam és az arcok felismeréséhez pedig az Eigenfaces alapú megoldást. Látszik, hogy nem volt könnyű feladat kiválasztani a legjobbakat a módszerek közül de ezektől remélem a legjobb eredményt kombinálva, amivel létrehozhatom a valós időben történő arcdetektálót.

6 Megvalósítás

A megvalósítás második szekciójában arról szeretnék írni, hogy miután kiválasztottam a megfelelő algoritmusokat és megismerkedtem velük azután, hogyan készítettem el a specifikált arcdetektáló és arcfelismerő rendszert.

A rendszernek több „fázisa” van mivel nem csak egy feladatot kell ellátnia, hanem először elő is kell készülni az arcok detektálásához. Így az első részben az arcdetekcióról és az arcfelismerő betanításáról szeretnék beszélni, azután pedig az arcdetektálásról mikor már prediktálni tudunk arcok alapján embereket. Miután ezt a két részt lefedtem utána egy gyakorlati alkalmazással egybekötve megnézzük, hogy hogyan lehet jelenlétet generálni akkor mikor a rendszer már úgy ismer fel arcokat, hogy teljesen biztos benne, hogy ki néz a kamerába.

6.1 Python

Miért pont Python? Vagyis úgy jobb kérdés, hogy miért pont ezt a nyelvet választottam, ha van sok más elérhető wrapper OpenCV-hez és alapból C/C++ -ban van implementálva míg a többi nyelv csak wrapper?

Igazából mindegyiknek megvan az előnye és a hátránya, de a wrappereknek az az előnye, hogy amíg mi egy OpenCV függvényt hívunk meg Pythonnal, addig egy C/C++ kód fut le. Ezeknek az extra nyelveknek pont az a célja, hogy mindenből a pozitívumot válassza ki a kombinálás során. Tehát szinte ugyan olyan gyorsan fog futni a kód mintha az eredeti függvényekkel hívnám meg, pedig a Python lassabb a C++-nál.

Egy másik ok pedig amiért erre esett a választásom az pedig az, hogy Python kódot gyorsabban lehet írni, mivel egyszerűbb, mint a többi nyelv. Sokkal könnyebben és sokkal hamarabb lehet vele olyan megoldásokat leimplementálni ami egy másik nyelvel hosszadalmasabb lenne. Ez azért nagyon jó, mert így sokat tudtam kísérletezni a forráskód egyes részeivel és így több mindent ki tudtam próbálni gyorsan, mintha egy komplexebb nyelvet választottam volna.

Összefoglalva ez a két pozitívum volt az, ami miatt e nyelv mellett döntöttem és utólag nem bántam meg a választásomat mert egy kellemes, átlátható kód született, aminek háttérében viszont komoly algoritmusok állnak és így mindenki számára átlátható és érthető.

6.2 A rendszer terve

A programnak három részből kell állnia. Először is, legyen egy arc detektálónk ami működik is, ahhoz kell egy betanított *Viola-Jones* osztályozó, amit el tudunk tárolni egy .xml fájlban. Szerencsére nekem nem kellett törődnöm ezzel mivel az OpenCV library-ben megtalálható egy már nagyon sok mintán betanított klasszifikáló, ami nagyon jól működik.

Miután van arcdetektáló algoritmusunk, így kell egy osztályozó, ami, ha már van egy arcunk tudja majd valakihez kötni. Ez van megvalósítva az Eigenfaces-el, ami megmondja, hogy melyik archoz van közelebb az, amit osztályozni szeretnénk. Azért használom itt azt, hogy osztályozás, mert igazából, ha osztályokként gondolunk a különböző emberekre, akkor a problémát meg lehet fogalmazni úgy is, hogy melyik osztályba tudom sorolni a detektált arcot.

És végezetül, mikor már arcokat tudok felismerni is, akkor kell egy olyan modul, amivel valamilyen módon jelenléte tudok generálni. Itt sok probléma felmerülhet, mivel, ha csak az első felismerés címkéjét nézem, akkor azt lehet hibásan határozza meg, viszont utána lehet, hogy már jó lesz mindig. Különböző módszerekkel itt is lehet pontosítani a *recognition* (angolul: felismerés) részén.

6.3 Project felépítése

A programom több scriptből, mappából és egy konfigurációs json fileból áll, amiket most szeretnék bemutatni.

6.3.1 Mappák

- *cascades*: Ebben a mappában tárolom a Haar-Cascade-hoz tartozó .xml kiterjesztésű osztályozó fájlokat, amiket könnyen felhasználhatok különböző detektorok létrehozásához.
- *input_images*: Ennek a mappának az almappáiban vannak az egyes személyek mappái melyekben az illetőről készült fényképek helyezkednek el. Ha nem webkamerán keresztül történik az adatok létrehozása akkor használjuk ezt a mappát.
- *output_images*: Olyan a felépítése, mint az *input_images*-nek, mivel az ott tárolt képek „eredményeit” tárolom itt vagy pedig a kamerás adatgyűjtés eredményeit, azaz az emberek arcait.
- *saved_model*: A betanított modellt mentem el ide és a nevek számmá konvertált sorozatát, amiből majd vissza tudom állítani az eredeti neveket.

6.3.2 Scriptek

- *face_recognizer_menu.py*: Ez az a script, amit, ha elindítunk egy konzolos menüt láthatunk, hogy könnyebben navigáljunk az egyes pontjai között a programnak.
- *prepare_faces_for_training.py*: Ha az *input_images*-ben tárolt képeket akarjuk előkészíteni, akkor ezt a scriptet kell lefuttatni.
- *prepare_faces_from_training_from_webcam.py*: A webkamera képét felhasználva tudunk real-time arcokat gyűjteni egy illetőtől és úgy személyeket hozzáadni. Az itt előkészített képek az *output_images*-ben lesznek elérhetőek.
- *train_face_recognizer.py*: Az *output_images* mappában lévő képekkel és annak almappáinak nevével (mivel azok szolgálnak címkéként) betanít egy arcfelismerőt.
- *recognize_face_on_camera.py*: Ha tesztelni akarjuk a felismerőt akkor ezt a scriptet futtatva megjelenik egy GUI ahol a kamera képe látszik, és

az arcok bekeretezve. Az arcok felett pedig látható lesz a predikció a konfidenciával együtt.

- *recognize_face_create_attendance_sheet.py*: Ez már az elkészített alkalmazás, ami egy jelenlétet generál le a predikciók által, aminek kimenete egy Excel táblázat a detekciók nevével és időpontjával.

6.3.3 Konfigurációs fájl

Létrehoztam egy konfigurációs fájl is: *settings_for_recognition.json*. Ez azért nagyon fontos mert több scriptben használom ugyan azokat a változó értékeket, mint például a felismerésnél a kép mérete. Így csak 1 helyen kell módosítanom, ha valami más konfigurációt szeretnék kipróbálni és nem fognak abból hibák adódni, hogy valamit máshogy állítottam be az egyik script-ben, mint a másikban.

