ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

OKOS TÜKÖR RENDSZER TOVÁBBFEJLESZTÉSE ÉS ARCFELISMERÉS OPTIMALIZÁLÁSA

Szerző(k): **Tóth András Tibor**
mérnökinformatikus BSc. szak, III. évf.

Tóth Patrik Sándor
mérnökinformatikus MSc. szak, III. évf.

Mészáros Szabolcs
mérnökinformatikus MSc. szak, III. évf.

Konzulens(ek): **Dr. Halász József**
Egyetemi Docens

Székesfehérvár, 2020.

Okos tükör rendszer továbbfejlesztése és arcfelismerés optimalizálása

Napjainkban megállíthatatlanul terjednek az okos eszközök, életterünk minden egyes részében már találkozhatunk ilyenekkel. Ezek közül egy kevésbé feltárt része az okos tükrök, amik arra alkalmasak, hogy alacsony kognitív funkciókat igénylő feladatok közben információt szolgáltassanak a felhasználóknak. A dolgozat a csapatunk által 2019-ben bemutatott koncepció folytatásaként jött létre, és az alábbiakban az említett eszközön végzett kutatási, fejlesztési és tesztelési folyamatok kerülnek bemutatásra, amelyben új típusú, az irodalomban kevéssé tárgyalt részfolyamatok vizsgálatát is végeztük. Pl. az irodalomban nem ismert az érzelmi domének arcfelismerésre vonatkoztatott hatása.

A kutatás célja a szoftver architektúra átalakítása, hatékonyabbá tétele volt. Továbbá két, hasonló technológiára épülő arcfelismerési eljárás egymással való összevetése annak érdekében, hogy meghatározzuk a projekt számára megfelelőbb rendszert, illetve még annak feltárása, hogy a felhasználók különböző érzelmi állapotai milyen minőségben befolyásolják az azonosítását.

Mivel projekt magnitúdójában túllépte tavalyi mivoltát, a fejlesztés és kutatás haladásának megszervezésére a Scrum metodológia lett kiválasztva. A program korábbi verziójában egy monolitikus szoftver komponenst alkalmaztunk, de a párhuzamosított munkavégzés érdekében ez később három különálló komponensre lett bontva. A szoftverkomponensek elkészítéséhez használt technológiák a dolgozatban részletes bemutatásra kerülnek. Az arcfelismeréssel kapcsolatos mérésekhez a FaceGen által generált adathalmazt alkalmaztunk. Jelentős előrelépések történtek a teljes szoftver architektúra kialakításában. Az egyik ilyen komponens, az adathozzáférési réteg, ami egy adatbázis és az arra épülő webszerver melyek ORM technológiát használva kommunikálnak egymással. A másik két komponens: a tükör felhasználói felülete és a regisztrációs weboldal, az említett webszervert használva éri el az adatokat. Az arcfelismerés során implementálásra került egy új reprezentációs réteg, mely nagyobb dimenzió számmal operál, aminek a hatékonysága kiértékelésre, és a FaceGen modellek segítségével két alapérzelem megjelenésének hatása leírásra került.

Mind az okostükör projektben, mind az irodalomban kevéssé ismert elemek feltárásában sikerült jelentős előrelépéseket elérnünk, ami a termékfejlesztés sarokköveként értelmezhető.

Further development of a smat mirror system and optimazioton of face recogntition

In recent years, we have seen an unstoppable adoption of smart devices in almost all parts of our lives. One area of smart devices that have yet to gain widespread adoption is smart mirrors, which are devices that offer easily digestible information to their users while they are performing low-cognitive tasks. This paper is the continuation of a 2019 paper by our team, where we presented and elaborated on the above concept. This paper details the research, development, and testing processes carried out since then, while also discussing new, lesser-known sub-processes, such as the effects of emotional domains on facial recognition.

The goal of our research was to determine the effect of different emotional states on the accuracy of facial recognition while comparing two different implementations of the same underlying technology and thus decide on which implementation our team should move forward with, all the while improving upon the software architecture and increasing its efficiency.

As mentioned, two different implementations of facial recognition have been used, one being the implementation described in our previous paper and a new representational layer which operates with a higher number of internal dimensions. These two implementations have been compared using a standardized face dataset created via a program called FaceGen.

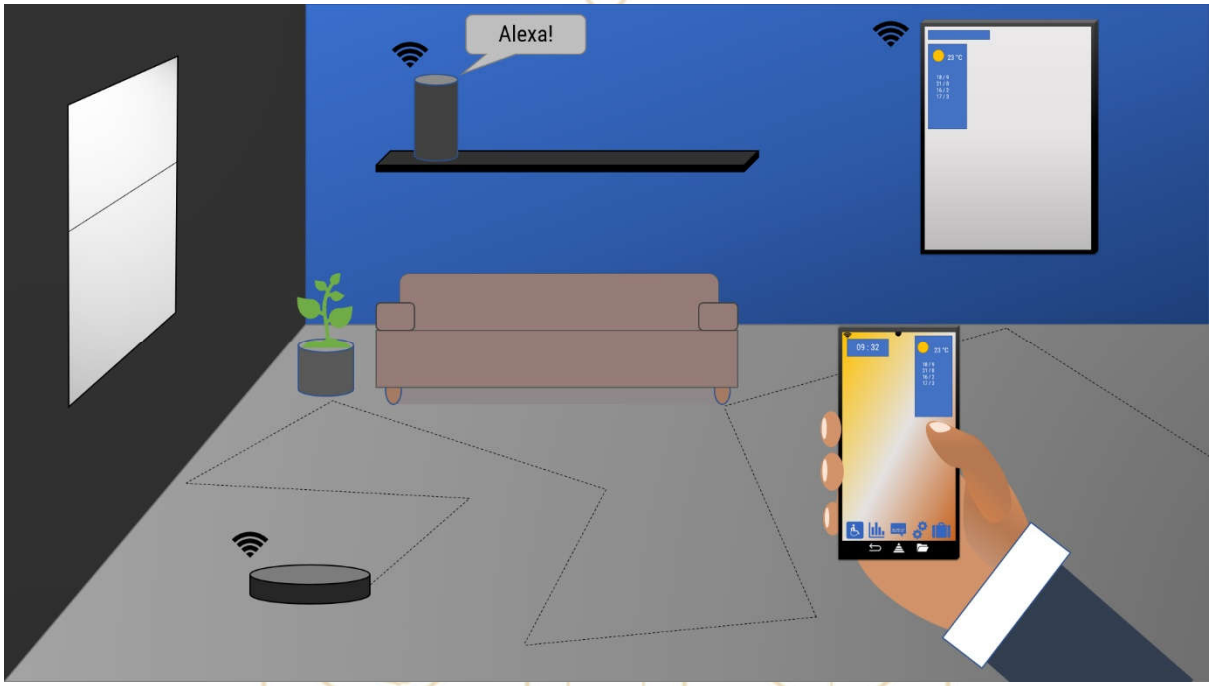
This project has grown in magnitude to a point where our team had to reorganize to utilize project management methodologies, namely, Scrum, and in order to effectively parallelize our workflow, we have separated the software into three distinct components. One of these components is the data access layer, which is a database and webserver, which uses object relational mapping (ORM) to communicate with each other. The other two components are the User interface of the mirror and a webpage that is used for user registration and configuration. Technologies used in the making of these components will be detailed below.

Overall, we have achieved what we have planned with this project, in regards to our research and development goals as well.

Tartalomjegyzék

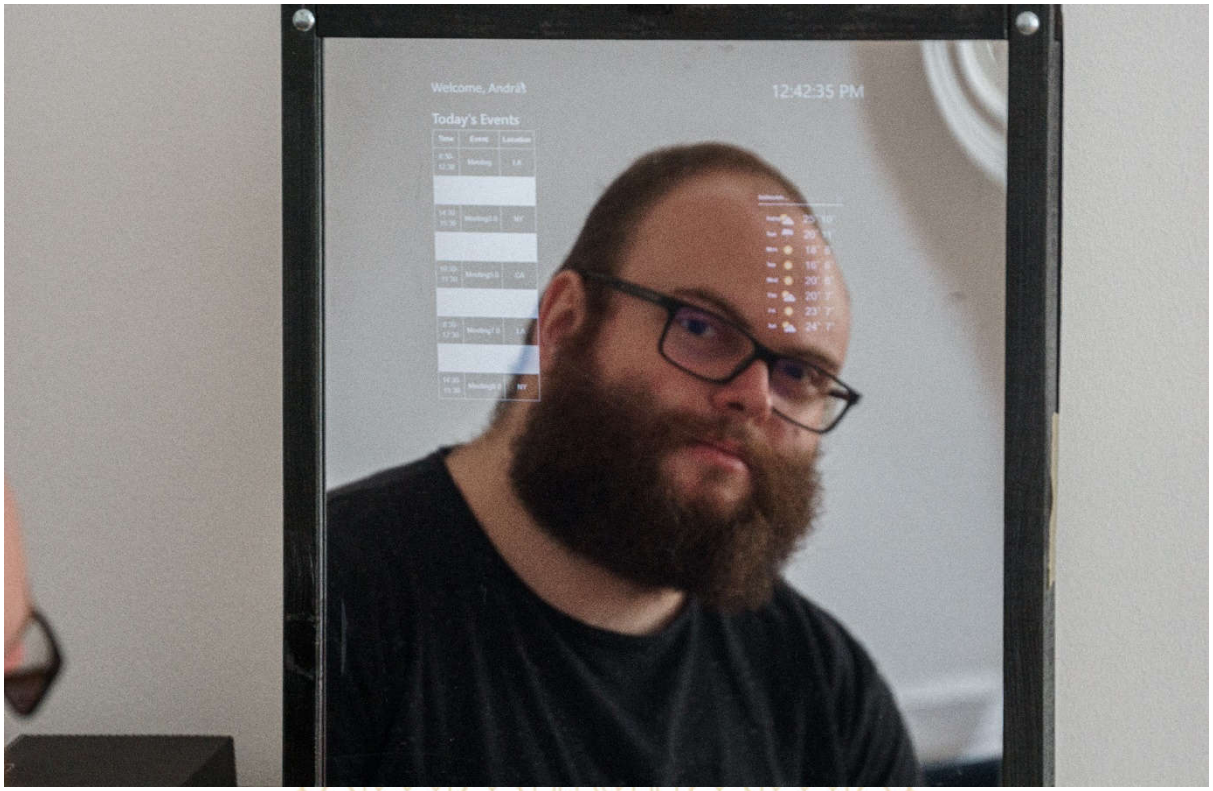
Bevezetés

Az okos eszközök napról napra egyre jobban a mindennapi életünk részévé válnak. Az emberek zöme naponta használja az okostelefonját és más ahhoz csatlakoztatott okos eszközeit, hogy optimalizálják az idejük felhasználását, vagy életmódjukat, annak érdekében, hogy minél nagyobb szinergiában legyenek a mai, összekötött világgal. Ahogy egyre jobban terjednek az okoseszközök, az "okos" technológiák kezdenek megjelenni az épületekben, főleg az otthonokban, hogy részesei legyenek annak a phenoménnek, amit az Internet of Things-nek (IoT) (1. ábra) hívunk. Ennek az IoT szférának egy kevésbé feltárt része az okos tükrök szegmense.



1. ábra Az Internet of Things világa

Az okostükrök alapvetően olyan eszközök, amelyek képesek arra, hogy információt szolgáltassanak (2. ábra) nekünk miközben alacsony kognitív funkciókat igénylő feladatokat végzünk, min például a fésülködés, vagy fogmosás. A tükör egy speciális, féligáteresztő bevonatú plexi vagy üveglapból, egy kijelzőből és az ezt működtető szoftverből és hardverből áll. Az évek folyamán sok kevésbé ismert cégnek voltak próbálkozásai ezen a téren, a nagyobb cégeknek, pedig szabadalmai, de olyan termék, amely jól elterjedt volna még nem jött létre.
A csapat az előző TDK dolgozatában bemutatott egy koncepciót és egy kezdetleges prototípust, ebben a dolgozatban pedig egy valódi terméknek szánt eszközök szoftver architektúráját és a technológiák finomhangolása kerül bemutatásra.



2. ábra Okostükör működés közben

A kutatás célja két hasonló technológiára épülő arcfelismerési eljárás egymással való összevetése annak érdekében, hogy meghatározzuk a projekt számára megfelelőbb rendszert. Cél még annak feltárása, hogy a felhasználók különböző érzelmi állapotai milyen minőségben befolyásolják a felhasználók azonosítását.

A rendszerek kiértékeléséhez egy a FaceGen [Singular Inversions Inc., 2020] program által létrehozott 5 felhasználóval rendelkező felhasználói bázis képeiből generált vektorok lettek felhasználva. Mindkét esetben egy Pythonban létrehozott script elindítja a megadott adathalmazra a felismerő rendszert és a feldolgozást, majd olyan formátumban visszaadja a felhasználók képei közötti távolságokat melyek Excelbe importálhatóak és kiértékelhetőek.

A fejlesztés célja az új szoftver architektúra kialakítása volt. A szoftver három különálló komponensre lett bontva. Ezek a következők: regisztrációs felület, a tükör szoftvere és az adat elérési réteg.

Az eszköz tervezése során több kérdés is felmerült a felhasználói interakcióval kapcsolatban. Többek között a legfontosabb kérdés, milyen módon tudja a felhasználó regisztrálni magát a tükör felhasználói közé?

A fejlesztés elején a regisztráció és felhasználók kezelése manuálisan történt azon az eszközön, ahol a tükör szoftvere futott. Ez prototípusnak megfelelő volt, azonban hosszú távon nem volt jó megoldás. Így elkészítésre került egy felhasználói weboldal, ahol a regisztráció elvégezhető, a felhasználó pedig kezelni tudja a saját és a tükör beállításait.
Abban az esetben, ha a tükör nem rendelkezik belső arc tárral, vagy ismeretlen felhasználóval találkozik egy QR kódot jelenít meg a felhasználó felé, melyet telefonjával/tabletével beolvasva egy web felületre kerül, ahol regisztrálhatja magát, valamint személyre szabhatja az általa a tükrön látni kívánt tartalmakat.

Arcfelismerés

Jelen dolgozatban bemutatásra kerül két hasonló technológiával működő arcfelismerő rendszer összehasonlításának általunk eszközölt módszere és az ehhez felhasznált technológiák.
Az arc az emberi ismertetőjegyek legjelentősebbike, segítségével sokkal kifinomultabb biometriai felismerő rendszereket van lehetőség létrehozni. Segítségével minimális felhasználói interakcióval van lehetőség embereket azonosítani. Az arc különböző megkülönböztető jelei, a szemek, a száj és az orr és más térbeli geometriai markerek alapján különbözteti meg az alanyokat. Arcfelismerés egy sokoldalúan felhasználható lehetőség, legyen szó fogyasztói termékekről, megfigyelésről, esetleg bűnüldözésről.
Alapvetően egy arcfelismerő rendszer 3 lépésben működik [W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, 2003].
Ezek közül az első az arc megtalálása és elkülönítése a kép többi részétől.
A második lépés a markerek megkeresése és azok valamilyen módon való reprezentációja, eltárolása. A harmadik és egyben végső lépés az arc felismerése, azaz egy személyazonosság hozzárendelése.

Arcfelismerés módjai

Az arc felismerésének két módja a verifikáció és identifikáció.
Verifikáció esetén az adott archoz tartozó markereket egy egyezőnek vélt, eltárolt sablonnal vetik össze, ez megállapítja, hogy az adott személynek a markerei egyezést mutatnak-e azzal, akinek vallja magát. Ez egyetlen összehasonlítást jelent.

Identifikáció esetén a megadott arcot az összes eltárolt lehetőséggel összevetik, annak érdekében, hogy a legnagyobb egyezést kapjuk. Emiatt az identifikáció időigénye négyzetesen, rosszabb esetben exponenciálisan növekedik az adatbázis méretével.

Arcfelismerést befolyásoló hatások.

Az emberi arcot nem lehet egy megmásíthatatlan, örökös jellemzőnek venni, hiszen belső és külső hatások nagyban formálják annak kinézetét[W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, 2003], [U. Jayaraman, P. Gupta, S. Gupta, G. Arora, and K. Tiwari, Apr. 2020]. Az itt található arc felismerését befolyásoló hatások kategorizálása és a kategóriák kifejtése U. Jayaraman et al Recent Development in Face Recognition [U. Jayaraman, P. Gupta, S. Gupta, G. Arora, and K. Tiwari, Apr. 2020] cikke alapján készült.

Belső hatások

Arckifejezések:

Az arcokra kiülő érzelmek lokális variációkat eredményeznek, azaz bizonyos részei változnak csak meg számottevően. Ezen változásoknak a helyeit nehéz meghatározni, és ennek következményeivel számolni

Kor:

Mivel testünk szövetei idővel változásokon mennek át, így az arcunkon lévő megkülönböztető jegyek is idővel változnak. Könnyen előfordulhat, hogy egy évekkel ezelőtt készült kép alapján egy arcfelismerő rendszer, de még egy átlagember sem tud azonosítani egy illetőt.

Kitakart arc:

Az arc egyes részeit eltakarhatják bizonyos dolgok, legyen az a haj vagy szakáll, esetleg szemüveg vagy más.