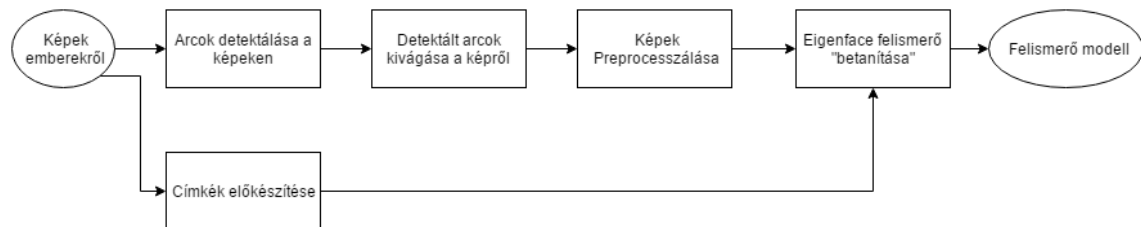
6.4 Arcfelismerő tanítása

Ez az egyik legkritikusabb pont egy ilyen alkalmazásnál, mivel, ha rosszul készítjük elő az adatokat, rosszul címkézzünk vagy esetleg kihagyunk valakit akkor már is nem úgy fog működni, mint ahogy mi azt elképzeltük. Ezért ezt a lépést kell megtenni a legóvatosabban és hogy például címkehibával (mikor rossz osztályba sorolunk valakit, mert Dóra Péter helyett Dóra Petit írunk) ne kelljen törődnünk, így minél nagyobb részét ennek a gépre kell bízni, mert az nagyobb biztonsággal konvertál stringket számmá.

Több lehetőséget létrehoztam a programban annak megadására, hogy melyik arc kihez tartozik. És természetesen a különböző módszerek használhatók egyben is. Például a tanár kérhet képet a diákok arcáról, amiket mappákban tárol. De mi van akkor, ha nincs kép valakiről? Ez sem lehet gond, mert akkor csak leülteti egy kamera elé a tanulót és elkészül pillanatok alatt a saját mappája. A módszereknek ugyan az az alapja, aminek ábráját itt láthatjuk.

A kódomban a *prepare_faces_for_training.py*, *train_faces_for_training_from_webcam.py* és a *train_face_recognizer.py* scriptek

felelősek azért, hogy létrehozzam az arcfelismerőt ami igazából a szíve a software-nek mert anélkül nem tudunk senkire mondani semmilyen predikciót.



6.1. ábra. Arcok betanítása felismeréshez

6.4.1 Arcok kivágása a képről

Először kell egy kép egy emberről, ezt kaphatjuk többféle módon is, videó framejeit feldolgozva, vagy sima digitális fényképezővel, mobillal készült képeket beolvasva.

Ezután arcokat kell detektálni a képen és nem elég csak detektálni, hanem lokalizálni kell úgy, hogy megkapjuk annak a négyzetnek a koordinátáit, amiben az arc elhelyezkedik, így „kivághatóvá” válik mivel tudjuk, hogy melyik részét kell nézni a képnek ahhoz, hogy ott arc legyen.

Ezek kívül még egy fontos része van, a személy nevét át kell alakítani egy címkévé, amit én egész számokkal határoztam meg. (Például: *Peti* = 1, *Dóri* = 2, *Gábor* = 3)

6.4.2 Arcok előkészítése

Miután megvan az arc, elő kell készítenünk, ami annyit jelent, hogy megnézzük a hisztogramját a képnek és megpróbáljuk kiegyenlíteni a sötét és világos árnyalatokat.

Így kicsit általánosítani tudunk, és megóvni a programot attól, hogy kisebb fényváltozásokra rossz működése legyen.

Még az előkészítésbe tartozik bele, hogy át kell méreteznünk az arcképet, hogy mindegyik ugyan akkora legyen a tanítás során. Én ezt a méretet *128x128*-ra választottam meg mivel ezzel kaptam a legjobb eredményeket. Mivel kisebb lesz a kép, információvesztést szenvedünk de ez nekünk nem is annyira gond, mivel így az általánosabb jellemzőkre tudunk koncentrálni nem pedig a kisebbekre, amik elrontanák a felismerést (egy pattanás a bőrön, is elég változás ahhoz, hogy messzebb kerüljön az arc az arctérben és így pontatlanabb predikciót kapjunk).

Mappaszerkezet tanításhoz, amit az arcok kivágása és előkészítése után kapunk:

ouput_images/

Gabor/

gabor_1.jpg

gabor_2.jpg

...

Peter/

peter_1.jpg

peter_2.jpg

...

Dora/

dora_1.jpg

...

...

6.4.3 Felismerő betanítása

Most, hogy már vannak arcaink előkészítve egy mappán belül és annak almappáiban (a emberek nevével elnevezve) vannak a képek egy személyről, amik mind

azonos méretűek és mind nagyjából azonos hisztogrammal rendelkeznek alkalmazni tudjuk az Eigenfaces módszert. A 3.1.1-es fejezetben leírt elmélet alapján járunk el és így meg tudjuk tanítani az arcokat a programnak, amiket utána fel tud majd ismerni.

Az OpenCV-ben felkészültek ilyen igényekre és elkészítettek egy FaceRecognizer osztályt, amivel könnyedén alkalmazni tudjuk a fent leírtakat. Csak kell egy tömb, amiben a képeink vannak az arcokról, meg egy másik, amiben a címkék vannak ugyan olyan sorrendben, mint ahogy a képek a másik tömbben.

A címkéket úgy kapjuk meg, hogy a tanítás előtt végig iterálunk a tanítómappa almappáin és mindegyik nevét beolvassuk és átkonvertáljuk egy egész számmá. Ekkor a nevek lehetnek a kulcsok a számok pedig az értékek és így könnyen konvertálhatunk nevet számmá, és számot névvé.

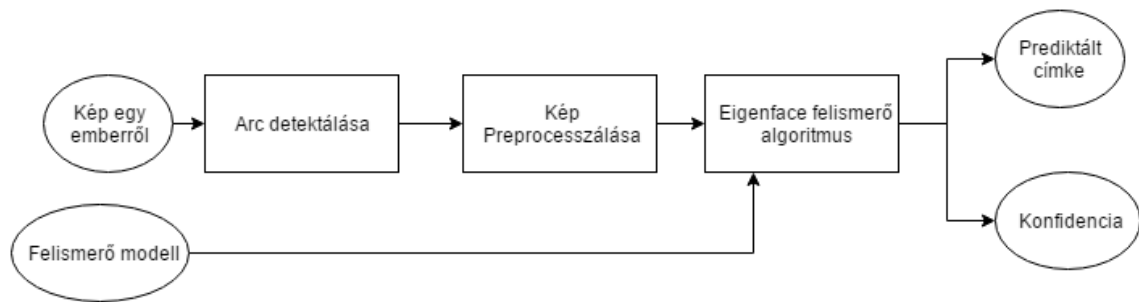
Ezután az osztály `FaceRecognizer.train(faces, labels)` függvényével elvégezzük a betanítást. Ennek futási ideje meglepően gyors, mivel kevesebb mint 1 másodperc alatt (25 osztály és osztályonként 4 kép) létrehoz egy modell fájlt amit el tudunk menteni későbbi használatra. És ezt a modellt fogjuk használni a többi felismerő kódban.

Ez a gyorsaság nagyon jó nekünk mert egy olyan helyzetben mikor többször is nő az „adatbázisunk” azaz nő a felismerendő emberek száma, nagyon gyorsan újra tudjuk tanítani a rendszerünket és már használhatjuk is a frissen elmentett modellt.

6.5 Arcfelismerő alkalmazása

Miután létrehoztunk egy arcfelismerő modellt utána már alkalmazhatjuk is képeken vagy pedig video feed-en, csak be kell töltenünk a lementett modellt. Az alkalmazásra készített scriptjeim: `recognize_face_on_camera.py`, `recognize_face_create_attendance_sheet.py`.