Külső hatások

Megvilágítás:

A megvilágítás intenzitása, iránya, vagy az általa generált árnyékok nagy mértékben befolyásolhatja, vagy el is rejtheti arcunk megkülönböztető jegyeit.

Felbontás, méretarány:

Mind a felismeréshez, és az azonosításhoz használt kép felbontása, mind az azon lévő arc méretének aránya a kép méretéhez képest nagy befolyásoló tényező.

Testtartás:

Ez egy nagymértékben befolyásoló tényező, hiszen sok esetben előfordulhat, hogy a képen az alany nem teljes testtel és arccal a kamerának fordulva kerül lefotózásra. Ennek eredménye lehet az, hogy az arc bizonyos részei nem látszódnak, illetve a markerek egymáshoz való relatív elhelyezkedése változhat.

Zaj:

Minden esetben előfordulhat, hogy a kép alkotásáért felelős szerkezet valamilyen jellegű zajnak van kitéve, amely a készült kép részévé válik. Ez egy tipikus probléma lehet egy átlagos kameraszenzorral a sötétben, illetve rossz megvilágítással készült képekre.

Elmosódottság:

Sok tényező miatt lehet az elkészült kép elmosódott: hosszú záridő, stabilizáció hiánya, vagy egyszerűen az alany mozgása.

Arc adatbázisok

A két rendszer összehasonlításához egy olyan arc adatbázisra volt szükségünk, amelyekben minden alanynál azonos mennyiségű kép található méghozzá olyanok, melyek a legtöbb külső és belső hatásoktól mentesek, leszámítva az érzelmek által generált arckifejezésektől. Egy ilyen kombinációt nehéz volt fellelni, ezért készítettük el a saját adatbázisunkat. A következő részben bemutatásra kerülnek a megfontolásra került már létező adatbázisok.

AT&T adatbázis [F. S. Samaria and A. C. Harter, 1994]

Talán az egyik legrégebb óta elérhető adatbázis. 400 képet tartalmaz 40 alanyról. Minden alanyról 10 kép található meg, melyeket különböző időpontokban, megvilágítással,

arckifejezéssel és eltakarással, azaz szemüveggel vagy anélkül. Alacsony felbontású, fekte-fehér képek. Mindenképpen említésre méltó, de a mi esetünkben a képek sokfélesége miatt nem volt megfelelő.

Labeled Faces int the Wild (LFW) [L. J. Karam and T. Zhu, 2015]

Az LFW adatbázis talán az egyik legelterjedtebb adatbázis, hiszen mérete miatt sokoldalúan használható. 13000 arckép található benne 5749 alanytól, amelyeket az internetről gyűjtöttek be. Itt minden alanynak változó mennyiségű kép áll rendelkezésre. Jelen esetünkben a minőség megfelelő lett volna, hiszen jó felbontású, színes képek állnak rendelkezésre, amelyek már forgatva és arcra illesztve elérhetőek. Annak érdekében, hogy a méréseink az arckifejezések tekintetében kontrolláltak legyenek nem emellett döntöttünk.

Mega Face [I. Kemelmacher-Shlizerman, S. M. Seitz, D. Miller, and E. Brossard, 2016]

Ez az adatbázis egy millió képet tartalmaz több mint 690 ezer személyről. A képek minősége, az alanyok rassza, neme és a személyenkénti képek száma nagymértekben változó.

FG-NET Adatbázis

Említésre méltó, hiszen ebben az adatbázisban 82 alanynak összesen 1002 képe található, alanyonként több különböző életkorukban készített képpel. Kifejezésben, pózban és megvilágításban is van variáció. Az egyetlen problémát az jelentette, hogy a képek analóg fotókról lettek digitalizálva, így sok esetben nem kívánatos zajt hoztak a rendszerbe.

Browns adatbázis [N. Nixon, 2014]

Ez az adatbázis egy könyv alapján készült, melyben Nicholas Nixon fotói találhatóak nejéről és 3 testvéréről. 4 alany 36 képe található meg benne, minden kép között 1 év eltéréssel. A képek nagyon jó minőségűek. Ennek a cikknek nem témája az arcfelismerés nehézségei a kor változásával, de mindenképpen említésre méltó az adatbázis.

Módszerek az arc megtalálására és elkülönítésére

A projektben használt két rendszer első eltérése az a módszer, hogy hogyan találja meg a feldolgozásra kerülő képeken az emberi arcokat.

Haar kaszkád

A korábban használt megoldás a Viola-Jones objektum detektálási keretrendszeren alapult [P. Viola and M. Jones, 2001][P. Viola, M. Jones, and M. Energy, 2004]. Ez a rendszer különböző leírók alapján képes több fajta objektumot a képeken megtalálni, legyen az autók, emberek, vagy a mi esetünkben arcok. Ezek a leírók Haar tulajdonságokat [C. P. Papageorgiou, M. Oren, and T. Poggio, 1998] képeznek, a mi esetünkben olyanokat, amelyek az emberi arcra könnyedén ráilleszthetők. A keretrendszernek vitális része a kaszkádosított működés, ami garantálja, hogy csak a kép azon részein végződnek műveletek, ahol nagy eséllyel található arc. Ennek a publikációnak a megszületése óta több cikk is jelezte, hogy ennek a megoldásnak a teljesítménye nagy mértékben romlik, ahogy nő a variáció az emberi arcok között, abban az esetben is, ha összetetteb tulajdonságokat és osztályozókat alkalmaznak [K. Zhang, Z. Zhang, Z. Li, S. Member, Y. Qiao, and S. Member, 2016].

MTCNN [K. Zhang, Z. Zhang, Z. Li, S. Member, Y. Qiao, and S. Member, 2016]

Az arcfelismerés terén sok esetben a korábbi megoldások leváltásra kerültek konvolúciós neurális hálózatokra, legyen szó klasszifikációról vagy felismerésről. Ennek nagy előnye, hogy az optimális algoritmust a neurális háló alakítja ki, nem pedig egy emberi programozó, hiszen a neurális háló olyan összefüggéseket is felfedhet amelyekre a programozó nem gondolna.

Ennek a megoldásnak nagy előnye, hogy az arcfelismerés és azok kiegyenesítése közösen történik, multitaszk módon, ezzel növelve a pontosságát.

Az általunk vizsgált két rendszer ezeket a technológiákat használja, a régebbi rendszer a Haar Kaszkádon alapuló megoldást, míg a bevezetni kívánt az MTCNN-t.

A projektben használt arcfelismerő rendszerek alapja

Az általunk tesztelt rendszerek mindkét esetben a Florian et al által bemutatott FaceNet [F. Schroff, D. Kalenichenko, and J. Philbin, 2015] rendszer implementációi. Ez a keretrendszer verifikációra, identifikálásra és az egy adott emberhez tartozó képek kluszterezésére, azaz csoportosítására is alkalmas. A működésének alapja az, hogy minden képhez egy n dimenziószámú vektort rendel. Így a vektorok Euklideszi távolsága közvetlenül megfeleltethető az arcok hasonlóságának.

A cikkben tárgyalt rendszernél a Schroff és társai a távolságot 0,0 és 4,0 között állapították meg, ahol 0,0 a teljes egyezés a 4,0 pedig a teljes különbözőséget jelölik. Schroff és társai a határértéket 1,1-nek határozták meg [F. Schroff, D. Kalenichenko, and J. Philbin, Jun. 2015], ami alatt ugyanannak a személynek tekintik a két képen szereplő alanyt.

Ennek a megoldásnak a segítségével a verifikáció két vektor távolságának (3. ábra) és egy megszabott határnak az összevetéséből áll, az identifikáció pedig egy k – legközelebbi szomszéd alapú osztályozássá.

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}; \quad d = |x - y| = \sqrt{\sum_{i=1}^{n} |x_i - y_i|}$$

3. ábra Két vektor távolságának kiszámítása

A két általunk vizsgált rendszer különbsége, hogy az eredeti rendszer 128 dimenziós vektorokat alkalmaz, míg az új bevezetni tervezett 512 dimenziós vektorokat használ. A FaceNet-et leíró cikk [F. Schroff, D. Kalenichenko, and J. Philbin, Jun. 2015] említést tesz több fajta dimenziószámmal készült mérésre, de abban a megvalósításban azért maradtak a 128 dimenziónál, mert az 512-es ha statisztikailag inszignifikánsan is, de rosszabbul teljesített, ezt a cikkben arra vezették vissza, hogy a nagyobb dimenzió szám miatt több időre van szükség a neurális hálózat taníttatásához, hogy hasonló eredményeket érjenek el.
Számunkra ez egy nagy kérdés volt, hiszen az új rendszer egyik nagy lehetősége lett volna a grafikus feldolgozó egységek által nyújtott vektorműveltekből adódó teljesítmény növekedés, de csak abban az esetben, ha az nem pontosság rovására érhető el.

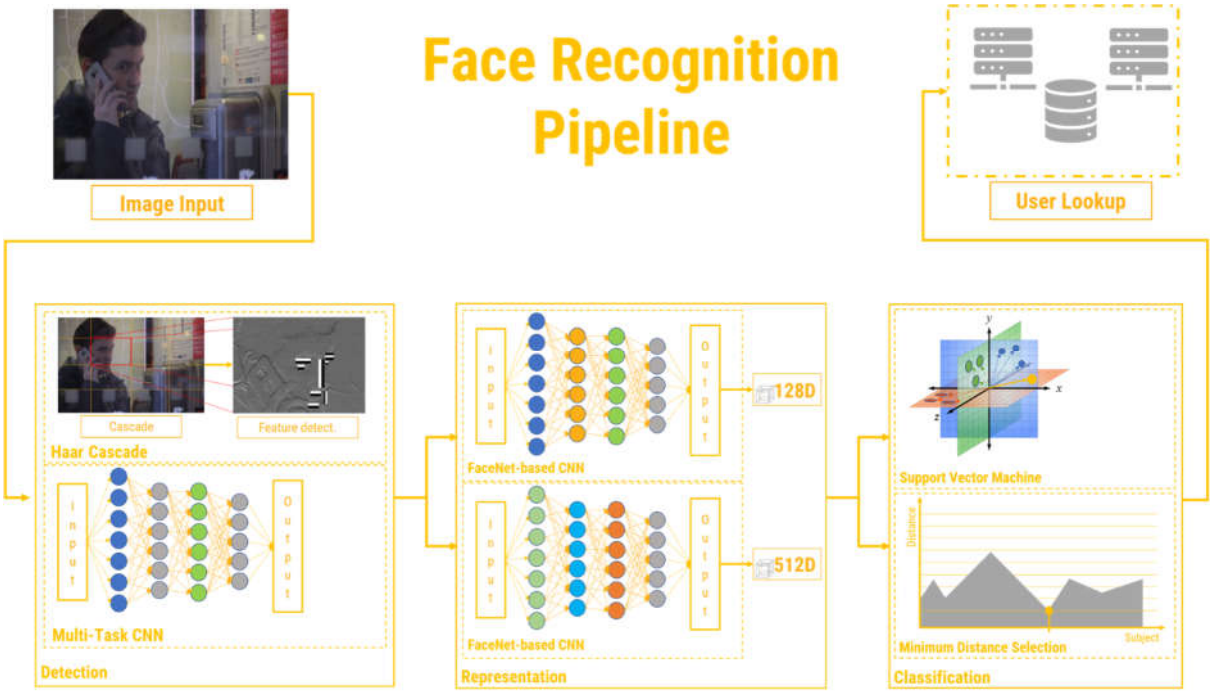
Célkitűzés

Célkitűzésünk összefoglalva a dolgozat a keretein belül a következők voltak:

- Egy okostükör működéséhez szükséges mögöttes szoftverarchitektúra megalkotása.
  - Az ehhez szükséges technológiák megkeresése.
- A technológiák használatához szükséges eszközök megkeresése és elsajátítása.
- Az általunk ismert arcfelismerési technológiák összehasonlítása és a számunkra megfelelő kiválasztása.
- Az összehasonlításhoz szükséges adathalmaz létrehozása.
  - Mérések elvégzése.
  - Mérések kiértékelése.
- A szoftverarchitektúrát alkotó elemek létrehozása és összehangolása.

Módszertan

Arcfelismerés



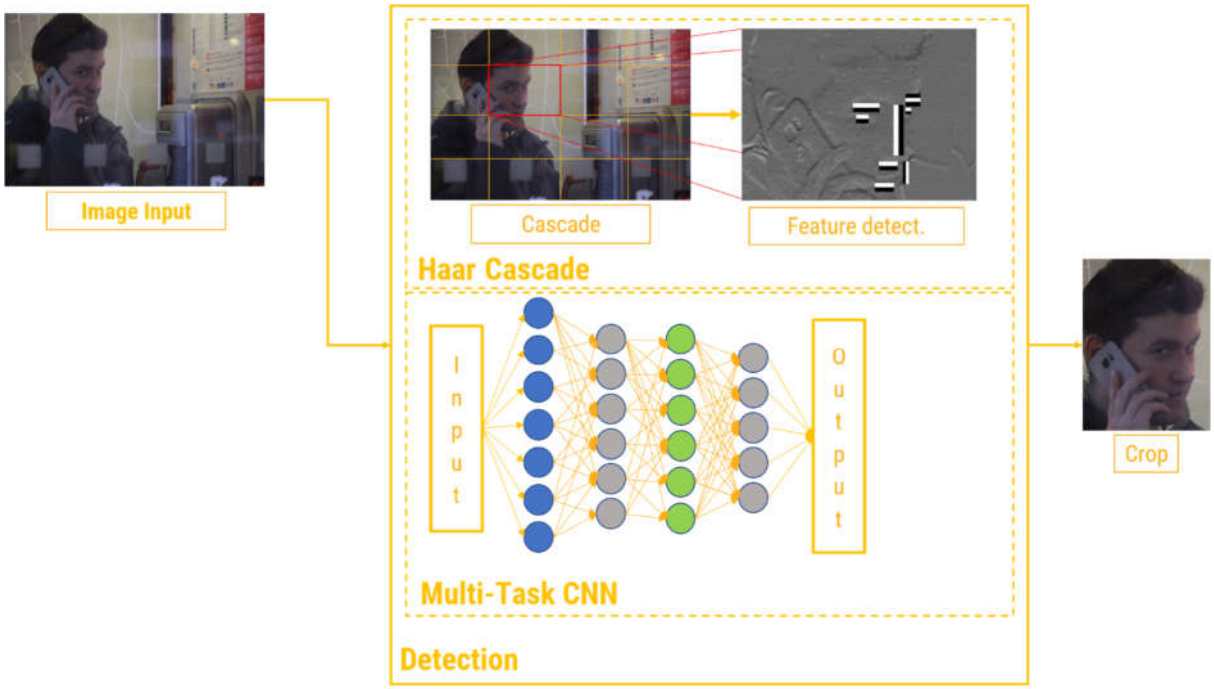
4. ábra Arcfelismerési folyamat futószalagja.

Az arcfelismerő rendszerünk három alkomponensre vagy rétegre bontható, melyek futószalag szerűen dolgoznak össze (4. ábra). Az első komponens feladata a *detection*, azaz, hogy található-e a bemeneti képen arc. Amennyiben találtunk arcot a képen, a *detection* réteg előkészíti az arcot a reprezentációs rétegnek, mely minden esetben egy mesterséges neurális hálózat, aminek a feladata, hogy a bemenetére kapott képet egy meghatározott dimenzió számú vektortérbe térképezze fel, amely egyértelműen meghatároz egy pontot, ami hozzáköthető egy-egy archoz. A végső komponens a klasszifikációs réteg, ami kiválasztja melyik felhasználó illik a legjobban a kapott vektorhoz.

Detection Réteg

Az új reprezentációs réteg implementálása folyamán frissítésre került a face detection réteg is. Az eredeti megoldásban egy viszonylag régebbi, de gyors futási idővel rendelkező algoritmust, a Haar Cascade-t (5.ábra) használtuk az arcok detektálására. A Haar Cascade [P. Viola, M. Jones, and M. Energy, 2004] egy előre meghatározott minta halmazt keres a bemeneti képen, egyszerű mintákkal kezdve, majd a találati mennyiség alapján egyre komplexebb mintákat

használva találja meg az arcot. Ezt egy neurális hálózat alapú modell (5. ábra), az MTCNN [K. Zhang, Z. Zhang, Z. Li, S. Member, Y. Qiao, and S. Member, 2016] váltotta fel, mely GPU gyorsítás felhasználásával összehasonlítható futási idő mellett több jelentős előnyt biztosít. Az MTCNN képes natívan kezelni az elforgatott arcokat, valamint képes párhuzamosan feldolgozni a bemeneti képeket. Továbbá az MTCNN detection réteg szorosan együtt működik a reprezentációs réteggel - a felfedezett arcokat előkészíti, így a reprezentációs réteg egyből egy tensor tömbbé alakított bementet kap, jelentősen felgyorsítva annak működését.