Ahhoz, hogy felismerjünk egy arcot ahhoz megint detektálni kell a képen az arcot, aztán pedig elő kell készíteni és utána a lementett és betöltött modellel már prediktálni tudunk.



6.2. ábra. Arc felismerése képen

Ahogy a 6.2. ábra is mutatja az elkészített képen először az arcot kell detektálni. A képen, ha detektáltunk arcot akkor azt ki kell vágni és preprocesszálni kell, avagy olyan formába kell hozni, hogy az arcfelismerőnek ne legyen idegen. Azaz ugyan olyan méretű kell, hogy legyen, mint amivel tanítottuk és ugyan úgy a hisztogramot ki kell egyenlíteni, hogy zavaró tényezőket kiszűrjünk a képről a világítással kapcsolatban.

Az előzőleg elkészített felismerő modellt, ha betöltöttük és az arcról a képet elkészítettük akkor felismerhetjük rajta a személyt. Ekkor a `FaceRecognizer.predict(faceImage)` függvényt meghívva visszakapunk egy címkét és egy konfidencia számot, ami igazából egy távolság mérték az elkészített arctérben. Ha ez a szám meghalad egy értéket azaz a `threshold`-ot akkor nem kell jónak tekintenünk. Viszont, ha egy bizonyos érték alatt marad akkor jó a felismerés és egy bizonyossággal mondhatjuk, hogy kihez tartozik az arc.

A `recognize_face_on_camera.py` igazából egy teszt script, amivel kipróbálhatjuk az elkészített modellünk teljesítőképességét, mivel itt látjuk mind a nevet, amit a címkéből konvertáltunk mind a konfidencia számot.

6.6 Jelenlét generálása

A jelenlét generálásánál kicsit tovább kell mennünk annál, hogy egy bizonyos `threshold`-ot átlépjen a predikciónk mivel lehetséges, hogy véletlenül a legelső felismerésnél a küszöb alatti számmal egy másik ember nevét kapjuk és ha ezt már el is mentenénk ahogy megkaptuk akkor nagy hibát követnénk el mivel hamis adatokat

generálnánk. Ezért kell módosítani kicsit a felismerésen, akkor mikor nagyobb pontosságot akarunk elérni. Ezt úgy tehetjük meg, hogy adunk egy kis „emlékezetet” a programnak. Így a t időpillanatbeli arc felismerve függ a $t-1$, $t-2$, $t-3$, ..., $t-n$ felismert arctól. Azt, hogy hány másik detekciótól függ az éppeni felismerés azt mi állíthatjuk be a konfigurációs fájlban.

A kódban ez a módszer úgy jelenik meg, hogy a szokásos és már leírt módszerrel predikciókat készítünk az detektált arcokról ezeket folyton eltároljuk (ha a threshold alatt marad) egy listában amit előzőleg létrehoztunk a program indulásánál. Ezután mindig az utolsó X elemet (aminek számát mi adtuk meg, hogy mennyire legyen pontos a detekció) nézzük csak a listában a többit pedig eldobjuk mivel úgy se használnánk. Python-ban elég egyszerűen lehet listából set-et létrehozni: `s = set(predictions)`. Ezzel a módszerrel a megmaradt X elemű listát „átkonvertálok” egy set-re. Ez azért fontos mert a set-ben csak egyszer jelenhet meg egy értékkel rendelkező elem, tehát nincsenek benne duplikátumok.

Ennél az elkészített set-nél, ha leellenőrizzük, hogy hány elemű, abból meg lehet állapítani pontosan, hogy az utolsó X detekciónál hány különböző embert detektáltunk. Ha ez a szám egynél többel egyenlő, akkor azt mondjuk, hogy nem volt elég pontos, mivel több mint egy ember volt felismerve. De ha pontosan 1-el egyenlő akkor eleget tett az elvárásainknak és elmenthetjük a detekciót és annak időpontját.

Miután kilépünk a programból, akkor legenerálja az Excel táblázatot amit le is ment, így azzal már egy másik kódrészlettel könnyedén elküldhetjük egy előre megadott email címre vagy pedig feltölthetjük Google Drive-ra.

7 Tesztelés

Mint minden komolyabb rendszert, ez is tesztelni kell, hogy lássuk, hogy teljesít felügyelt és valós környezetben is, mivel így határozható meg hasznossága és pontossága. Diákként egy ilyen programot elég nehéz tesztelni, mivel nem sokszor, sőt szinte soha nem áll rendelkezésre 20-25 ember, akiknek arcát megjegyezhetné a rendszer és aztán egy kamera képén felismerhetné.

Így nem tudtam komolyabb méréseket végezni a pontosság megfigyelésére, és különböző futási időkre de megpróbáltam egy általános szemlélethez jutni a tesztelésem által.

Még egy fontos jellemzője lenne a rendszernek, hogy bizonyos hardware-en, az én esetemben a Raspberry Pi-on mennyire terhelhető egy rendszer és mi az az esetleges maximum képszám amit már nagyon lassan dolgoz fel, vagy pedig teljesen összekavarodik az arcok sokaságában és folyton rosszul detektál, esetlegesen a memóriából fogy ki.

Az itt felsorolt nehézségek miatt az interneten megtalálható több kép gyűjteményt (kép adatbázist) átnéztem és úgy döntöttem, hogy a sok közül az egyiket fogom használni. Egyik ilyen adatbázis a *The Yale Face Database* [15] amit 1997-ben készítettek el. Ebben megtalálható 15 ember arca és személyenként 11 kép van róluk, amik különböző megvilágításban készültek, különböző arckifejezésekkel. Vannak képek, ahol jobban, vannak, ahol kevésbé kivehetőek az arcok, illetve több arckifejezés is megjelenik, hogy nehezítse a felismerést. Sok képen szemüveg is van, hogy megnézzük az mennyire befolyásolja a detektálást.

Az algoritmusokat, amiket használok az elkészített rendszerben külön külön leteszteltem mind pontosságra, mind futási időre különböző körülmények között, hogy lássam mik lehetnek az esetleges kritikus pontok a programban, amiken javítani kell.

7.1 Arcdetektálás tesztelése

7.1.1 Pontosság

Írtam egy olyan scriptet, ami végig iterál egy könyvtárban található képeken és az ott talált arcokat bekeretezi piros négyzettel és ezeket az új képeket pedig egy másik mappába lementi. Itt végig kellett néznem az összes képet és kézzel összeszámolni, hogy hány jó detekciónk volt.

Nem nézhetjük csak azt, hogy mikor detektált a script és mikor nem, mivel megeshet az, hogy egy képen detektál valamit de az nem egy arc, csak egy annak gondolt része a képnek. Vagy az is gond lehet, hogy mikor csak 1 arc van egy képen akkor többet észlel mint 1, mert az eredeti arcon kívül mást is annak titulál.

A scriptet lefuttatva nagyon jó eredményeket kaptam mivel a 165 képből csak 2 nem talált arcot. Igaz ez egy olyan dataset amin az arcokon kívül nincs sok zavaró tényező, mint a való életben de akkor is elég impresszív teljesítmény. Zavaró tényező alatt értem a háttérrel, ahol előfordulhatnak minták, amik becsapják az algoritmust. Ezt 4.5 másodperc alatt nézte végig, ami azt jelenti, hogy átlagosan egy képpel 0.02 másodpercet törődött.