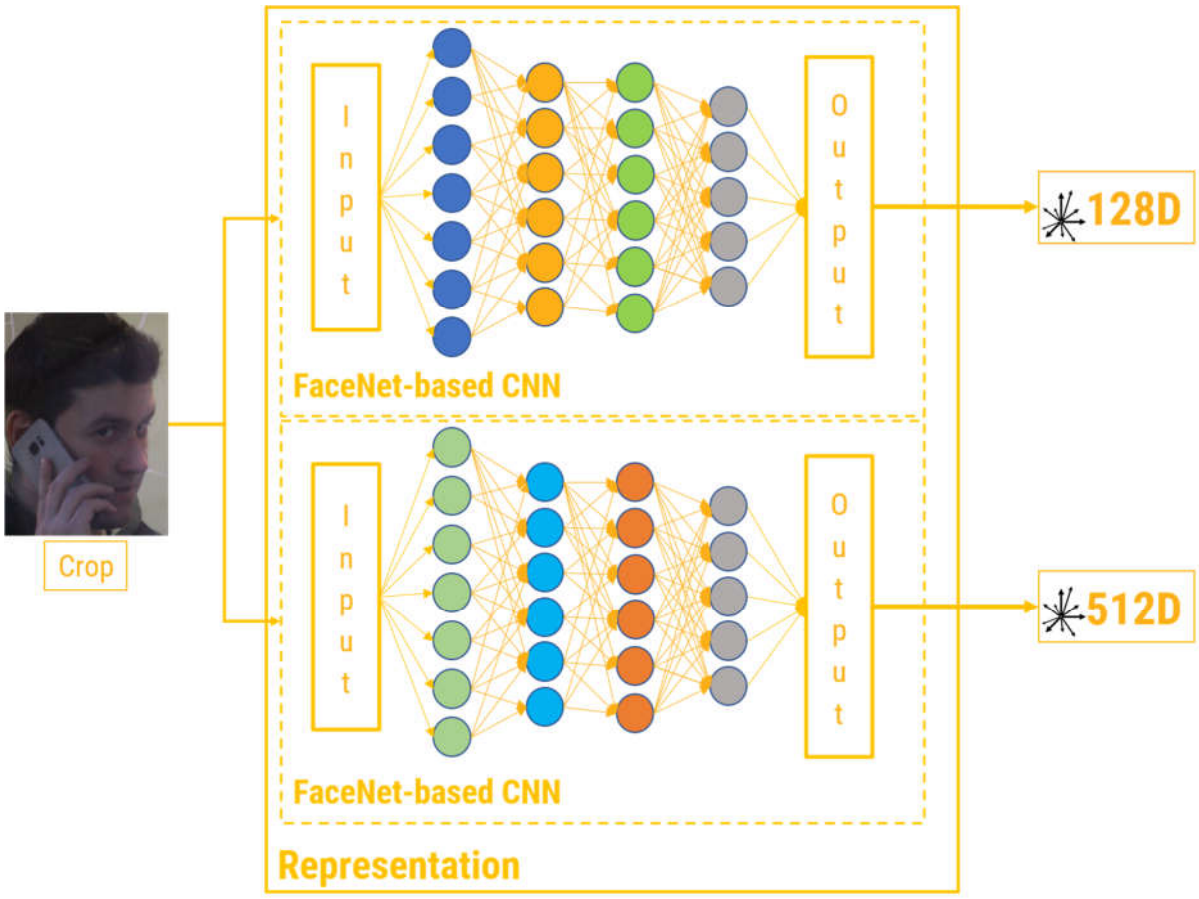
5. ábra: Az arc megtalálásáért felelős réteg működése. A különböző megoldások egymás alternatívái, nem egyszerre működnek.

Reprezentációs Réteg

Előző dolgozatunkban [A. T. Tóth, P. S. Tóth, and S. Mészáros, 2019] és cikkünkben [P. S. Tóth, A. T. Tóth, and S. Mészáros, 2019] bemutattunk egy, a fentebb említett elvek alapján működő megoldást, mely a reprezentációs rétegben 128 dimenzióval dolgozott (6. ábra). Ez a dimenzió szám jelentős redukciót jelent egy teljes archoz képest, azonban elég magas felbontású ahhoz, hogy egyértelműen azonosítani lehessen egy-egy arcot.

Azonban a technológia fejlődésével, többek között a GPU gyorsítást kínáló keretrendszerek elterjedésével lehetőség nyílik magasabb dimenzió számú eredmények kinyerésére hasonló futási idők megtartásával, így felmerül a pontosság növelésének lehetősége.

Ennek érdekében csapatunk által implementálásra került egy GPU gyorsítást alkalmazó reprezentációs réteg, mely 512 dimenziós vektortérben dolgozik (6. ábra).



6. ábra: A reprezentációs réteg működése. A felső neurális hálózat a 128 dimenziós, az alsó az 512 dimenziós implementációt jelzi. A különböző implementációk egymás alternatívái, nem egyszerre működik, csupán az összehasonlíthatóság miatt ábrázoljuk így.

Ezt a Pytorch [A. Paszke *et al.*, 2019] csomag keretein belül valósítottuk meg. A Pytorch a Facebook által fejlesztett Torch mesterséges intelligencia könyvtár Python programozási nyelvre optimalizált verziója, mely mára széles körben elterjedt a neurális hálózatokkal kapcsolatos kutatási területeken amiatt, hogy a Torch *tensor*-okból felépített neurális hálózatokat futás közben, avagy dinamikusan lehet változtatni.

Az eredeti reprezentációs rétegünk implementálása során még nem volt elérhető, de a fentebb említett, új reprezentációs réteg kidolgozására már a Pytorch CUDA verziója kilépett a béta fázisból.

A CUDA egy masszívan párhuzamos programozást lehetővé tevő hardver közeli nyelv, mely az Nvidia grafikus feldolgozó egységei által jelentős mértékben fel tud gyorsítani párhuzamosítható program kódokat. Habár a GPU gyorsításnak vannak hátrányai, melyekről később írni fogunk, a CUDA alapú futtatás jelentős mértékben fel tudja gyorsítani a neurális

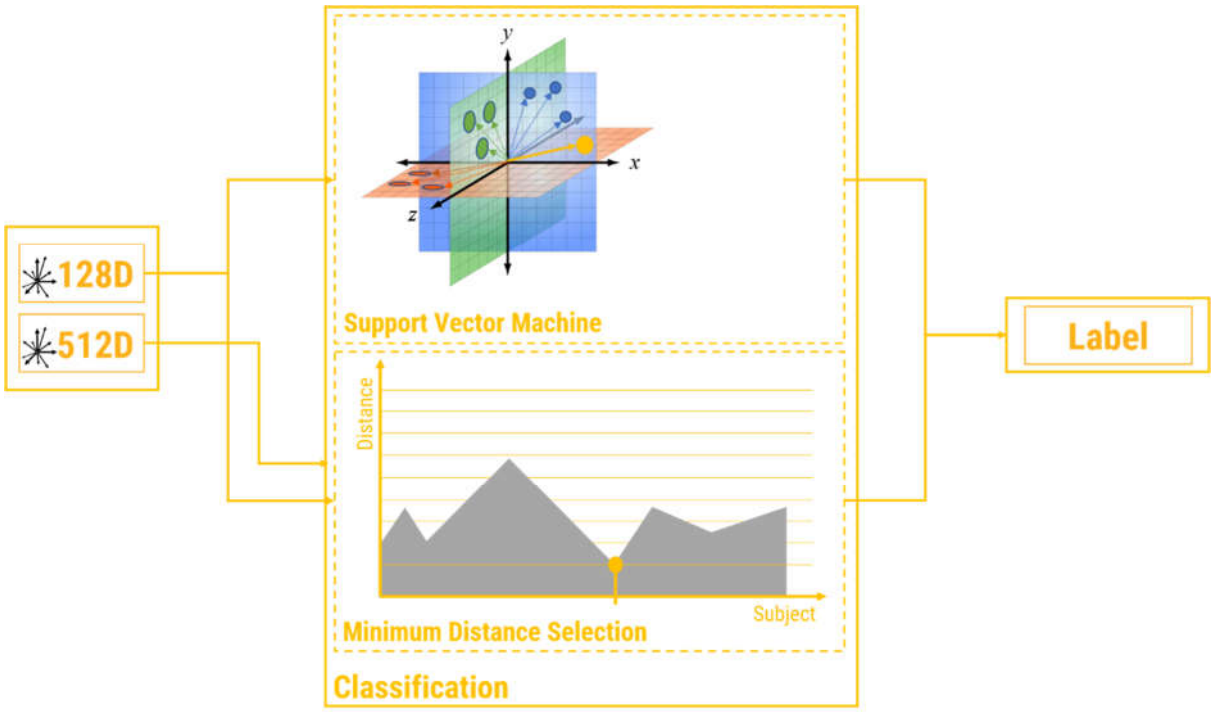
hálók használatát, mivel a Pytorch, ha a környezet képes rá, CUDA optimalizált *tensor* adatstruktúrákat használva mátrix műveletekre bontja a neurális háló működését [A. Paszke *et al.*, 2019], ami kézenfekvő egy grafikus feldolgozó egységnek.

A két implementáció teljesítménye alapos összehasonlításra került, mérésre került a futási idő, tárhely komplexitás, valamint a pontosság.

A két réteg összehasonlításra került batch feldolgozás és *single-image* feldolgozási idő szerint is. Közvetlenül a Python kódban mértük egy-egy utasítás teljesítési idejét, valamint, hogy mennyi ideig tart feldolgozni azt a 330 képet, melyet a kutatási kérdések megválaszolására használtuk.

Klasszifikációs Réteg

A munkánk során jelentős szakirodalmi kutatás következtében sikerült azonosítanunk az előző dolgozatban bemutatott arcfelismerő rendszer gyengeségeit, mely elsősorban az eredetileg implementált klasszifikációs réteg volt (7. ábra). Ez, az OpenFace a [M. S. Amos, Brandon, Bartosz Ludwiczuk] megoldás keretében egy *Support Vector Machine*-t jelentett, amely egy olyan *machine learning* megoldás a klasszifikálás feladatára, ami egy adott térben lévő vektorok elsődleges csoportosítása után egy újonnan érkező vektort megpróbál valamelyik csoporthoz rendelni. Ez alapvetően egy nagyon jó megoldás a klasszifikációra, azonban, mivel ez egy *machine learning model*, így a jó teljesítmény eléréséhez nagy mennyiségű adat, és tanítás szükséges [T. Mitchell, 1997]. Sajnos az a képmennyiség, amit a projekthez használtunk nem bizonyult elégségesnek, így úgy döntöttünk, hogy más oldalról közelítjük meg a klasszifikációt. Az eredeti Facenet [F. Schroff, D. Kalenichenko, and J. Philbin, 2015] publikációban a szerzők azt az érvet hozták fel a megoldásuk mellett, hogy két arc összehasonlítása az ő módszerükkel egy egyszerű Euklideszi távolság számításra vezethető vissza, ami alapján csapatunk implementált egy egyszerű “minimum távolság” klasszifikációs réteget, mely a referencia adatokból azt az egy vektorhoz tartozó címkét adja vissza, amihez az éppen vizsgált vektor a legközelebb van.



7. ábra: A klasszifikációs réteg működése. A klasszifikációs megoldások egymás alternatívái, nem egyszerre működnek. A Support Vector Machine csak 128 dimenziós vektorokat fogad.

Felhasználói Felület

Munkánk során lecserélésre került a tükör kezelőfelülete is. A platform függetlenség érdekében megváltunk a C# és WPF keretektől és egy multiplatform felületet kerestünk. Választásunk a Chromium Embedded Framework-öt [M. Greenblatt, 2008] megvalósító Eel [S. H. Williams, 2018] Python *package*-re esett. Ezzel a megoldással képesek vagyunk modern, web technológiák használatával megalkotni a felhasználói felületet, minimális időbefektetéssel. A CEF a Google Chrome böngészőt használja HTML, CSS és Javascript alapú tartalom megjelenítésre. Az Eel használatával képesek vagyunk a Python kódból egy web alapokon megírt felületet indítani, és irányítani, mely aszinkron módon hozzáférhet a Pythonban megírt metódusokhoz, valamint a Python kód is képes meghívni Javascript függvényeket. Ezen technológia használatával rengeteg nyílt forráskódú tartalmat fel tudunk használni a tükör különböző felhasználó-orientált funkcióinak ellátásához, mint például az időjárás információk lekérdezése és megjelenítése, vagy naptár és határidőnapló funkciók teljesítése.

Az eszköz tervezése során több kérdés is felmerült a felhasználói interakcióval kapcsolatban. Többek között a legfontosabb kérdés, milyen módon tudja felhasználó regisztrálni magát a tükör felhasználói közé? Zsigeri válasz lenne az érintő kijelző, azonban ezt az ötletet hamar elvetettük, egyrészt a fürdőszobai használat miatt, mivel a párásodás befolyásolja a kapacitív érintés érzékelést, valamint a nem túl esztétikus ujjlenyomat foltok elkerülése érdekében.

Felvetült az az ötlet, hogy gombokat helyezzünk el az eszközön, ám ez nehézkessé tette volna a szöveg bevitelt. Végül a választás egy külső kliens alkalmazására esett.

Abban az esetben, ha a tükör nem rendelkezik belső arc tárral, vagy ismeretlen felhasználóval találkozik, egy QR kódot jelenít meg a felhasználó felé, melyet telefonjával/tabletével beolvasva egy web felületre kerül, ahol regisztrálhatja magát, valamint személyre szabhatja az általa látni kívánt tartalmakat. Amint végzett a regisztrációval és személyre szabással, a web applikáció generál a felhasználó készülékén egy új QR kódot, amit a tükörnek megmutatva a tükör rögtön felismeri, és letölti az adatait az adatbázisból.
Abban az esetben, hogyha hibás felismerés történne, hasonló módon a felhasználó be tudja léptetni magát QR kód segítségével, ami a belépés után egy arc tár frissítést is eredményez, amikor a jelenleg a tükör előtt álló felhasználóról egy új referencia adatpont készül, mely növeli a felismerés bizonyosságát.

Normál használat esetén ez az arc tár frissítés sztochasztikusan is végbemehet, ha felismerés során készített képből generált vektor a legközelebb eső referencia ponttól 0,64 és 0,2 távolságra esik. Ezzel elérve azt, hogy huzamosabb használat során a rendszer egyre több adat pontot gyűjt be a felhasználóról, így minimalizálva a téves felismerés esélyét. Ezen határok úgy kerültek megállapításra, hogy a lehető legnagyobb mértékben növelje a tár frissítés a következő felismerések pontosságát. Emiatt a nagyon közel eső vektorokat nincs értelme elmenteni, hiszen nem nyújtana megfelelő mennyiségű új információt, valamint viszonylag távoli vektor mentése magával vonja a félre klasszifikálás veszélyét, ezzel felborítva a teljes felismerő rendszer működését. A sztochasztikus vezérlés annak az érdekében került bevezetésre, hogy arc tár ne legyen elárasztva hatalmas mennyiségű adattal.
Ezen arc tár frissítést egyelőre egy 0,33-as valószínűségi változó vezérli, melyet később szeretnénk egy normál eloszlásra cserélni annak érdekében, hogy csak a legnagyobb extra információs értékkel rendelkező új adatpontok kerüljenek mentésre.

Mindezek mellett, a tükörnek észlelnie kell, ha a felhasználó változtatást eszközölt a profiljában. Ennek érdekében implementálásra került egy szinkronizáló rendszer, mely az adatbázis egy tábláját előre meghatározott időközönként olvassa, és ha talál rekordot benne, akkor újból lekérdezi a rekordban található felhasználó adatait, majd törli a rekordot a táblából, valamint, ha új kép került csatolásra a felhasználóhoz, akkor automatikusan átadja az arcfelismerő rendszernek a képeket, mely elvégzi a vektorizálást, és lokálisan, lemezre menti

a kibővített referencia adathalmazt. Eredetileg a teszteléshez készült rendszer a felhasználói felület indításakor ellenőrzi, hogy van-e az előző futása óta új felhasználó vagy új kép, és abban az esetben, ha van, akkor az adott mappastruktúra alapján inicializálja a referencia adatokat, amiket lemezre is ír. Alapvetően egy *pickle* fájlba íródnak ki az adatok, mely egy BSON, binárisan szerializált objektum leíró. Azért esett a választás a binárisan kódolt formára, mivel a Torch *tensor*-ok alapvetően ezt a formátumot támogatják, és így jóval gyorsabb a szerializáció. Azonban később igény támadt a felhasználók *tensor*-ait adatbázisba mozgatni, így JSON formába is tudnia kell szerializálni a rendszernek. Erre van lehetőség, azonban a Torch *tensor*-okat Numpy *tensor*-rá kell alakítani, ami abban az esetben, ha GPU-gyorsított *tensor*-okkal dolgoztunk, azzal jár, hogy "le kell csatolni" a GPU memóriájáról a *tensor*-t. Ezt a Torch környezet {tensor objektum}.cpu().numpy() függvénye megteszi, és átmozgatja a CPU memóriájába, innentől egyszerű Numpy vektorként szerializálható JSON formátumba is.
Abban az esetben, ha létezik a lemezen szerializált adat, és annak az időbélyege frissebb, mint a legutoljára szinkronizált képé, abban az esetben a rendszer elkerüli a meglévő képek újra vektorizálását, és a szerializált adatokat tölti be a memóriába.
Mint ahogy fentebb említésre került, ez a rendszer a tesztelési szakaszban született, jelentősen felgyorsítva, és leegyszerűsítve a tesztelést végző személy munkáját, azonban miután a felhasználói adatok adatbázisba lettek áthelyezve, erre a rendszerre nem volt szükség, és beolvasztásra került a szinkronizáló rendszerbe.