Ebből is látszik, hogy az algoritmus mennyire pontos és gyors is. Ezért is használtam ezt, kamerából érkező videó feldolgozására, mivel bővel 25-30 képet fel tudok dolgozni másodpercenként semmilyen komolyabb hardware-el.

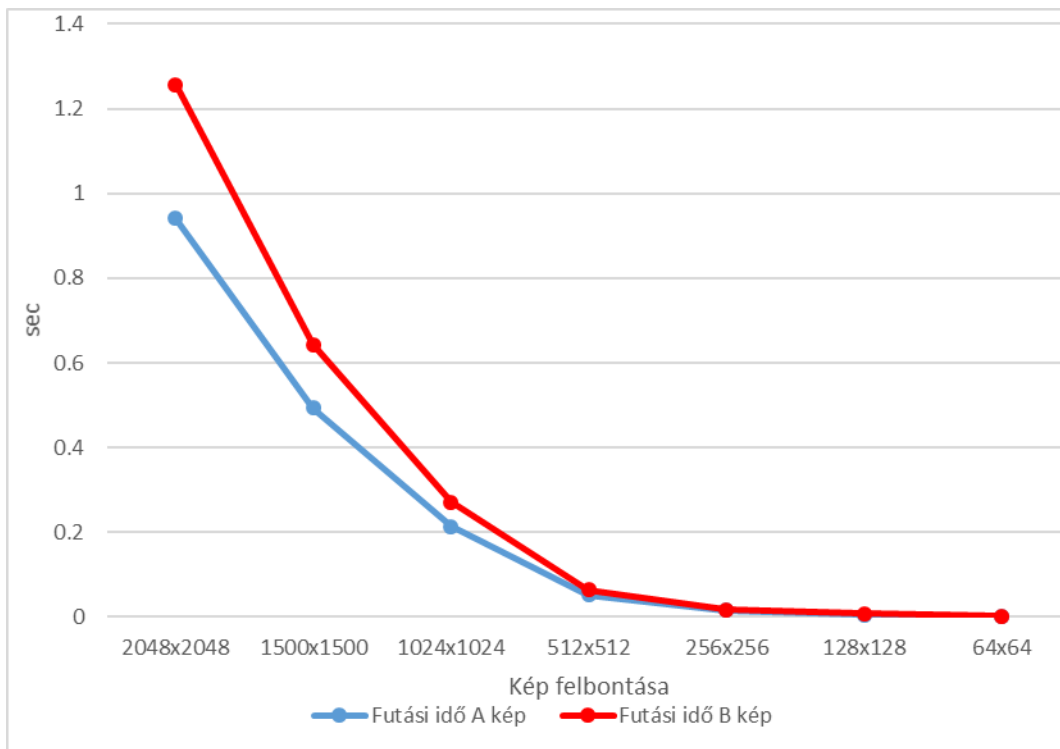


7.1. ábra. 30 példa az arcok detekcióira

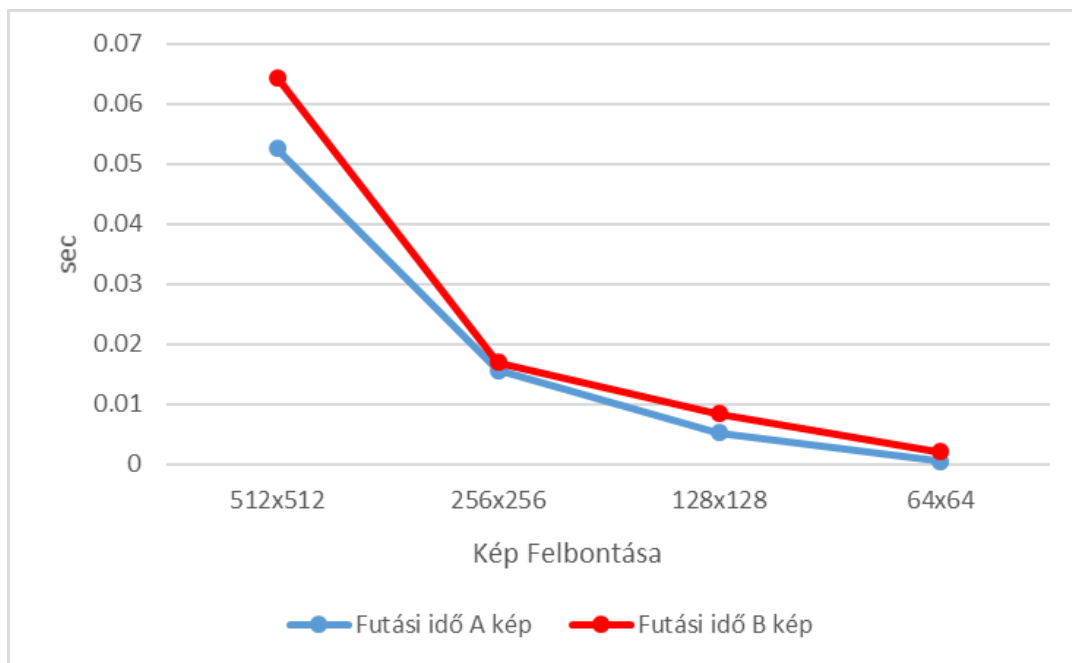
7.1.2 Futási idő

Ahhoz, hogy a futási időt meg tudjam állapítani, kiválasztottam két nagy felbontású képet (2048x2048) amit a telefonommal készítettem. Itt azt néztem, hogy a pixelek számának csökkentésére hogyan reagál a rendszer. Ezeket az eredményeket kaptam:

<i>Felbontás</i>	<i>Futási idő A kép</i>	<i>Futási idő B kép</i>
2048x2048	0.941396103	1.256641061
1500x1500	0.493082886	0.643728091
1024x1024	0.213076725	0.271072195
512x512	0.052653608	0.064387358
256x256	0.015653533	0.016936519
128x128	0.005247142	0.008429646
64x64	0.000564906	0.002139305



7.2. ábra. *A kép felbontásának csökkenésével nő a gyorsaság*



7.3. ábra. *Alacsonyabb felbontású képeknél nagyon gyors az algoritmus*

Innen is látszik, hogy mivel a Viola-Jones detektor végig nézi az összes pixelt a képen, így annak növelésével drasztikusan növelni tudjuk a futási időt. Viszont mindegyik képen megtalálta az arcot, még a 64×64 -es felbontású képen is ami mutatja, hogy robosztus algoritmusról van szó, ami képfelbontástól függetlenül detektál.

Tehát ha nagyon nagy méretű képe lenne a kameránknak, akkor még mindig csinálhatjuk azt, hogy első lépésben átméretezzük, és csak aztán kezdünk el detektálni arcokat. Így a futási időt csökkentjük viszont a pontosság megmarad.

Probléma akkor adódhat, ha jó minőségű képek kellenének az arcról, amiket észrevettünk a képen mivel a felbontás csökkenésével romlik a minőség is. Viszont ezt is meg lehet oldani, úgy, hogy ha tudjuk, hogy mennyivel csökkentettük a képet, és tudjuk a koordinátáit annak a négyzetnek, amiben az arc van mivel ekkor az eredeti képre vissza tudjuk számolni a koordinátákat és onnan ki tudjuk majd vágni.

7.2 Arcfelismerés tesztelése

Ennek a tesztelésére szintén a Yale arc adatbázisát használtam, mivel ott több különböző emberről több kép van és így az arfelismerőt be lehet tanítani, ami olyan mintha a éles programba tanítottuk volna be a felismerést.