Az arc adatbázis, azaz az „arc tár" elkészítése

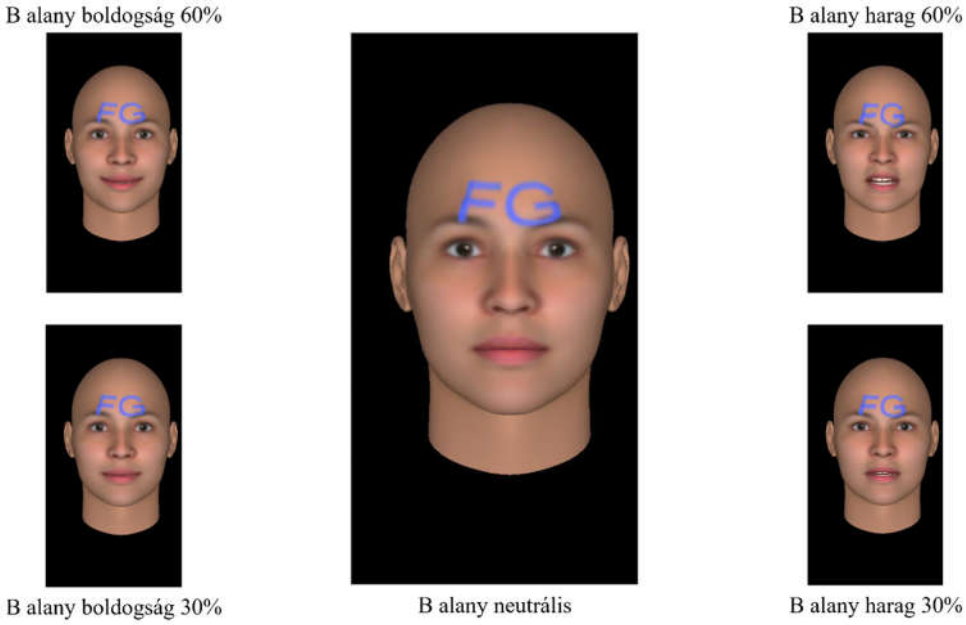
Az adatbázis elkészítéséhez a FaceGen Modeller demo [Singular Inversions Inc., 2020] verziója került felhasználásra. Ezek segítségével lett megalkotva az adatbázis. Az arcokat a véletlenszerű opció segítségével hoztuk létre. A programban ezután lehet állítani az úgynevezett Action Unitok-at [J. F. Cohn, Z. Ambadar, and P. Ekman, 2007][E. B. Roesch, L. Tamarit, L. Reveret, D. Grandjean, D. Sander, and K. R. Scherer, 2011], amik felelősek az arc egyes elemeinek mozgatásáért, mint például a mosoly. nagy segítség volt, hogy ezek a beállítások kimenthetőek a programból, így ezt nem kellett minden egyes arcnál újra százalékonként beállítani, egy egyszerű szövegszerkesztővel az XML dokumentumot elegendő volt módosítani.
Minden felhasználóhoz 2 érzelem garnitúra készült, egy a boldogságot és egy dühöt reprezentáló arckép csoport, az ehhez tartozó megfelelő Action Unitokat a Roesch et al [E. B.

Roesch, L. Tamarit, L. Reveret, D. Grandjean, D. Sander, and K. R. Scherer, 2011] cikke alapján választottuk ki. Három csoport került megalkotásra érzelmenként: a semleges, ez 0-10%-os intenzitás, az alacsony intenzitás, azaz 20-30% illetve a magas intenzitású csoport, ami 50-60%-ot jelent (8. ábra).

Az arcképek minden egyes érzelmi intenzitáshoz tartozó XML leiró beimportálása alapján a ShareX [ShareX Team, 2020] nevű programmal lettek kivágva, mivel ez a program a Windows API-t használva egyből feltudja ismerni a programban lévő ablak helyét és méretét, így csak a szükséges képet menti ki, nem volt szükség további körül vágásra.

A képeket minden felhasználó esetében azonos sorrendben kerültek elkészítésre, így a Bulk Rename Ultility [J. Willsher, 2019] használatával csak ki kellett választani a csoportokat, azaz a képeket 11-esével és a készítés időpontjával rendezve egy növekvő számot a program segítségével a fájl nevéhez fűzni, ami minden esetben az alany betű jele, az applikált érzelem és annak intenzitása volt egy alulvonással elválasztva.

Ennek a folyamatláncolatnak a végére megszületett az adatbázisunk, ami 5 alanyról összesen 330 képet tárolt.



8. ábra: B alany különböző érzelmekkel és intenzitásokkal.

A pontosság mérése

Miután a képek megszülettek és szétválogatásra kerültek, a semleges képek kerültek be az arcfelismerő rendszerbe referenciaként, míg az alacsony és magas érzelmi intenzitással ellátott képek kerültek ezekkel identifikációra a már említett képekhez generált vektorok segítségével,

pontosabban a két kép vektorainak távolságával.

Ennek a feladatnak az elvégzését két, Pythonban létrehozott script bonyolította le, a projekt arcfelismerésért felelős program ugyanis ebben íródott. Mindkét összehasonlítás esetén a referenciának és a kiértékeléshez használt felhasználó betűjele, az applikált érzelem betűjele, annak intenzitása és a két képet reprezentáló vektor távolsága exportálásra került egy *CSV* fájlba. Ezek későbbiekben még kiegészítésre kerültek azzal, hogy az adott távolság a 128, illetve az 512 dimenziós vektorok távolsága-e, illetve azzal, hogy megegyező alanyok, vagy pedig különböző alanyok között történt-e az összehasonlítás.

Fejlesztés

A dolgozat ezen részén bemutatásra kerülnek az általunk használt fontosabb programozási nyelvek, fejlesztő eszközök és Frameworkök (9. ábra), azaz keretrendszerek, amelyek egy újabb szoftver absztrakciós réteget nyújtanak, amellyel egy előre megírt tárházból tudjuk új képességekkel a kódunkat felruházni.

9. ábra: A felhasznált technológiák.

Program nyelv

A fejlesztés kezdetén az adatbáziskapcsolatot létesítő program még C# programnyelven került megírásra, ez még elegendő volt olyan esetekben amikor az okostükör egy laptop segítségével került bemutatásra, de egy céleszköz esetén egy ilyen feladatkörre nem feltétlenül jó választás egy teljes értékű számítógép és a Windows operációs rendszer financiális szempontból. A

Linux rendszerek emiatt előtérbe kerültek, hiszen ezek között sok olyan található, amelyek hobbi és kereskedelmi célra is ingyenesen felhasználhatóak.

A C# programozási nyelv a .NET Framework-re építkezik, amiben megtalálható a Common Language Runtime, ami egy virtuális gép a .NET alapú kódok végrehajtásához. Erre építve a Mono *framework* által lehetséges .NET Framework-öt alkalmazó programok Linux rendszerekre.

Sajnos a C# program által használt kiegészítő osztályok nagy része csak X86-os utasításkészletet támogató rendszereken képes futni, így a Windows kötöttségeket meg lehetett szüntetni, de az architektúrálisakat nem.

Emiatt lett bevezetve az *ORM* komponenst megvalósító szoftverrészhez Java programnyelv és az elképzelés, hogy azt egy alkalmazás szerveren keresztül érjük el, így a Pythonban megírt komponensek el tudják érni az adatokat, anélkül, hogy az adatbáziskapcsolattal kellene foglalkozniuk.

Dependency management

A *dependency management*, azaz függőségkezelés egy megoldás arra, hogy az adott projekt rendelkezzen minden *plugin*-nel, *framework*-kel, ami szükséges a projekt futásához, lefordításához a lehető legkevesebb beavatkozással a programozó részéről. Ezen felül elérhetővé teheti még a *plugin* forráskódját és dokumentációját is. Ez azon praktikát erősíti, hogy a szoftverek tervezésekor kisebb, egy feladatkörrel rendelkező komponensekből épüljenek fel, aminek jól definiált interfészeik vannak.

Ezek forráskód szinten működnek és projekt specifikusan, ezzel operációs rendszerek közötti váltás is megkönnyíthető.

A C# szoftverhez a Visual Studioban egyből elérhető a NuGet csomagkezelő, Python környezetben a Pip, a Java projektben pedig az InteliJ-be integrált Maven [Apache Software Foundation, 2020] lehetőség került kihasználásra.

Fejlesztőeszközök

Fejlesztőeszközök területén viszonylagos bőséggel találkozik manapság egy szoftverfejlesztő. A választást nem könnyítette meg, hogy ezek nagy része ingyenesen használható, vagy pedig egy egyszerű regisztráció után diákok számára ingyenes elérhetők.

A projekt esetében a fejlesztés nagy része a JetBrains termékeivel történt. Ez a cég annak érdekében, hogy a diákok a lehető legjobb eszközökkel tanuljanak egy felsőoktatási intézmény

által kiadott e-mail címmel való regisztráció után az összes fejlesztő eszközkét elkérhetővé teszi az adott illetőnek.

A Java webalkalmazás fejlesztésére az InteliJ IDEA Ultimate segítségével történt, ennek nagy fokú integráltsága lehetővé teszi, hogy a különböző technológiák összehangolása a projekt elkészítéséhez a lehető legegyszerűbben megtörténjenek. Ezeknek köszönhetően a Maven-en keresztül bonyolított függőségkezelés, a Hibernate [Jb. Inc., 2020] működéséhez szükséges *mapping file*-ok gyorsan pár mozdulattal elkészíthetőek.

A Java alkalmazás létrehozása során SonarLint [SonarSource S.A, 2020] nevű kiegészítő került felhasználásra, ami szintén az egész fejlesztőkörnyezetbe integrálódik. Működése tulajdonképpen nagyon hasonló egy helyesírás ellenőrzőhöz, az olyan kód részleteket, amik az általánosan elfogadottnak vélt, úgynevezett „jó gyakorlatnak" nem felelnek meg, a megszokott módon aláhúzással jelöli, hogy teendő van vele. Minden ilyen szabály esetén, ha a felmerülő problémát szemrevételezzük, találhatóak ajánlások, hogyan kellene módosítani a kódot, hogy biztosan a megfelelő működést eredményezhesse. Ennek segítségével több ezer felderített *bug* elkerülhető, méghozzá azonnal, hiszen a visszajelzés egyből elérhető.
Említésre méltó még a JetBrains reSharper nevezetű terméke, amely hasonló képeségekkel van felruházva, ez pedig a .Net keretrendszerhez érhető el, ezt a fejlesztés kezdeti szakaszaiban került felhasználásra a Visual Studio 2017-be integrálva.

Betűtípus a fejlesztő környezethez

Említésre méltó még a fejlesztés során használt betűtípus. Annak ellenére, hogy sokaknak ez apróságnak tűnhet, nem elhanyagolható a tény, hogy egy program fejlesztése közben sokat számít annak az olvashatósága, amit az ember éppen ír és olvas. A JetBrains Mono [JetBrains s.r.o.] egy úgynevezett fix szélességű betűtípus, ez azt jelenti, hogy karaktertől függetlenül mindegyiknek a szélesség megegyező. Ez sokat segíthet abban az esetben, ha a kódunk megfelelő behúzással van ellátva, nagy mértékben javítja az olvashatóságot.

Verziókezelés

Végső, de nem utolsó sorban a verziókezelés kerül bemutatásra. Ennek a szerepe az, hogy minden fájlban rögzíti a bekövetkező változásokat. Ennek segítségével problémamentesen visszalehet állni egy korábbi verzióra, ha az valami nem kívánt változást eszközölne.

Annak érdekében, hogy a csapat minden tagja bármikor elérhesse a munkájához szükséges forrás fájlokat létrehozásra került egy GitHub *organization*, ez kvázi a csapatunknak feletethető meg. Itt létrehozott *repository*-kban lehetőség van a fejlesztőeszközökbe épített, vagy külön telepített *command line* alapú alkalmazással a helyi számítógépen találat forráskódokat egy közösen elérhető felületre feltölteni. Ehhez szükséges volt minden projektnél a *gitignore* állomány megfelelő kitöltése, ezzel segítve azt, hogy egy állomány gond nélkül futtatható legyen, amint letöltésre került. Ennek módja, az adott projektnek a helyi Git kliensben vagy fejlesztő környezetben való inicializálása, majd a változtatások *commit*-tálása. A *commit*-okat véglegesíteni a távoli *repository*-ban a *push* lehetőséggel lehet, így, ezek után a változtatások már mások számára is elérhetőek voltak.
A verziókezelés segítségével felmerülő problémák esetén könnyen visszaállíthatók előző verziók az adott projektből.

Adatbázis

A projekt gerincének egy relációs adatbázison alapuló megoldást választottunk. Azért erre esett a választásunk, mert a csapat tagjai mind találkoztak és dolgoztak már ilyen technológiával, így a közös koncepciók használata megkönnyítette a szoftver komponensek összehangolását. A relációs adatbázisok tabuláris struktúrája teljesen megfelel a projekt igényeinek. Sok esetben problémás lehet a relációs adatbázisok esetén az, hogy ha a még nem kristályosodott ki a végleges struktúrája a rendszernek, ennek elkerülésére érdekében a rendszerben a felhasználók és programok paramétereit tároló táblák kulcs/érték elrendezésben lettek kialakítva, ezzel segítve a séma flexibilitását. Ennek ellenére a fejlesztés során került arra sor, hogy a sémában található táblák átalakításra kerüljenek, de ezek máshol okoztak nagyobb problémát, erről bővebben az *ORM*-el foglalkozó részben lehet majd olvasni.

Az adatbázis kiválasztása során a MySql adatbázis egyik *fork*-ja a MariaDB 10.3-as verziója került kiválasztásra. Ez a variáns teljesen kompatibilis a MySql 5.7-es verziójával, emellett nagyobb gyorsaságot képes elérni, a fejlesztés során így megmaradt az a lehetőségünk, hogy indokolt esetben visszaváltsunk a MySql rendszerre.

A fejlesztés elején egy több projektet futtató szerveren kapott helyet az adatbázis rendszer, ez később kiderült, hogy I/O konfliktusokat generált, ezért külön dedikált hardvert kapott a rendszer. A lehető legkisebb *bottleneck* elérése érdekében, egy SSD-re került a rendszer és az adatbázis, megfelelő mennyiségű memória társaságában és gigabites internet elérési sebességgel.

Az adatbázis szerver erőforrásainak kihasználtságára nem voltak előzetes becsléseink így olyan operációs rendszert választottunk hozzá, ami a lehető legtöbb erőforrást képes szabadon hagyni. Ez alapján Linux alapú rendszerek jöhetnek szóba. A választás végül a DietPi-ra [D. Knight] került a választás. Ezt alapvetően *Single Board Computer*-ekre találták ki, mint a Rapsberry Pi, de létezik belőle x86 utasításkészletre létrehozott verzió is. Ez a disztribúció a neve alapján is arra lett kitalálva, hogy a lehető legtöbb erőforrást szabadon hagyja egy *SBC* esetén, így még egy gyengébb processzorral rendelkező asztali számítógép esetén is elképesztően gyors rendszert kapunk.

A rendszer telepítése után már csak az adatbázis szoftver telepítése és konfigurálása maradt. A telepítés rendkívül egyszerű volt, hiszen a DietPi egy Debian alapú disztribúció így az ott megszokott Aptititude csomagkezelőn keresztül elérhető volt a MariaDB rendszer telepítője. A telepítés utáni teendők kimerülnek a felhasználók jelszavai és jogosultságaik beállításában, illetve a biztonsági beállítások módosításában, hogy a rendszer ne csak a *localhost*-ról legyen elérhető.

Annak érdekében, hogy a projekt minden tagja számára elérhető legyen az adatbázis az internet szolgáltató dinamikus IP cím kiosztása mellet szükség volt egy úgynevezett Dinamikus DNS szolgáltatóra volt szükség, ami a szolgáltató által kiosztott dinamikus IP címet egy konzisztens *domain* névhez köti. Ehhez a noip.com szolgáltató ingyenes megoldás lett kiválasztva, ennek mindössze annyi megkötése van, hogy 30 naponként újra kell aktiválni az általunk használt *domain*-t. Ezek után csak a használni kívánt portok továbbítását volt szükséges konfigurálni az internetet és az adatbázis szervet összekötő routeren.

Az adatbáziskezelő szoftverek közül a fejlesztés során az Oracle SQL Developer és a JetBrains által fejlesztett DataGrip volt alkalmazva. Az SQL Developerben elérhető a MySQL kompatibilitás, mivel az a termék is az Oracle portfoliójának része és ezért az összeköttetéshez szükséges szoftverkomponens könnyen beszerezhető. A fejlesztés során kiderült, hogy az SQL Developerben található lehetőségek a kínált MySQL csatlakoztató felülettel korántsem ideálisak, feltehetően azért, hogy az Oracle sarkallja a felhasználókat, hogy az Oracle adatbázisokra váltsanak. Emiatt lecserélésre került a program a JetBrinas DataGrip-jére, az ehhez tartozó licensz a JetBrains oldalán beszerezhető, a cég elképzelése szerint a hallgatóknak a lehető legjobb szoftvereket kell elérhetővé tenni a tanuláshoz, így egy oktatási intézmény által kiadott e-mail címmel való regisztráció után a cég teljes termékpalettájához hozzá lehet férni amíg a hallgatói jogviszony érvényes.

A DataGrip képes volt automatikusan detektálni a megadott IP és *port* mögött futó adatbázist és az alapján a megfelelő modulokat letöltve csatlakozni.

ORM

Az *ORM* technológiák a perzisztencia egyik lehetséges megvalósítása, azaz az adatok megőrzése azután, hogy az alkalmazásunk befejezte a működését.

Az *ORM*, azaz *Object-relational mapping* [G. Gregory, F. O. By, and L. Demichiel, 2006] egy olyan technológia, amely a relációs adatbázisok tábláit hagyományos objektum orientált programozásban használatos osztályokra vetíti le, olyan módon, hogy az adatbázis tábla egy sora egy objektumnak felel meg, a tábla oszlopai pedig az adott osztály tulajdonságainak. Egyetlen megkötés ebben a rendszerben, hogy az osztály minden tulajdonságának leképezhetőnek kell lennie olyan típusúra, amelyet támogat az adatbázis. Ezzel a módszerrel nagyon sok kód kézzel való megírása elkerülhető, mert a legtöbb *framework* képes kód generálásra. Két megközelítés létezik ennél a technológiánál: *code first* megközelítés, azaz a megírt kód alapján generál adatbázis táblákat a *framework*, illetve a *database first* megközelítés, ahol már meglévő adatbázis táblákból generál osztályokat a *framework*. A projekt ezt a megközelítés alkalmazza.

Nagy általánosságban ezek a *framework-ö*k a programozási nyelvek lehetőségeivel leképezik az adatbázis táblákban létrehozott kapcsolatokat (*one to many*, *many to one*, *many to many*) különböző *collection*-ok használatával, illetve a *constraint*-eket, amik beállításra kerültek (például *Set* – halmaz alkalmazása olyan esetben amikor a *unique constraint* megkövetelt.)

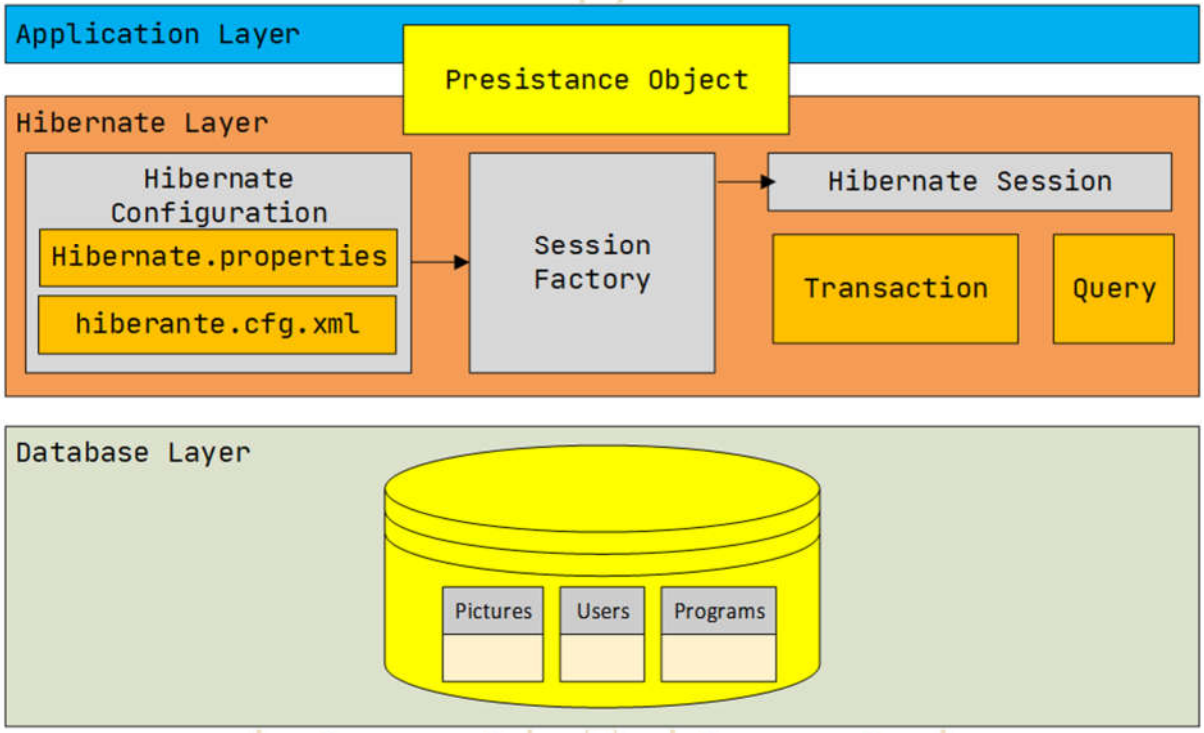
Entity Framework

A projekt kezdeti szakaszaiban a C# kód és az adatbázis összekötésére az Entity Framework *ORM framework* került felhasználásra. A kezdeti nehézségeket a *framework*-kel a megfelelő *plugin* kiválasztása volt az adatbázis kapcsolat létrehozásához. A *plugin* nevekben elhelyezett verziószám ebben az esetben azt jelölte, hogy a MySQL melyik verziójával volt kompatibilis, így a legfrisseb verzió nem volt megfelelő az általunk használt MariaDB adatbázishoz, ugyanis az csak az 5.7-es MySQL verzióval kompatibilis.

A kapcsolat létrehozása után a már meglévő adatbázisból már pár lépésben generált kódot a kiválasztott adatbázistáblákhoz. A fejlesztésnek ebben a szakaszában sok esetben minimális átalakításokat kellet alkalmazni az adatbázistáblák *constraint*-jein.

Emiatt többször szükséges volt újra generálni az adott osztályokhoz tartozó kódokat, ez pedig azzal járt, hogy az adott osztályhoz hozzáadott kiegészítő kódok elvesztek. Ennek elkerülése végett bevezetésre kerültek az adatbázistáblákat reprezentáló osztályokhoz a C# által támogatott *extension class*. Ezek képesek arra, hogy a bennük létrehozott függvényeknek megadhatunk egy osztályt argumentumként, a működésük pedig azonos lesz egy olyan statikus függvénnyel, mint amiket az adott osztályban lehet létrehozni. Ennek köszönhetően a generált kódok kiegészítései minden esetben megmaradtak, ha újra kellett az adott táblához tartozó osztályt generálni.

Hibernate



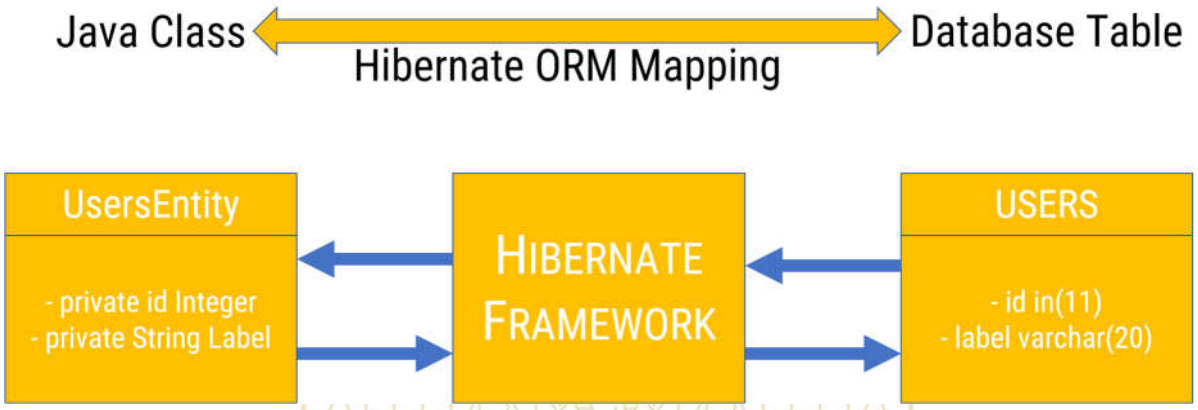
10. ábra: A Hibernate ORM framework felépítése.

A Java platformra való áttérés után új *ORM framework*re is szükség volt. A Hibernate 5.4 [Jboss Inc., 2020] (10. ábra) került kiválasztásra, az egyik legnagyobb múlttal rendelkező Java ORM Framework. A Java Persistance API-ra épített technológia így könnyedén használható Java Standard Edition és Enterprise Edition alkalmazásokkal.

Egyik nagy előnye ennek a *framework*-nek, hogy implementálásra került az úgynevezett *lazy loading* [G. Gregory, F. O. By, and L. Demichiel, 2006], amely azt jelenti, hogy az objektum inicializációja a lehető legutolsó pillanatban történik csak meg, addig csak az objektum egy *proxy*-ját, helyőrzőjét használjuk.

Ennél a *framework*-nél szintén a *database first* megközelítés került használatra, hiszen a Java nyelvre váltás idősávjában az adatbázis séma többé-kevésbé már véglegesítve volt.

A Hibernate *framework* legtöbb beállítása is legenerálásra került az osztályok létrehozásakor, ezt az InteliJ Ultimate Persistance modulja végezte el, miután a megfelelő IP-t, *port*-ot és a belépéshez szükséges adatokat megkapta. Mivel a DataGrip és az InteliJ ugyanannak a fejlesztő csapatnak a terméke az adatbáziskapcsolat létrehozása azonos könnyedséggel ment végbe.



11. ábra: Egyszerű példa egy Java osztály és adatbázis tábla megfeleltetése.

Minden egyes osztály létrehozásánál generálásra került egy úgynevezett *mapping file* az adott osztályhoz, amiben megtalálható, hogy az osztály mely tulajdonsága melyik oszlopnak feletethető meg (11. ábra). Ezeket annotációval is meg lehet adni az osztály forráskódjában, de ha a különálló fájl kerül felhasználásra, az a forráskód minden módosítása után sértetlen marad. A generált *mapping file*-ok esetében csak minimális módosításokra volt szükség, annak érdekében, hogy az adatbázisban beállított *auto increment* kulcsok és az idegen kulcsos összekapcsolások működőképessé váljanak.

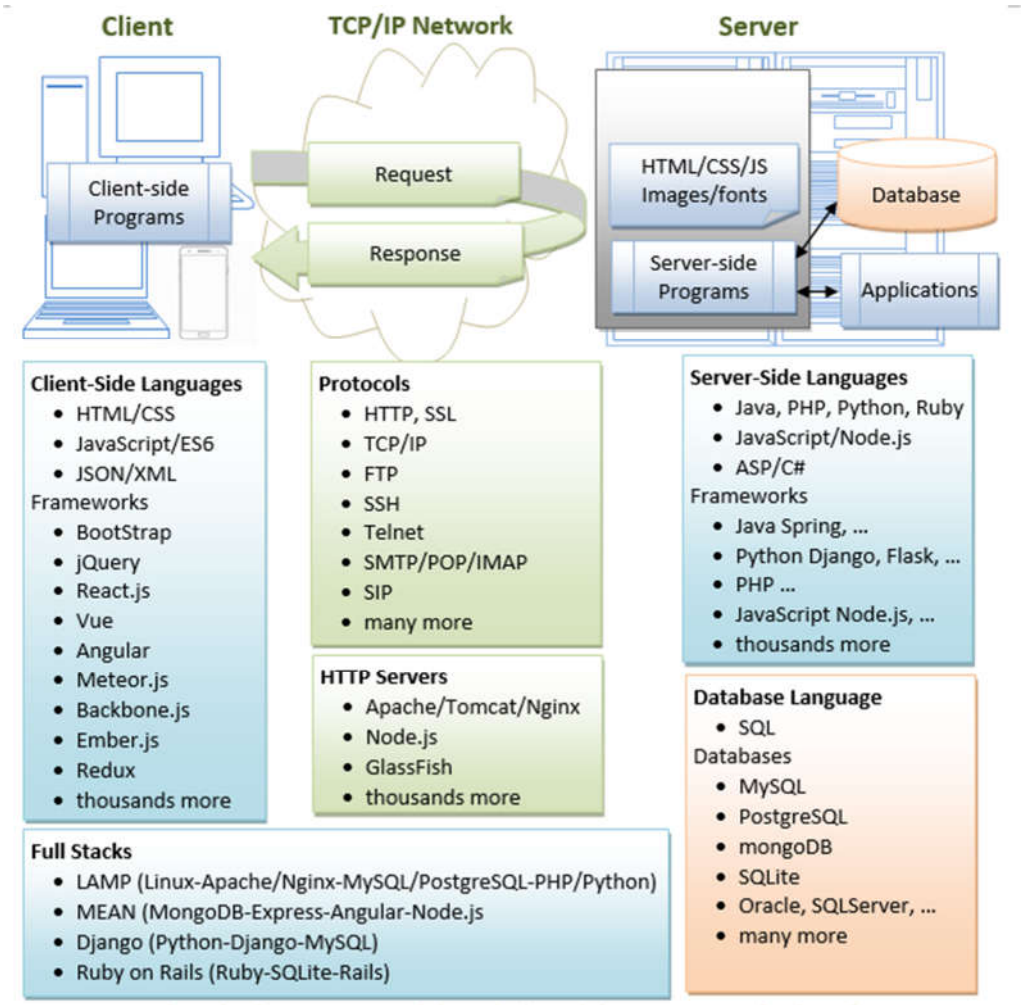
JSON

A szerializáció egy folyamat mely során a meglévő objektumainkat más reprezentációs formába alakítjuk, bináris, vagy valamilyen strukturált formában JSON, vagy XML formába öntjük. Erre azért van szükség, hogy az adatbázisban található adatok könnyedén elérhetőek legyenek a webserveren keresztül, anélkül, hogy közvetlenül az adatbázishoz kelljen a kliensnek csatlakoznia.

Az objektumok szerializációjához a Google által fejlesztett GSON *plugin* lett kiválasztva. Ez a *plugin* JSON formátumba képes alakítani az objektumokat, illetve JSON formátumban tárolt objektumokat képes deszerializálni. A használata úgy lett kialakítva, hogy olyan egyszerűen lehessen használni, mintha egy objektum *toString()* metódusát hívnánk meg.

Támogatja azt is, hogy bizonyos tulajdonságokat maszkoljunk előle, így azok a szerializációkor nem kerülnek bele a végeredménybe. Nagy szükség volt arra is, hogy megfelelő módon ki tudja nyerni az objektumban található *collection*-ökben tárolt adatokat és azokat a JSON eredményben megfelelően reprezentálja.

Apache Tomcat



12. ábra: Egy szerver és kliens közötti kapcsolat HTTP protokollal.[30]

A projekt alkalmazás szerver részéhez az Apache Tomcat [Apache Software Foundation, 2020] került kiválasztásra, ez egy nyílt forráskódú alkalmazás szerver, amely implementálja a Java Servlet, Java Server Pages és WebSocket technológiákat, ezek segítségével egy HTTP webszerver környezetet nyújt, ahol Java kód futtatható. A projekt esetében a 8.5-ös verzió került felhasználásra. Az egyszerűség kedvéért a webszerver ugyanazon a gépen fut, ahol az adatbázis is be van üzemelve, de természetesen a kettő két külön gépen is helyet foglalhatna. A működése annyiból áll, hogy az adott webcím mögött elérhető szervernek a kliensek képesek

egy HTTP *request*-et (12. ábra) küldeni, ami arra egy HTTP *response*-zal (12. ábra) válaszol, a kért tartalmat a *response*-ba ágyazva, ehhez Java Servlet-ek kerültek felhasználásra.

A Java Servlet-ek a REST, azaz a REpresentational State Transfer szoftver architektúrát felhasználva valósítják meg a web szolgáltatást. Ezeket *RESTful* web szolgáltatásnak nevezik, ezek segítségével számítógépek és más eszközök képesek a helyi hálózaton, vagy az interneten keresztül együttműködni. A *request*-et intéző készülék ezeken keresztül képes elérni és manipulálni erőforrások szöveges leképezéseit előre meghatározott műveletek segítségével. Az elérést lehetővé tévő protokoll, úgynevezett *stateless* protokoll, ebben az esetben a HyperText Transfer Protocol. A *stateless* protokoll kialakításukban egyszerűek, a kliens egy *request*-et indít a szerver felé és az arra egy *response*-t ad vissza. A *stateless* protokoll használatával egyszerűbb szerver kialakításra van lehetőség, hiszen kevesebb erőforrásra van szükség ,valamint nincsen szükség munkamenet kezelésre, illetve minden csomag egymástól függetlenül közlekedik. Ezáltal minden kommunikáció úgy alakítható ki, hogy a kliens küld egy *request*-et a szerver felé, úgy, hogy abban a megfelelő paraméterek legyenek benne, arra pedig a szerver egy előzetesen megegyezett formában válaszol a *response* segítségével. A *response*-ban lehet beágyazva például egy JSON állomány, vagy egy kép. A szerver minden *servlet*-e más-más feladatokat láthat el, ezek mindegyike külön *URL*-en, vagyis végponton érhető el.