Ehhez az arcokat személyenként mappákba kellett csoportosítani 1mappa/ember és ez után az elkészült program *prepare_faces_for_training.py* scriptjével elő lehetett készíteni az arcokat aztán a *train_face_recognizer.py* -el pedig betanítottuk a felismerőt. A pontosság és futási idő tesztelésére írtam egy külön scriptet, ami az eredeti mappákat használja így olyan mintha a rendszert magát tudnánk használni, csak itt nem videóról ismer fel embereket, hanem külön betöltött képekről.

Azt próbáltam megvizsgálni, hogy a pontosság hogyan változik a betanításnál használt képek számával. Ha növeljük a kép számát akkor vajon nőni fog a pontosság is?

7.2.1 Pontosság

7.2.1.1 Összes kép felhasználása emberenként

Először kiválasztottam 5 embert az adatbázisból és azoknak az összes képével tanítottam be az arcfelismerőt, ami azt jelenti, hogy személyenként 15 kép állt rendelkezésemre. Ezután egy másik scripttel végig néztem, hogy ugyan ezeknek a személyeknek a képeivel hogy birkózik meg a rendszer és büszkén mondhatom, hogy *100%-os* lett a találati arány, ami bár nem meglepő, mivel olyan képeken prediktáltam amikkel tanítottam de azért nem rossz eredmény. Ebből látszik, hogy több embernél és több képnél nem kavarodik össze a program.

Ezek után kipróbáltam olyan arcokon is amik teljesen más emberektől vannak, és azt tapasztaltam, hogy visszaadja a legközelebbi arcot de nagyon nagy lesz a konfidencia száma. Míg a *100%-os* találati aránynál 0-9 között kaptam számokat, ekkor 2000 feletti volt ez a szám, ami azt jelenti, hogy messze helyezkednek el az arctérben. A valós programnál ilyenkor csak be kellene állítani egy 500-as thresholdot és a felett ismeretlenként kezelné a program a detektált arcot.

7.2.1.2 Képek egy részének a felhasználása emberenként

A teszt második felében csak 4 képpel tanítottam személyenként az Eigenface algoritmust és miután elkészült a modell, amit lementettem kipróbáltam az összes képpel, ami csak volt a személyről. Az alsó ábrán az látszik, a futás eredménye az első emberre.

```
D:\Face-Recognition>python rec_face_on_image_test.py
Predicted person is: subject_1; Confidence: 4.9891530556
Predicted person is: subject_1; Confidence: 19.9524428766
Predicted person is: subject_1; Confidence: 36.5925831674
Predicted person is: subject_1; Confidence: 4.97545668819
Predicted person is: subject_1; Confidence: 1473.13712846
Predicted person is: subject_1; Confidence: 1773.52868173
Predicted person is: subject_3; Confidence: 3122.24141091
Predicted person is: subject_1; Confidence: 1565.43142138
Predicted person is: subject_1; Confidence: 1057.27703538
Predicted person is: subject_1; Confidence: 1063.46133702
Predicted person is: subject_1; Confidence: 1684.43100201
```

7.4. ábra. *Futás eredménye az első személyre*

Hogy mi is látszik ebből? Először is az, hogy nagyon jól működött a felismerés. 11 képünk volt összesen, és 4-gyel tanítottunk emberenként tehát volt a rendszer számára 7 teljesen ismeretlen kép, mégis 1 kivételével mindig a helyeset adta vissza. Tehát az általánosító képessége jó volt az algoritmusnak.

Még az is látszik, hogy az első 4 alkalommal nagyon kicsi a konfidencia szám mivel azokkal a képekkel tanítottam meg a felismerőt, utána látszik, hogy 2000 alatti számokat kapunk, egyetlen egy eset kivételével amikor rosszul is működött a rekognizáció mert ott a 3-as embert prediktáltuk ami nyilván rossz válasz. De itt is legalább úgy mondott rossz választ, hogy kiemelkedő a többi közül mivel 3000 feletti a szám, amit kaptunk, tehát ezt is inkább ismeretlennek lehet mondani, mint felismerésnek.

Az összes többi esetben is kipróbáltam, hogy mik az eredmények és szinte ugyan azt kaptam, mint az elsőnél.

```

D:\Face-Recognition>python rec_face_on_image_test.py subject02
Predicted person is: subject_2; Confidence: 46.3712978314
Predicted person is: subject_2; Confidence: 8.95475223464
Predicted person is: subject_2; Confidence: 2.67205937012
Predicted person is: subject_2; Confidence: 4.10276672818
Predicted person is: subject_2; Confidence: 802.562066505
Predicted person is: subject_2; Confidence: 802.562066505
Predicted person is: subject_2; Confidence: 1743.08022666
Predicted person is: subject_2; Confidence: 1073.2362481
Predicted person is: subject_2; Confidence: 929.79833739
Predicted person is: subject_2; Confidence: 1682.79266284
Predicted person is: subject_2; Confidence: 811.691690438

D:\Face-Recognition>python rec_face_on_image_test.py subject03
Predicted person is: subject_3; Confidence: 10.9506634233
Predicted person is: subject_3; Confidence: 19.0277983257
Predicted person is: subject_3; Confidence: 16.6473134085
Predicted person is: subject_3; Confidence: 6.07491273711
Predicted person is: subject_3; Confidence: 1002.48995654
Predicted person is: subject_3; Confidence: 1002.48995654
Predicted person is: subject_3; Confidence: 2536.61530683
Predicted person is: subject_3; Confidence: 1166.27522245
Predicted person is: subject_3; Confidence: 1012.08315359
Predicted person is: subject_1; Confidence: 1604.18030793
Predicted person is: subject_3; Confidence: 1057.18656342

```

7.5. ábra. Második és Harmadik személyeknél a futás eredménye

A második esetben hibátlan volt a találati arány, a harmadik-nál pedig szintén csak egyszer hibázott a felismerő, ami egy nagyon jó aránynak felel meg. Egyik esetben se kaptunk nagyon eltérő távolság értékeket, ami mutatja, hogy körülbelül, hogy kell beállítani majd éles helyzetben a *threshold* értéket.

Eddig csak 5 emberre néztem meg a rendszer működését, de kipróbáltam ezután az összesre is, hogy vajon, hogy fog teljesíteni így a software. Meglepődtem az eredményeken mert az emberek számának a növelése nem jelentett pontosság beli csökkenést. Legrosszabb esetben kettőt hibázott a felismerő, tehát 80% felett volt a pontosság még így is egyes esetekben.

Általánosítva, az összes esetet összeszámolva 90%-os pontosságot határoztam meg, ami egy nagyon jó eredménynek számít.

7.2.2 Futási idő

7.2.2.1 Tanítás futási ideje

A futási időt itt több részre osztottam, mert mikor betanítom a felismerőmet, annak van egy megelőző szakasza, ahol végig megyünk a tanító képeken és detektáljuk, kivágjuk és lementjük egy másik mappába (*output_images*) a képet az arcról. Ez természetesen sok időt emészt fel, de ez attól is függ, hogy mekkora a képek felbontása. Az arcok detektálására megy el a legtöbb idő, aminek a futási idejéről a 7.1.2-ben olvashatunk.

Miután készen vagyunk az adatok előkészítésével, azt be tudjuk tanítani az Eigenfaces algoritmussal. Teszteléseim során, akármekkora adathalmazzal dolgoztam, soha nem haladta meg a *0.08* másodpercet (ez volt a legrosszabb eset).

Tehát összegezve a legtöbb időt az adatok előkészítése viszi el nem pedig a tanítás része.

7.2.2.2 Predikció gyorsasága

Mikor a programunkban prediktálni akarunk valós időben akkor annak is fontos a futási ideje. Ezt úgy mértem meg, hogy mikor a pontosságot néztem, akkor lemértem minden egyes predikciónak a futási idejét, a végén összegeztem őket és átlagot képeztem belőle.

Így megkaptam, hogy átlagosan *0.0181* másodperc alatt kapjuk vissza az eredményt. Ez minden esetben ugyan olyan, nem függ az eredeti kép felbontásától, mivel egy konstans szélesség és hosszúságra méretezzük át, tehát a képek mérete mindig ugyan olyan lesz a predikciónál.

Értékelés

A project során nagyon sokat tanultam a képfeldolgozásban használt algoritmusokról, technikákról és olyan ismeretekre tettem szert, ami a valós életben is használható, és ami fontos, hogy eladható és tovább fejleszthető. Nem csak eladni lehet ezt a tudást, hanem kutatásokhoz felhasználni.

Nem csak programozni tanultam így ez alatt hanem mélyebben megismerkedhettem az elmélettel, ami egy ilyen rendszer mögött van és hogy mennyire komplex egy-egy könnyűnek tűnő algoritmus. Sokszor nem is gondoltam volna, hogy milyen kis „trükkök” azok, amik segítenek pontosabbá, gyorsabbá és így jobbá tenni a programot.

Még sok olyan része van, ami fejlesztésre szorul és pontosításra, hogy egy robosztus és komoly rendszer legyen belőle de így is megállja a helyét. Ezt a tesztelés azaz a 7. fejezetben található mérések is alá támasztják.

Mindent egybe véve, a rendszer, amit elkészítettem az elvárásaimnak megfelelően működik és olyan pontossággal és olyan futási idővel, amik megengedek, hogy valós időben használható legyen.

Az arcok detekciója pontosan és gyorsan működik viszont függ a megvilágítástól, de csak extrém esetben detektál rosszul vagy pedig semmit (például teljen sötétség, vagy erős reflektorfény).

A predikció szintén gyors és pontos de nem 100%-os pontosságú. Ez maga után vonja azt, hogy nem elég, ha egyszer nézzük meg egy arcra, hogy ki lehet, hanem mint ahogy a programban van, több kép és predikció alapján mondhatunk jó végeredményt. Ez olyan gyorsan történik, hogy 20-30 arcot végig nézni egymás után maximum 1 másodperc, így nem kell egy személynek sokat időzni a kamera előtt.

A rendszer ezen tulajdonságaival megengedi, hogy olyan valós környezetben használjuk, ahol ki lehet használni adottságait és terhelhetni is lehet azokat.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak, akik segítettek a munkámat és hozzájárultak ahhoz, hogy a szakdolgozatomat minél jobban el tudjam készíteni.

Első sorban konzulensemnek Kertész Zsoltnak szeretném kifejezni a köszönetemet, aki mind elméleti mind gyakorlati szinten nagy segítséget nyújtott meglátásaival.

Továbbá a családomnak szeretném megköszönni azt a sok segítséget, türelmet és támogatást, ami nélkül nem juthattam el volna eddig az egyetemem és amivel most a szakdolgozatomat megírhattam.

Végül de nem utolsó sorban, barátnőmnek szeretném megköszönni a türelmet és a biztatást, amit a megírás közben kaptam tőle.

Irodalomjegyzék

- [1] Paul Viola, Michael Jones. *Rapid Object Detection using a Boosted Cascade of Simple Features*, 2001
- [2] *Robust Real-Time Face Detection Features* – Paul Viola, Michael Jones (2003)
- [3] OpenCV face detection Tutorial cikk.
http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html. 2016.11.11.
- [4] OpenCV face recognition cikk.
http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html. 2016.11.19.
- [5] Matthew A. Turk, Alex P. Pentland. *Face Recognition Using Eigenfaces*, 1991.
- [6] Eigenfaces – Scholarpedia szócikk.
<http://www.scholarpedia.org/article/Eigenfaces>. 2016.11.19.
- [7] OpenCV – Wikipedia szócikk. <https://en.wikipedia.org/wiki/OpenCV>. 2016.11.23.
- [8] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.
- [9] Raspberry Pi Hivatalos Oldala. <https://www.raspberrypi.org/>. 2016.11.20.
- [10] Raspbian operációs rendszer hivatalos oldala. <https://www.raspbian.org/>. 2016.11.27.
- [11] <http://www.pyimagesearch.com/2015/02/23/install-opencv-and-python-on-your-raspberry-pi-2-and-b/>. 2016.11.21.
- [12] Jorge Alberto Marcial Basilio, Gualberto Aguilar Torres, Gabrielsánchez Pérez, L. Karina Toscano Medina, Héctor M. Pérez - *MeanaExplicit Image Detection using YCbCr Space Color Model as SkinDetection*
- [13] Douglas Chai, King N. Ngan - *Face Segmentation Using Skin-ColorMap in Videophone Applications*, 1999
- [14] Navneet Dalal, Bill Triggs - *Histograms of Oriented Gradients for Human Detection*, 2005
- [15] Arc adatbázis – The Yale Face Database. <http://vision.ucsd.edu/content/yale-face-database>, 2016.11.28.

Függelék

F.1. Paraméterezés

A különböző paramétereket amit használtam megtalálhatók a *settings_for_recognition.json* file-ban.

F.1.1. Arcdetektálás

- **Többszörös detektálás kiszűrése** – MinNeighbours: 5
- **Kép skálázása** – ScaleFactor: 1.3

F.1.2. Arcfelismerés

- **Arc kép mérete** – 52x52-vel kaptam a legjobb eredményt
- **Minimum Konfidencia** – Minimum ekkora konfidenciával kell a predikciónak rendelkeznie: 2000
- **Egymás utáni predikciók száma** – 30 (ennyiszer kell ugyan azt az embert visszakapni prediktálásnál egymás után)

F.2. Környezet

A rendszer fejlesztése az alább felsorolt könyvtárak és software-ek felhasználásával történt egy *Dell Inspiron 5521* típusú gépen *Windows 10 (64 bit)* használatával.

- **OpenCV** – 2.4.12.
- **Python** - 2.7. (64 bit)

Az OpenCV installálható a hivatalos oldalukon megadott instrukciókkal Linux és Windows rendszeren is. Vagy használható még az Anaconda környezet is ahol egyszerűen és gyorsan fel tudjuk telepíteni az OpenCV-t.

F.3. Mellékletek

F.3.1. Github

Az alkalmazás forráskódja szabadon hozzáférhető egy *Github repository*-ban az alábbi címen, <https://github.com/gaborvecsei/Face-Recognizer>.