Adminisztrációs oldal fejlesztése

Kezdetekben a regisztráció és felhasználók kezelése manuálisan történt azon az eszközön, ahol a tükör szoftvere futott. Ez lassú és kevésbé kifinomult volt. Ahogy a komponensek egyre bonyolultabbak lettek, a felhasználó kezelésnek fejlődnie kellett.
Így elkészítésre került egy felhasználói weboldal, ahol regisztrálni és kezelni tudja a felhasználó a tükör beállításait. Eleinte a felhasználók egy helyi JSON állományban voltak tárolva, ez lett lecserélve egy távolról elérhető, központi adatbázisra.

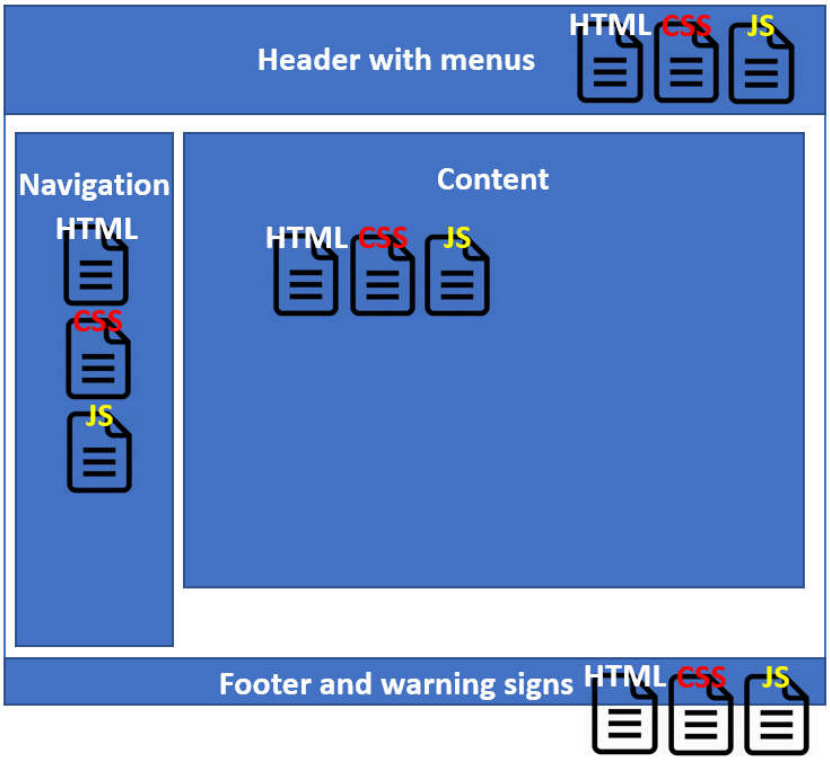
Az egyszerűség kedvéért különböző lehetőségek közül a Vue.js-re [E. You, 2014] és az Express.js [T. Holowaychuk and StrongLoop, 2010]-re esett a választás.

Frontend és Backend

Manapság elterjedt tervezési paradigma a kliens-szerver architektúra, aminek a megfelelője a web applikációk területén a *frontend*, és a *backend*.

A funkciók szeparálása több előnnyel is jár: könnyebb a kód karbantartása és programozása, kisebb a kód komplexitása, könnyebben átlátható végeredményt kapunk. Továbbá lehetőségünk van számításigényesebb feladatokat a kliens oldalán futtatni, így csökkentve a szerver terheltségét, valamint, ha felosztott kliens és szerver applikációval rendelkezünk, bármi hiba esetén nem az egész alkalmazás lesz elérhetetlen, hanem csak egy része.

Keretrendszerek összehasonlítása



13. ábra: Egy modern weboldal felépítése.

Az Angular [Google, 2016] egy régóta létező keretrendszer, melyet mai napig támogatnak a fejlesztői. Nem annyira kézenfekvő az elsajátítása, mint a Vue.js keretrendszeré. Angular-t a Google és a Wix használ.

Másik lehetőség a React lenne, ez szintén egy érettnek mondható keretrendszer, mai napig támogatott. Manapság ez egy keresett keretrendszer és ajánlott kezdőknek is. Facebook és Uber által is használt.

Vue még nem annyira elterjedt, hiszen ez egy újabb keretrendszer, de erős versenytárs lett a többi alternatívához képest, köszönhetően a nagy kínai cégeknek, mint az Alibaba és Baidu.

Vue.js

A Vue.js egy nagyon friss és könnyen elsajátítható, progresszív frontend keret, amivel szép és egyszerű felhasználó kezelést tudunk létrehozni.

A weboldalak egyre dinamikusabbak és egyre több lehetőséget kínálnak, részben a Javascriptnek köszönhetően. Ez azt jelenti, hogy rengeteg tevékenységi kör áthelyezésre került a kliensoldalra, Javascriptben megvalósítva, ennek következtében több ezer sor kód került a weboldalak HTML állományaiba rendszerezetlenül A káosz megszüntetése érdekében egyre több fejlesztő fordul különböző Javascript *framework*-ökhöz mint az Angular, React vagy a Vue.js.

A Vue.js egy ingyenes, sokoldalú keretrendszer, aminek a segítségével könnyű egy karbantartható és tesztelhető weboldalt létrehozni.

Könyvtárai segítségével komplett web applikációkat is könnyedén létre lehet hozni. Ilyen könyvtárak többek között a Vue.js Core, Vuex, Vue router.

Vue egy komponensekre (13. ábra) szétbontható weboldalt hoz létre, ahol mindennek megvan a saját HTML, CSS és Javascript kódja, illetve egy komponens többször is felhasználható.

Egyik egyszerű, de nagyszerű lehetőség ebben a keretrendszerben az, hogy a weblapon megjelenített változók értékük változása esetén az oldal frissítése nélkül aktualizálódnak. A regisztrációs és adminisztrációs oldal frontendje Vue.js-ben került elkészítésre, ez API hívásokkal kommunikál a háttérszolgáltatásokkal.

Express.js

A Express.js egy Javascripten alapuló backend környezet, amivel az adatbázist és a felhasználókat kezeljük, a frontend API hívásokkal éri el, így csak a weboldal felől érkező kéréseknek tesz eleget.

Az adatok első szinten adatellenőrzésen esnek keresztül, hogy biztosítva legyen az adat formailag, tartalmilag megfelelő módon érkezzen be a szerverhez.

Egy API hívás a felhasználó által, a frontenden megnyomott gombbal kezdődik, ilyenkor létrejön egy API *endpoint* hívás a backend felé például: https://localhost:8081/login, a backend értelmezi a bejövő adatokat és annak megfelelően reagál. Az adatok beérkezése egy JSON állományon keresztül történik.

Az API, azaz Application Programming Interface, sűrűn használt megoldás. Ez definiálja hogyan tudja a programozó vagy egy másik rendszer használni azt a szoftvert, meghatározza, hogy milyen kéréseket, hívásokat, adatokat lehet intézni a szerver felé.

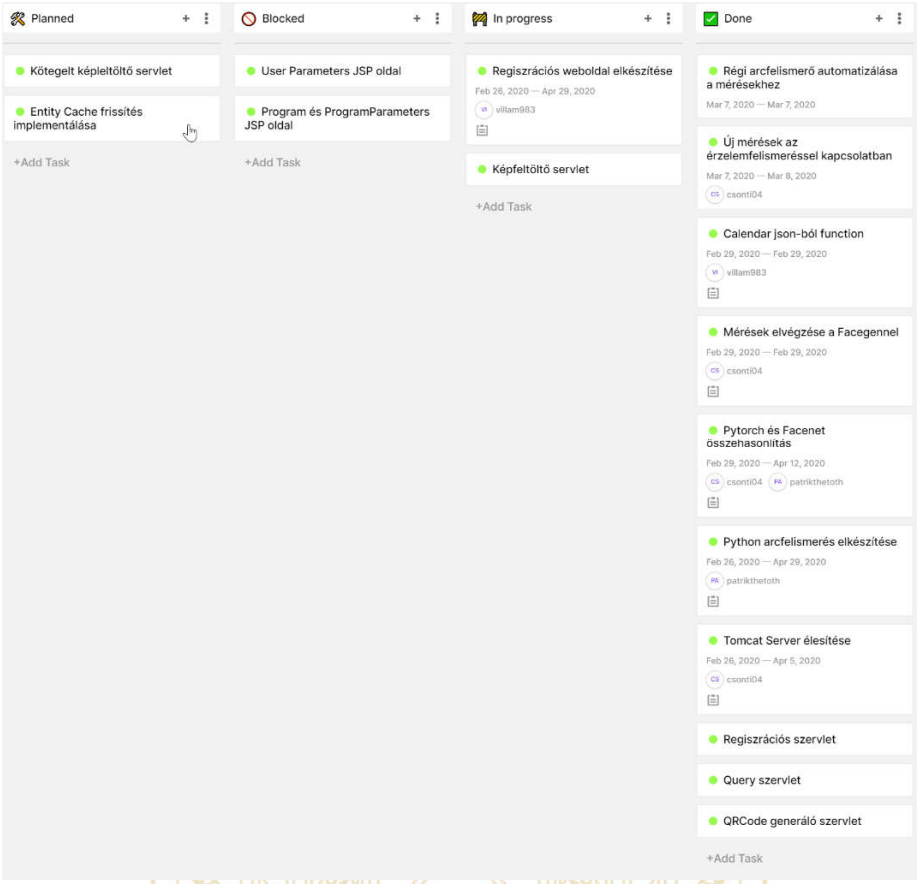
Adatbázis kezelés, Sequelize

A backend az adatbázissal egy Sequelize nevű *framework* használatával kommunikál.

A Sequelize egy Obejct Relation Mapping megoldás Node.js-hez, ami a Postgres, MySQL, MariaDB, SQLite és Microsoft SQL adatbázisokat támogatja. A Sequelize támogatja a Node.js aszinkron feldolgozást lehetővé tevő tulajdonságát, a *promise*-okat, ezzel eliminálva a várakozást lassú kiszolgálók esetén.

Projekt management

A projekt összetettsége és a limitált idő miatt szükséges volt az elvégzendő feladatok nyomon követése, illetve a pillanatnyi céljaink rögzítése valamilyen módon. Erre a sok lehetőség közül a Toggl Plan került kiválasztásra, azért, mert itt lehetőség van a feladatok Gantt diagram és Kanban tábla szerű követésére.
A Gantt diagram egy ütemterv, ami idősávokban ábrázolja, hogy az adott projekt részfeladatai hogyan állnak. Segítésével könnyen átláthatóak a projekt elemei, hiszen a vízszintes tengelyen az idő kerül ábrázolásra, a másikon pedig az elvégzendő, egymásra épülő feladatok láthatóak. Itt minden feladathoz lehetséges választani a projekt tagjai között, akikhez tartozik az adott feladat, megadni, hogy napi hány órát foglalkozzon az adott feladattal, így jól látható képet kaphatunk arról, hogy ki mennyire van leterhelve a fejlesztés során.



14. ábra: Project management során használt Kanban tábla.

A Kanban tábla (14. ábra) egy olyan reprezentációját adja a feladatoknak, ahol az adott teendő a bal oldalon belép a táblára és minden előre haladással a jobb oldalra halad. Az oszlopok a különböző készültségi fokozatokat reprezentálják.

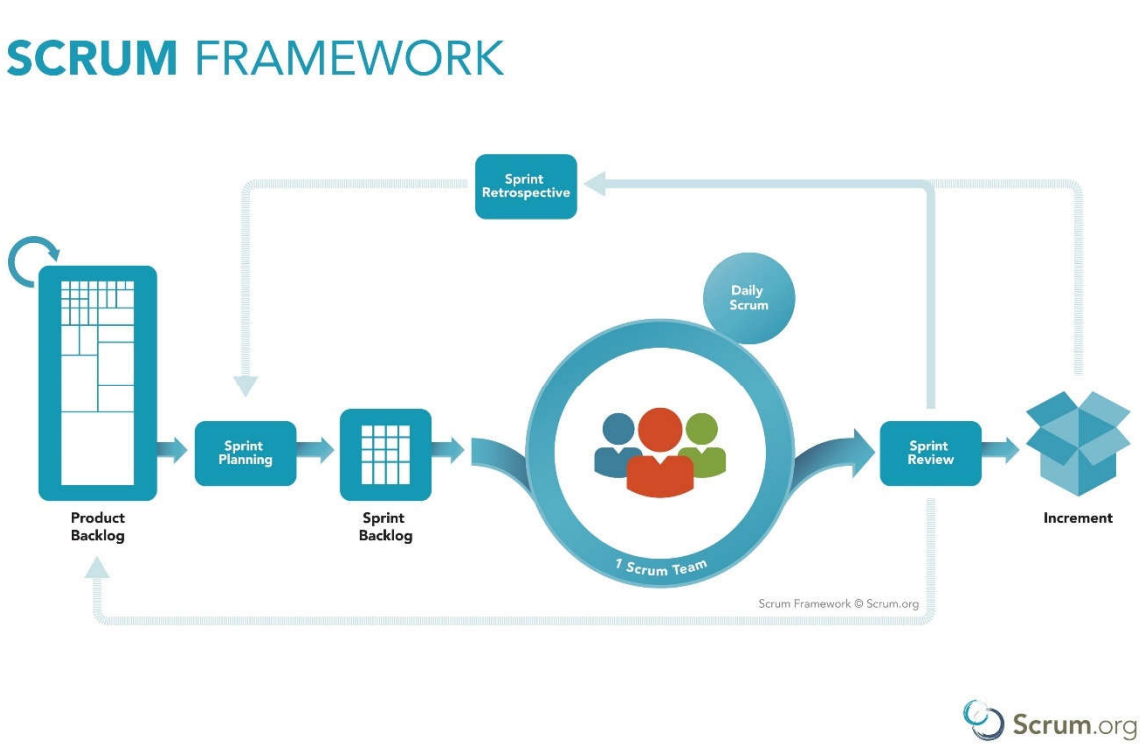
A Toggl Plan rendszerében a Gantt diagrammon felvett feladatok automatikusan megjelennek a Kanban táblán is, a kettő között könnyedén lehet váltani.

Scrum Metodológia [K. Schwaber, 2010]

A projektet Scrum keretrendszerben került fejlesztésre (15. ábra). Ennek a dolgozatnak az írói alkották a Scrum csapat tagjait, míg konzulensünk a Scrum Master és Product Owner szerepeket vállalta magára. A módszertan lényege az, hogy a fejlesztés úgynevezett sprintekre van felosztva, ami egy olyan időintervallumot jelent, mely alatt a projekt termékének egy új változatát, azaz *increment*-jét elő lehet állítani. Az *increment* a *product* backlogban található elvégzendő feladatokból az adott *sprint*-re kiválasztott és elkészített feladatokat jelenti. A *sprint*-eket egy hónapnál nem hosszabb időszaknak szokták meghatározni, a projekt esetében minden tagnak az egyetemmel és munkával kapcsolatos elfoglaltságai miatt jóval kisebb időintervallumra, 1 hétre kellett szabni.

A *sprint* tervezés alatt a *product blacklog*-ból kiválasztásra kerülnek azon feladatok, amelyeket az adott *sprint* alatt a csapat el kíván végezni, ezek a *sprint backlog*-ba kerülnek bele. A *daily scrum* egy olyan esemény, amely a következő napra segít szinkronba hozni a fejlesztők munkáit, a projekt esetében ez csak heti két alkalommal került megszervezésre. A *sprint review* a *sprint* végén eszközölt lépés, amikor a *product owner* és a Scrum csapat megvizsgálásra kerül, hogy a *product backlog*-ból mi került megvalósításra és a következő *sprint*-ben mit lenne szükséges megvalósítani.

Sprint retrospective lehetőséget ad arra, hogy a fejlesztő csapat tagjai reflektáljanak az előző *sprint* eseményeire, fejleményeire és ezáltal terveket hozhassanak létre a végrehajtandó fejlesztésekkel kapcsolatban.