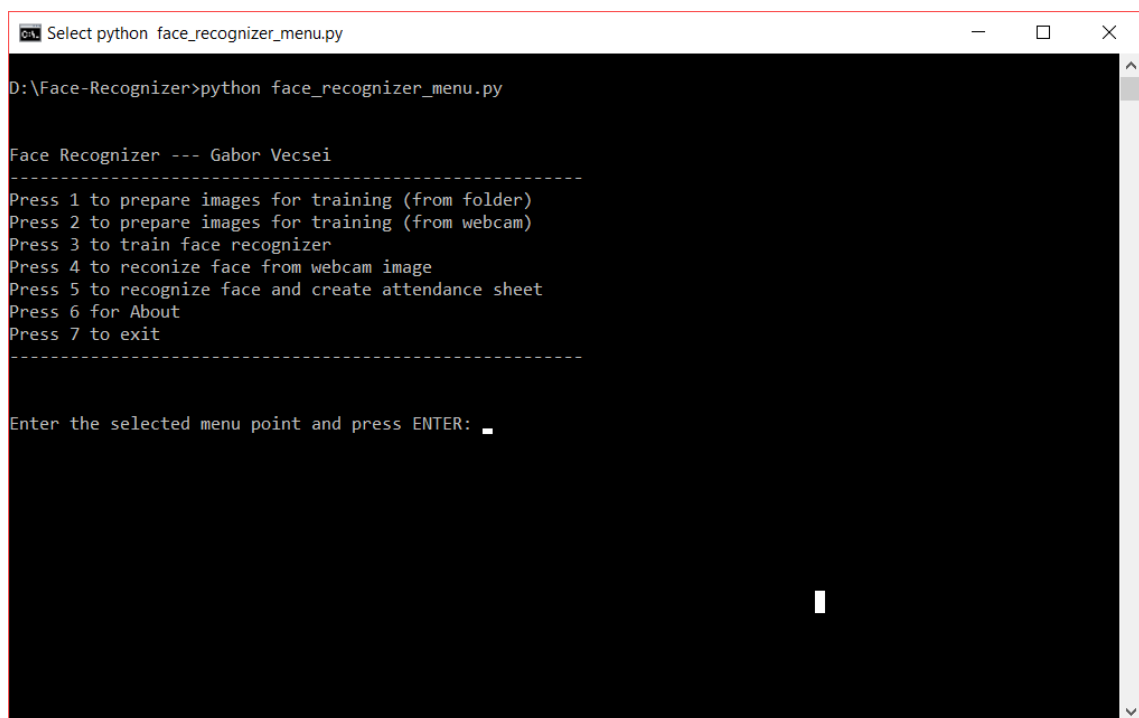
A project megnyitása után egy README.md fájlban le van írva minden instrukció ahhoz, hogy használható legyen az alkalmazás. Illetve ott is meg lehet találni, hogy mi mire jó.

- **cascades** – Ebben tárolom a felhasznált Haar Cascade file-t
- **documentation** – Itt megtalálható a szakdolgozatom PDF formátumban
- **input_images** – Itt tárolhatjuk a még előkészítetlen adatainkat (képeket) emberekről
- **output_images** – Itt lesznek eltárolva az előkészített képek, amiket majd a tanításhoz használunk
- **saved_model** – Miután betanítottuk a felismerőt, ide le lehet menteni a modellt későbbi használatra
- **face_recognizer_menu.py** – Ezzel lehet elindítani az alkalmazást
- **prepare_faces_for_training.py** – A nyers képeket (melyek az *input_folder*-ben vannak) előkészíti a betanításhoz
- **prepare_faces_for_training_from_webcam.py** – Webkamera segítségével helyben előkészítjük a képeket tanításra
- **recognize_face_create_attendance_sheet.py** – Jelenléti ívet generál a felismerésekből és lementi egy Excel táblázatba
- **recognize_face_on_camera.py** – Az arcok felismerését lehet vele tesztelni
- **train_face_recognizer.py** – Arcfelismerő modellt tudjuk vele betanítani az előkészített képek segítségével
- **settings_for_recognition.json** - Ebben találhatóak meg az egységes beállítások

F.3.2. CD-ROM-on

A fent felsorolt online szabadon elérhető forrásokat mellékeltem egy CD-ROM-on is. A könyvtárszerkezete megegyezik az online megtalálható projektével.

F.4. Programból képek



```
Select python face_recognizer_menu.py

D:\Face-Recognizer>python face_recognizer_menu.py

Face Recognizer --- Gabor Vecsei
-----
Press 1 to prepare images for training (from folder)
Press 2 to prepare images for training (from webcam)
Press 3 to train face recognizer
Press 4 to recognize face from webcam image
Press 5 to recognize face and create attendance sheet
Press 6 for About
Press 7 to exit
-----

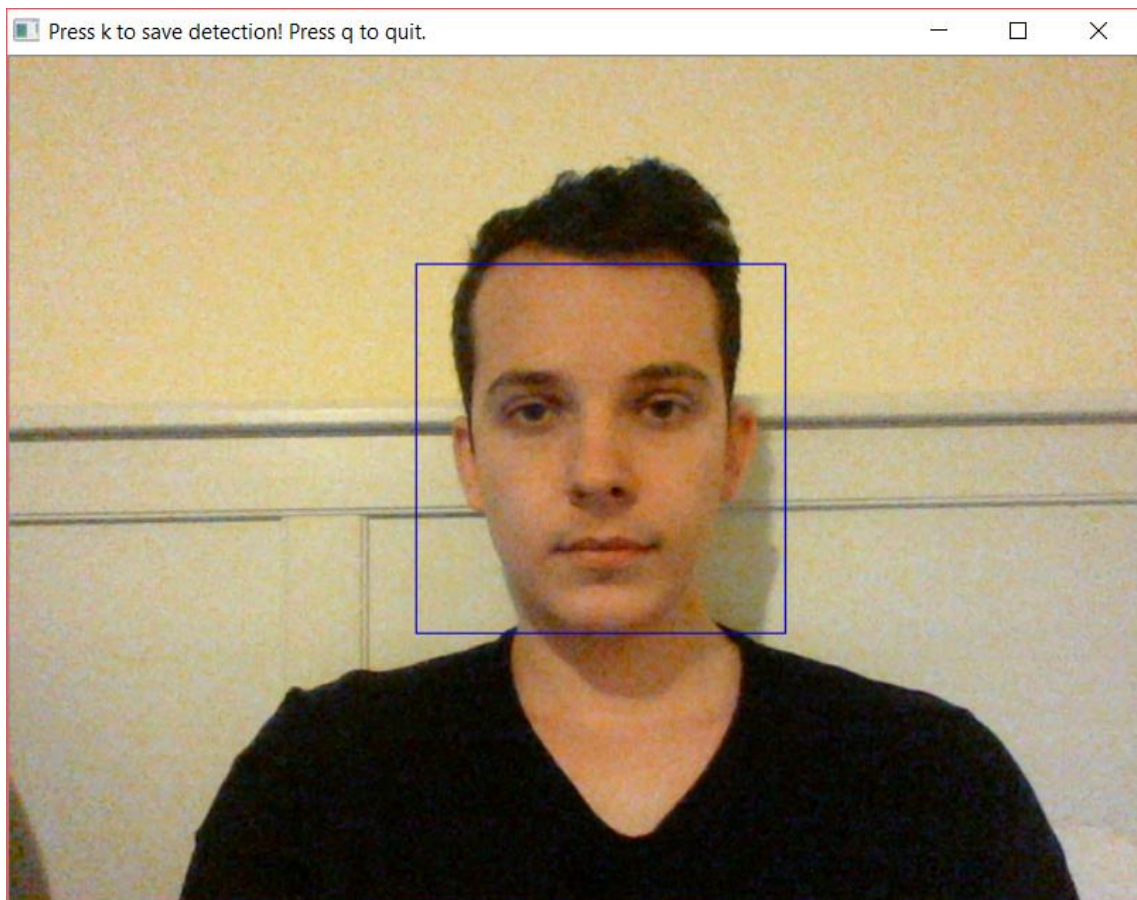
Enter the selected menu point and press ENTER: _
```