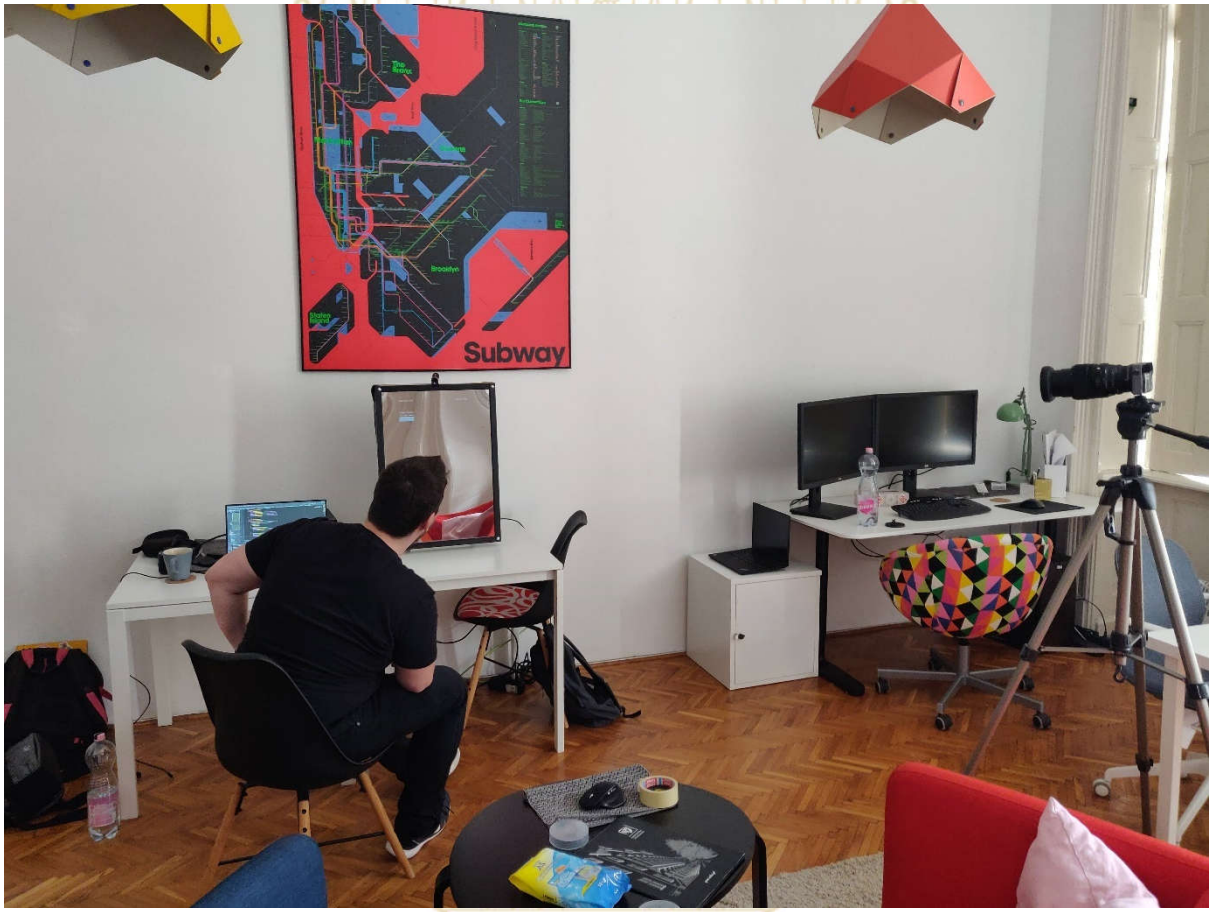
15. ábra: Scrum metodológia folyamatábrája [K. Schwaber, 2010]

Coworking

Annak érdekében, hogy a projekt a megfelelő ütemben haladjon a fejlesztők közül ketten elhatározták, hogy hetente egy egész munkanapot felhasználva a szombatokat a projekt előre gördítésére használják fel. Annak érdekében, hogy ezek az alkalmak a lehető legproduktívabban teljenek, megfelelő helyszínre volt hozzá szükség.

Az Óbudai Egyetem Alba Regia Műszaki Kara tudott számunkra szombatonként termet biztosítani, a hétvégi oktatásokból kifolyólag. Azokra a szombati napokra amikor az intézmény zárva volt pedig közösségi irodákban kerestünk alternatívákat.

Sajnos a jelenleg is zajló járványügyi helyzet nem tette lehetővé, hogy a munkát az Egyetem területén folytassuk, de a székesfehérvári Konnektor Közösségi iroda (16. ábra) lehetőséget adott arra, hogy ott helyet bérelve tovább folytassuk munkánkat hétvégenként. Ezek az alkalmak nagyban hozzájárultak ahhoz, hogy a projekt a mostani állapotát elérje, hiszen egy kvázi munkahelyként volt lehetőségünk tekinteni az irodára, ezzel is segítve, hogy a feladatunkra koncentrálhassunk.



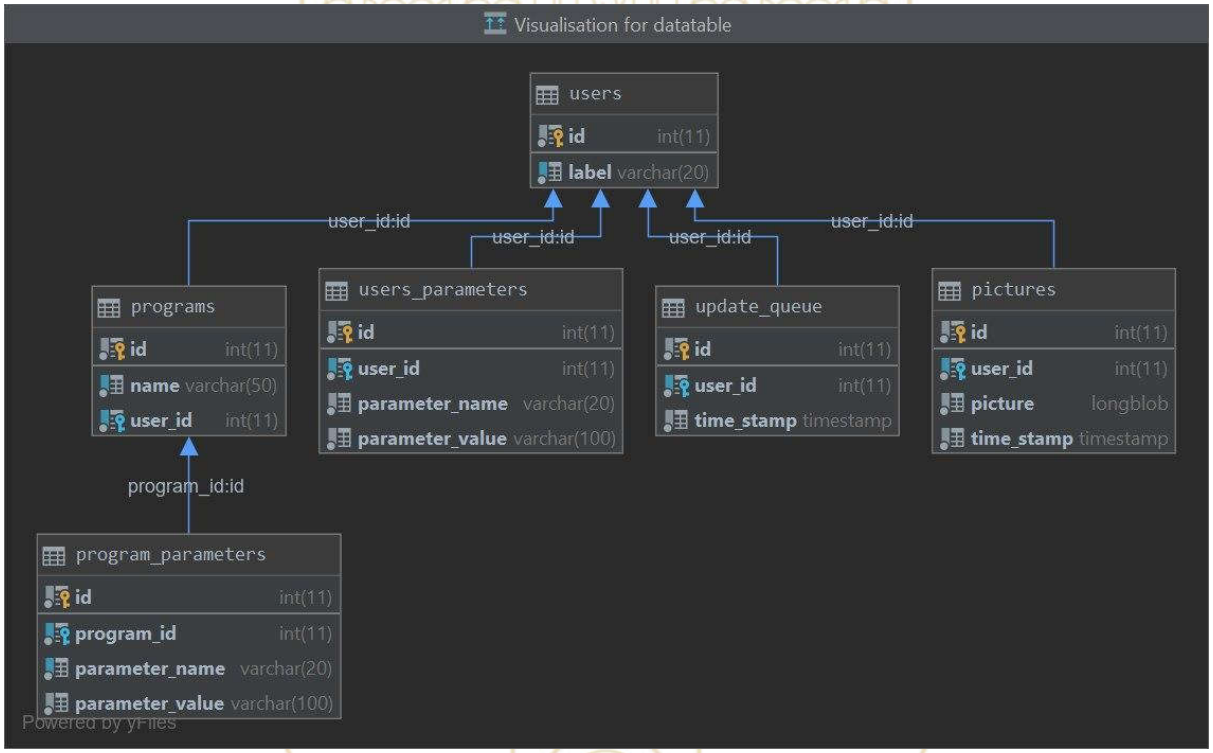
16. ábra: Konnektor közösségi iroda.

Eredmények és diszkusszió

Adatbázis

Elkészítésre került a MariaDb adatbázis (17. ábra), melyben a központosított adattárolás lett megvalósítva. A felhasználókat egy *Label*, azaz címke azonosítja, amely a regisztrációnál használt email címe a felhasználónak, amit egy *hash* függvény segítségével átalakítottunk. Ez egy teljesen egyedi, de mégis anonim azonosítója lett így a felhasználóknak.

Ezekhez a rekordokhoz idegen kulcsok segítségével vannak hozzákapcsolva a felhasználói paraméterek, a programok és a képeket reprezentáló táblák. A programs táblához tartozó program_parameters tábla hasonló módon, idegen kulcsokkal kapcsolja össze a felhasználók programjait és a felhasználó programjához tartozó paramétereket. A létrehozott táblák esetén automatikusan inkrementáló kulcsokat és a beépített időbélyegfunkciókat használtunk, hogy ezeket ne kelljen a programnak felügyelnie.



17. ábra: Az adatbázis séma felépítése.

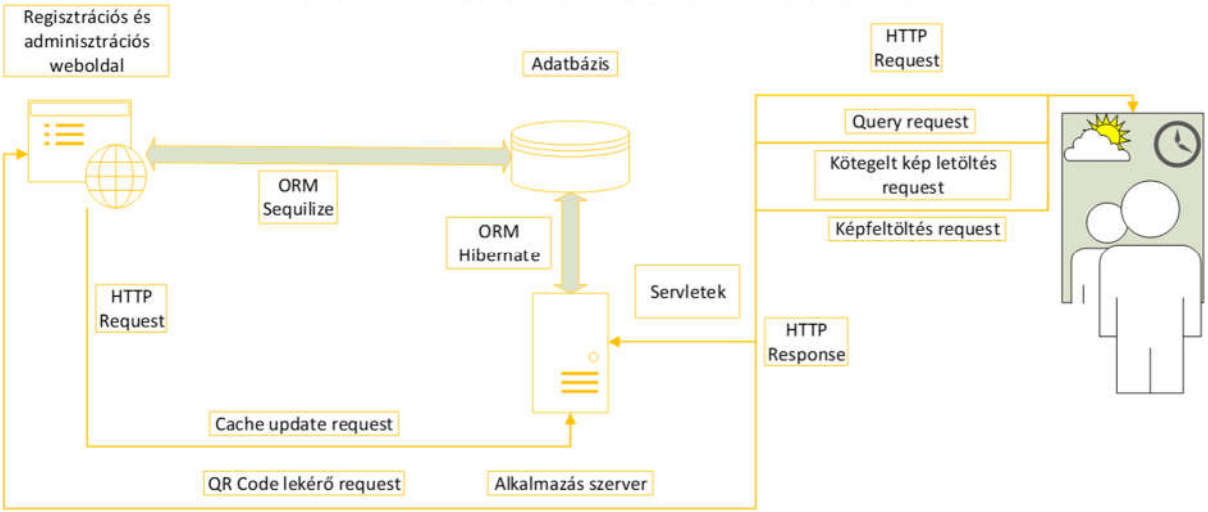
Java webszerver

Létrehozásra került a szerver alkalmazás, amely a Hibernate [Jboss Inc., 2020] keretrendszer segítségével csatlakozik a webszerverhez és a kliensről levéve a terhet közvetíti annak a kliens által lekért adatokat a Java Servletekkel kialakított *RESTful* architektúrával.

Ennek módja az, hogy attól függően, hogy a kliensnek milyen adatra van szüksége, az annak megfelelő Servlet-het tartozó végpontra küld egy *request*-et és az arra adott *response*-ban lévő adatot feldolgozza (18. ábra).

A szerver applikáció eredetileg a sok adatbázis tranzakció miatt, ami arra szolgált, hogy az adott személyhez tartozó információkat visszaadja, nagyban növelte a válaszidőt, ezért bevezetésre került egy *hash map*-pel kialakított UserEntity cache, ami nagyban gyorsította a szerver működését.

Ennek egyetlen hátránya az volt, hogy a kimentett elemek egy időpillanatot mutattak, így, ha a weboldalon valaki módosítja az adatait itt nem fog megjelenni. Így bevezetésre került egy megoldás a szerver UserEntity cache koherenciájának megtartására. Minden esetben, amikor változás kerül mentésre az adatbázisban a weboldalon keresztül, a weboldal küld egy *request*-et a szerver felé az adott felhasználó címkéjével, így a szerver az ahhoz tartozó rekordokat újra betölti az adatbázisból a *cache*-be.



18. ábra: Komponensek közti kommunikáció a fejlesztés jelenlegi állapotában.

Servletek

Regisztrációs servlet

Ennek segítségével a *request*-ben elhelyezett paraméterek alapján: a név, email cím és a jelszó *hash*-e létrehozza a megfelelő bejegyzéseket az adatbázisban, így elvégezve a regisztrációt, amivel a felhasználó már be tud lépni és beállítani, illetve később módosítani az adatait.

Lekérdező servlet

Ennek segítségével a tükör kliens le tudja kérni egy JSON állományba ágyazva a felhasználó összes adatát, ami alapján a helyi alkalmazások, mit az időjárás a megfelelő adatokat tudja mutatni.

QR kód lekérdező servlet

A tükör kliense lehetővé teszi, hogy egy QR kód segítségével regisztráljuk magunkat egy olyan tükörbe, ahol még nincsen kliensünk, illetve bejelentkeztessük magunkat akkor is, ha nem a megfelelő személynek ismer fel minket a tükör. Ez a *servlet* ezt a két QR kódot tartalmazó képet tudja visszaadni, ha a *request*-ben megtalálható a felhasználó címkéje és a QR kód típusa.

Képfeltöltő servlet

Ennek segítségével a megadott felhasználóhoz a *request*-be ágyazott képek feltölthetőek. Annyi megkötés került bele ebbe a megoldásba, hogy a feltöltendő képek nevének a felhasználó *label*-jével kell kezdődnie és egy alulvonással folytatódnia.

Képletöltő servlet

A tükör kliens munkáját megkönnyítendőn, létrehozásra került ez a *servlet*, aminek segítségével a *request*-ben elhelyezett felhasználó *label*-hez tartozó képeket egy tömörített állományként, hogy azt az arcfelismerő rendszer helyben képes legyen felhasználni.

Cache update servlet

Az szerver működésének meggyorsítása érdekében bevezetett gyorsítótár teljes egészében betöltődik amikor a szerver elindul, az után eszközölt változtatások abban nem jelennek meg, így az a megoldás született, hogy a beállításokat kezelő weboldal küld egy *request*-et az erre a célra kialakított végpontra, amiben a módosított felhasználó *label*-je mint paraméter megtalálható. Egy ilyen hívás esetén a szerver alkalmazás frissíti a *cache* megfelelő sorát.

A mesterségesen generált arcokra vonatkozó mérési eredmények

A módszertanban leírtak szerint az egyes generált arcok közötti távolság mint kimeneti tényező eredményét befolyásoló több faktor hatását vizsgáltuk, a STATISTICA programcsomag segítségével. General Linear Model alapján különálló faktoroknak számított az érzelmi intenzitás (alacsony, magas), az érzelmi kvalitás (öröm, harag), a vektoriális dimenzió szám (128, 512), és mindezen faktoroknak az interakcióját is megvizsgáltuk. Ahol szükséges volt, Newman-Keuls *posthoc* összehasonlítást végeztünk. A szignifikancia szintjét $p < 0,05$-ben állapítottuk meg.

Alanyon belüli és alanyok közötti mérések eredményei

Méréseink során a legalacsonyabb mért távolság 0,04 volt, míg a legmagasabb 1,46 egységnyi volt. Érdekesség képpen elmondható, hogy az 512 dimenziós vektorokat alkalmazó rendszer magasabb távolságokat regisztrált.

A 19-22 ábrákon látható, hogy az 1,1-es határhoz hogyan tartják magukat a 128, illetve 512 dimenziós vektorral operáló rendszerek olyan esetekben amikor két azonos, illetve két különböző alanyról van szó. A függőleges piros vonal mutatja az 1,1-es határt.

Az ábrákon jól látható, hogy az 512 dimenziós rendszer esetében a távolságok javarészt elkülönülnek (22. ábra), azaz minden esetben amikor két nem egyező alany került összehasonlításra a különböző alanyok képei alapján generált vektorok 1,1-nél magasabb távolságot adtak. A 128 (20. ábra) dimenziós rendszer esetén az alanyok közötti távolságok esetén szignifikánsan több esetben történt meg az, hogy rossz alanyt identifikált a rendszer. Verifikáció esetén, azaz alanyok képei között történő összehasonlítás esetén a két implementáció hasonló módon teljesítenek (19. és 20. ábra).

Ezen eredményeknek az összegzése a 23. ábrán látható. Ezen jól látni, hogy az alanyok közötti összehasonlítások távolságai átlagosan 1,1-es érték felett szerepelnek az 512 dimenziós vektorok esetében, míg 128 dimenziós vektorok esetében 1 egész alá esik az átlagos érték. Csoporton belüli mérések által produkált eredményekben megjelenik a már jelzett érdekesség, hogy magasabb a mért értékek átlaga az 512 dimenziós vektorok helyzetében.

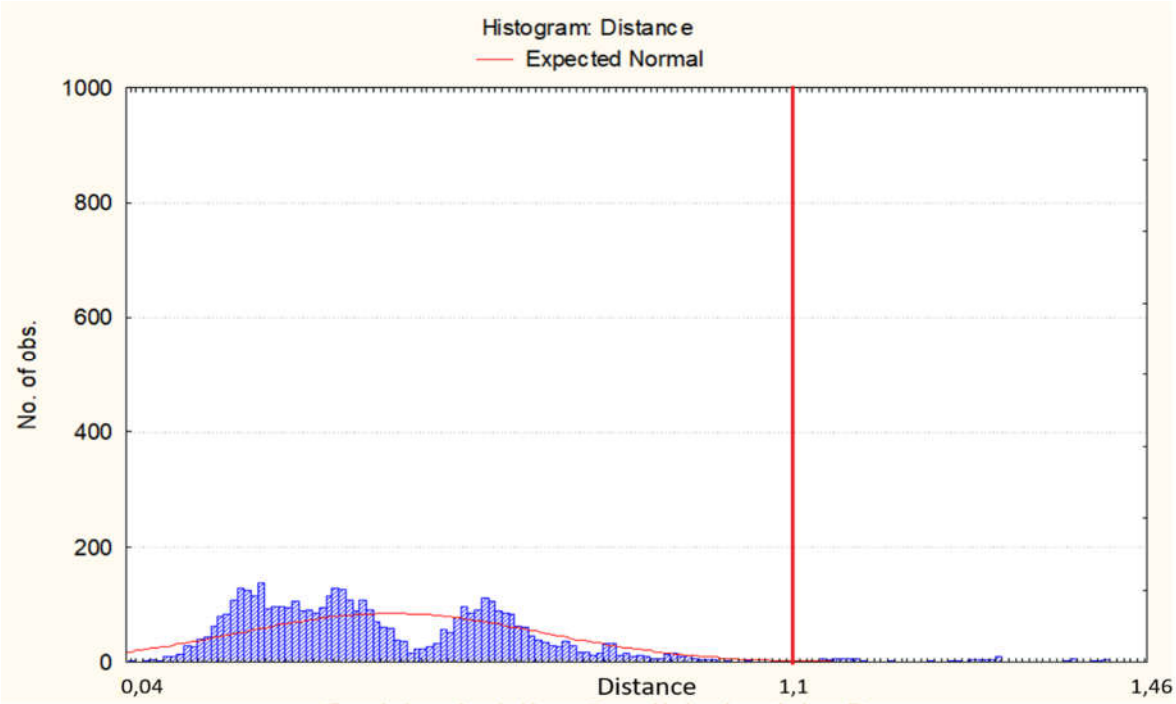
Érzelmek hatásai a mérések eredményeire

A 24. ábrán látható, hogy az érzelmek intenzitása verifikáció esetén negatív hatással vannak a mért távolságokra, ez alátámasztja azt a gondolatot, hogy az érzelmek által létrehozott lokális változások nehézséget okozhatnak az általunk használt, illetve általánosságban a mostani arcfelismerő rendszereknek. Ami még megjelent a mérési adatokban, hogyha minimálisan is, de itt is magasabbak az 512 dimenziós vektorral működő rendszer által mért távolságok.