7.6. ábra. Program főmenüje

```
python face_recognizer_menu.py
Face detected and cropped: output_images\Dori/dori_1.jpg_1.jpg
Face detected and cropped: output_images\Dori/dori_2.jpg_1.jpg
Face detected and cropped: output_images\Gabor/gabor_1.jpg_1.jpg
Face detected and cropped: output_images\Gabor/gabor_2.jpg_1.jpg
Face detected and cropped: output_images\Gabor/gabor_3.jpg_1.jpg
Face detected and cropped: output_images\Gabor/gabor_4.jpg_1.jpg
Face detected and cropped: output_images\Mona/mona_1.jpg_1.jpg
Face detected and cropped: output_images\Mona/mona_2.jpg_1.jpg
Face detected and cropped: output_images\Peti/peti_1.jpg_1.jpg
Face detected and cropped: output_images\Peti/peti_2.jpg_1.jpg

Face detection and preparation stopped.
Now you can train the face recognizer.
Press (x) to go back to the main menu
x
```

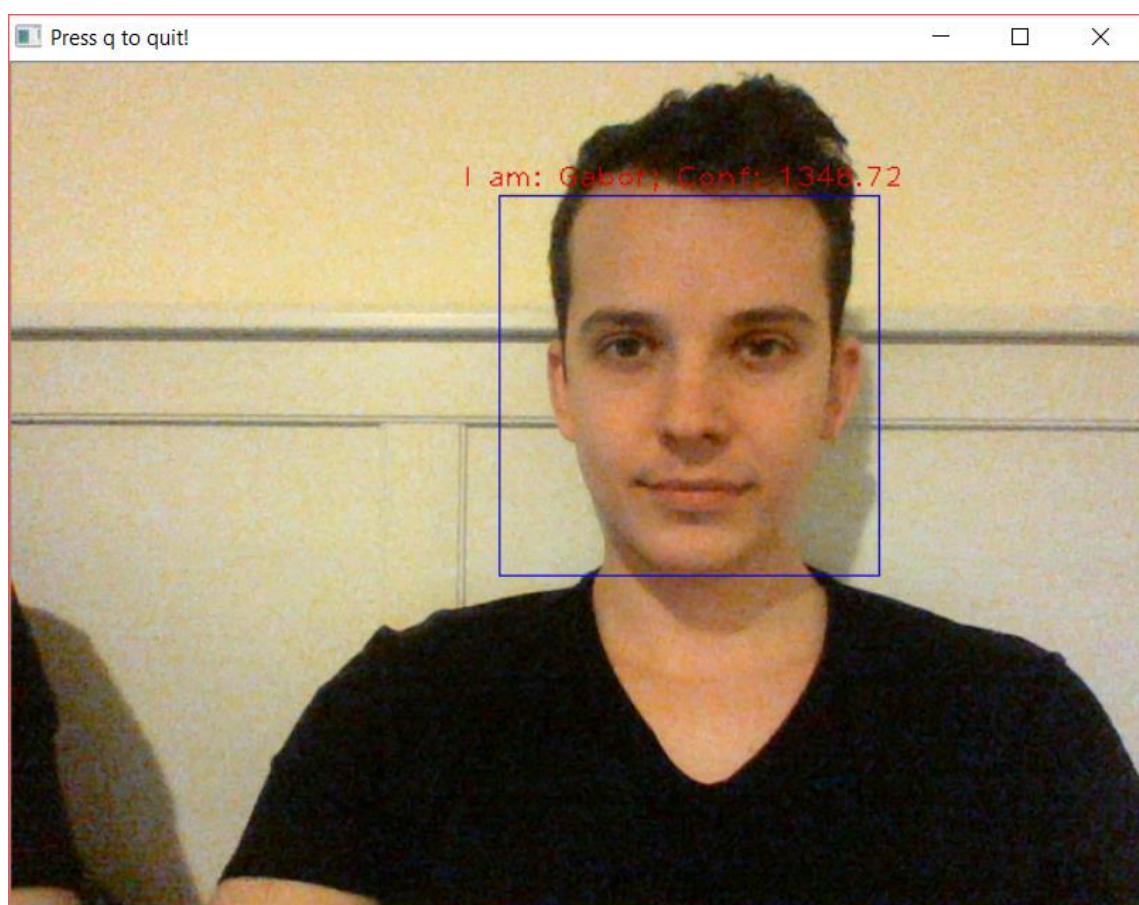
7.7 ábra. Felismerő képek előkészítése mappából



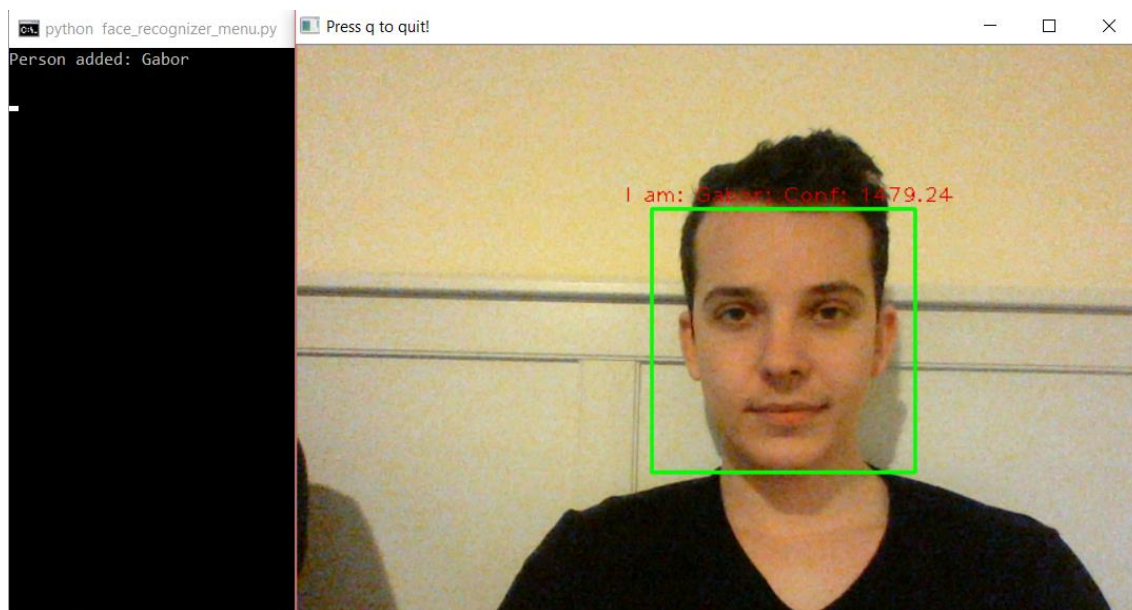
7.8. ábra. Felismerő képek előkészítése közvetlenül a kamerából

```
python face_recognizer_menu.py
Starting the training
Training is completed in: 0.00151808129543 secs!
Would you like to save the trained model?(y/n)
y
The model is saved: saved_model/trainedModel.yml
The name array is saved: saved_model/peopleNames.npy
Training is completed!
Now you can use it!
Press (x) to go back to the main menu
```

7.9. ábra. Felismerő betanítása az előkészített képekkel



7.10. ábra. Felismerő kipróbálása



7.11. ábra. Arcfelismeréssel jelenléti ív generálása