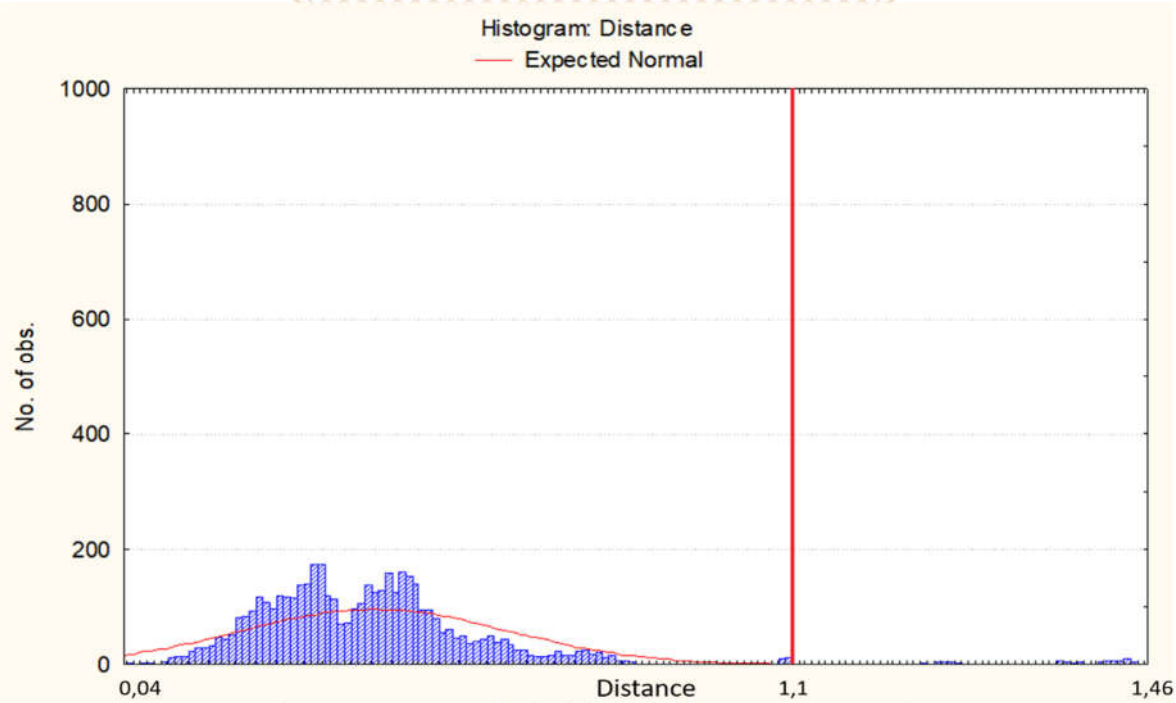
A 25. ábra azt mutatja, hogy a düh és boldogság a felhasználók arcán milyen mértékben befolyásolta a mért távolságokat, abban az esetben, amikor a két kép egy alanyhoz tartozik. Mindkét rendszer esetében látható, hogy a düh az nagyobb mértékben befolyásolta a távolságok alakulását, ezt alátámasztani látszik, hogy a Roesch et al [E. B. Roesch, L. Tamarit, L. Reveret, D. Grandjean, D. Sander, and K. R. Scherer, 2011] által közölt publikációban több Action Unit van meghatározva a dühhöz, illetve Hjortsjö könyvében [C.-H. Hjortsjö, 1970] is az arc több része van meghatározva olyan elemként, mely részt vesz a dühhöz köthető arckifejezéshez, azaz az arc több része változik meg az érzelem intenzitásának növekedésével.

A 26. ábrán az előző ellenpárja, a felhasználók közötti mérések által generált távolságokat mutatja be, illetve az érzelmek és annak intenzitásának mértékét. Az előző adatokhoz hasonlóan, felhasználók közötti összehasonlítások esetén nem befolyásolja olyan mértékben a távolságokat ez a két faktor.

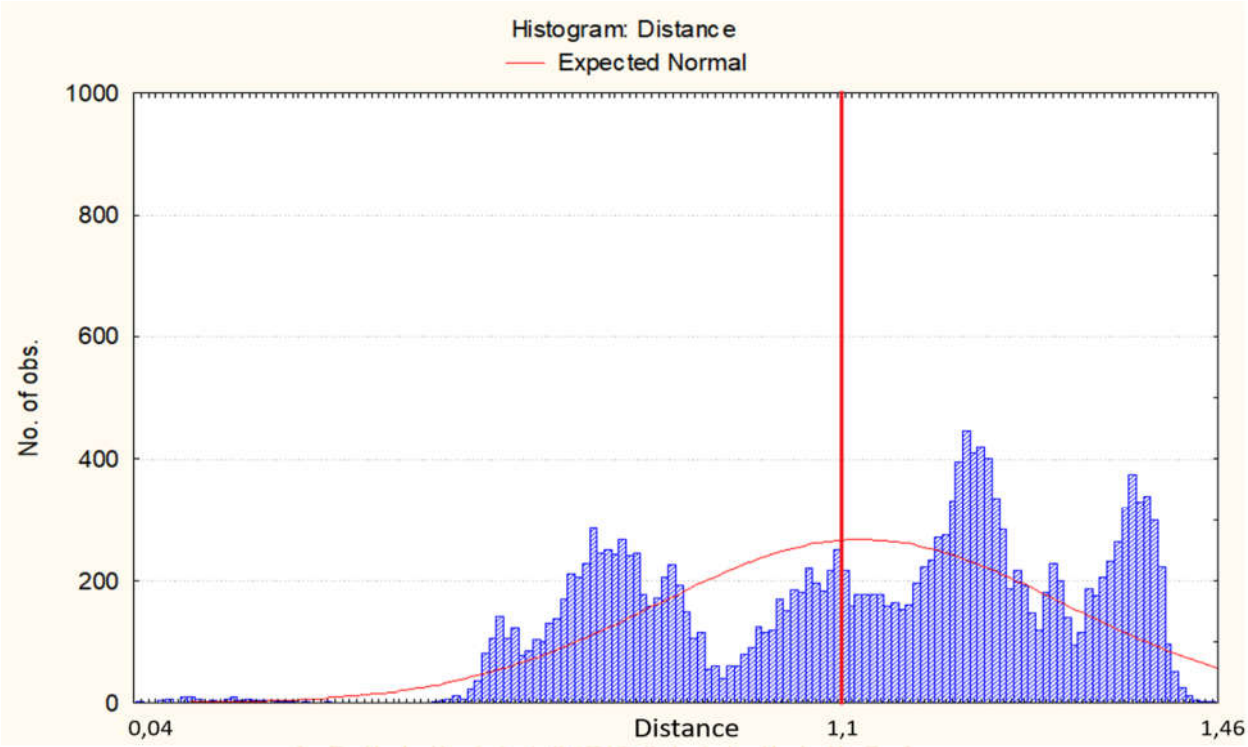
Arra a következtetésre jutottunk az adatok kiértékelése utána, hogy az 512 dimenziós rendszer alapvetően jóval nagyobb grádienssel képes leírni egy emberi arc vonásait, ezzel az árnyaltabb leírással pedig nagyobb távolságok járnak.



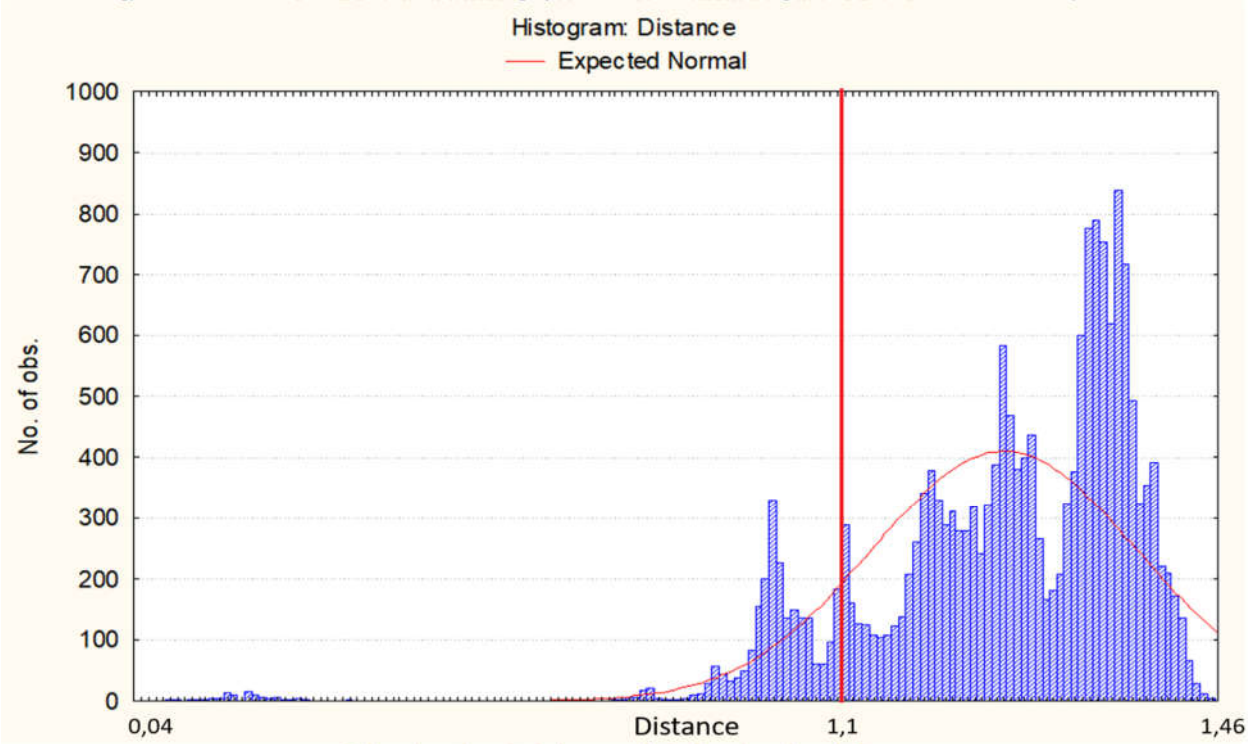
19. ábra: 128 dimenzió, alanyokon belül mért távolságok a vízszintes tengelyen, a függőleges tengelyen pedig az adott távolság gyakorisága. Ezen mérések esetében látható, hogy az esetek többségében a határ alatt helyezkedik el.



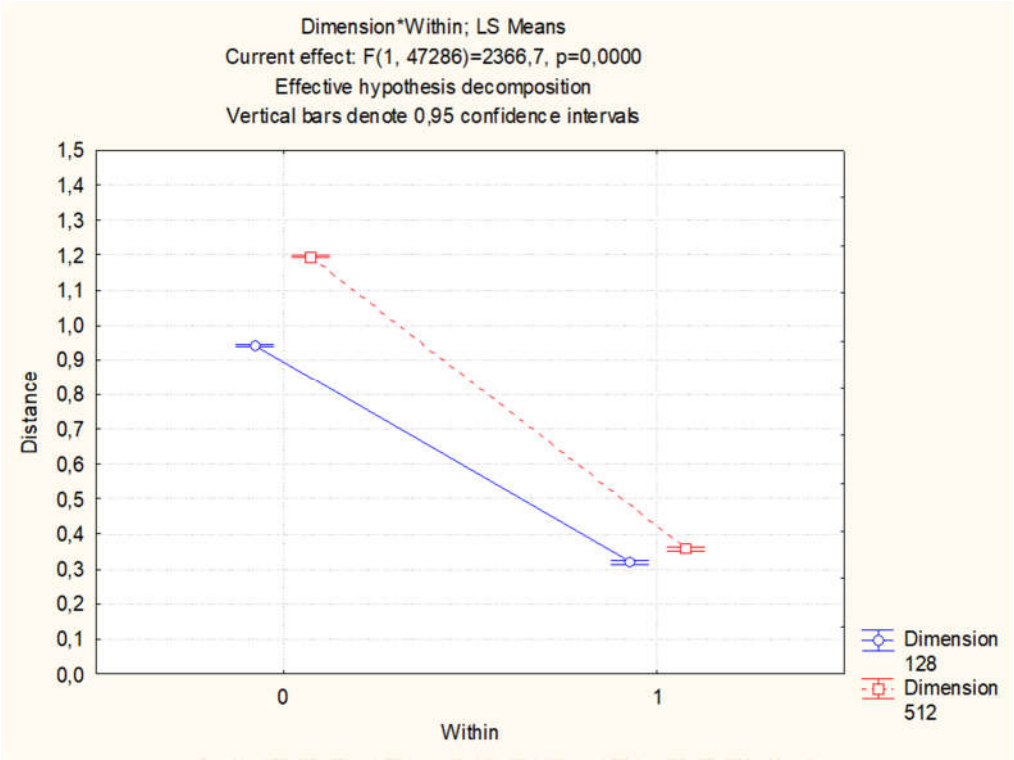
20. ábra: 512 dimenzió, alanyokon belül mért távolságok a vízszintes tengelyen, a függőleges tengelyen pedig az adott távolság gyakorisága. A 128 dimenziós kontrollcsoporthoz képest viszont a mérések normál eloszlásának a csúcsa közelebb van a mért minimális értékhez.



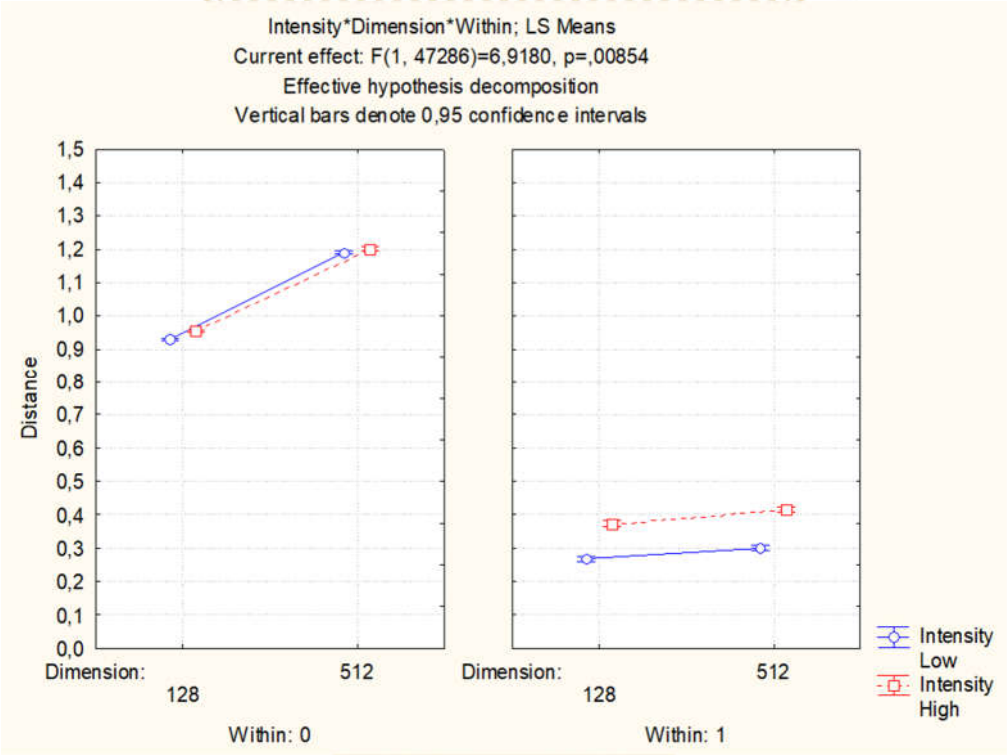
21. ábra: 128 dimenziós vektor, alanyok között távolságok a vízszintes tengelyen, a függőleges tengelyen pedig az adott távolság gyakorisága. Az ábrán látható, hogy a mérések nem felelnek meg az eredetileg meghatározott 1,1-es határnak, azok értékei nagyon szétterülnek, a mérések szignifikáns része a megszabott határ alatt helyezkedik el.



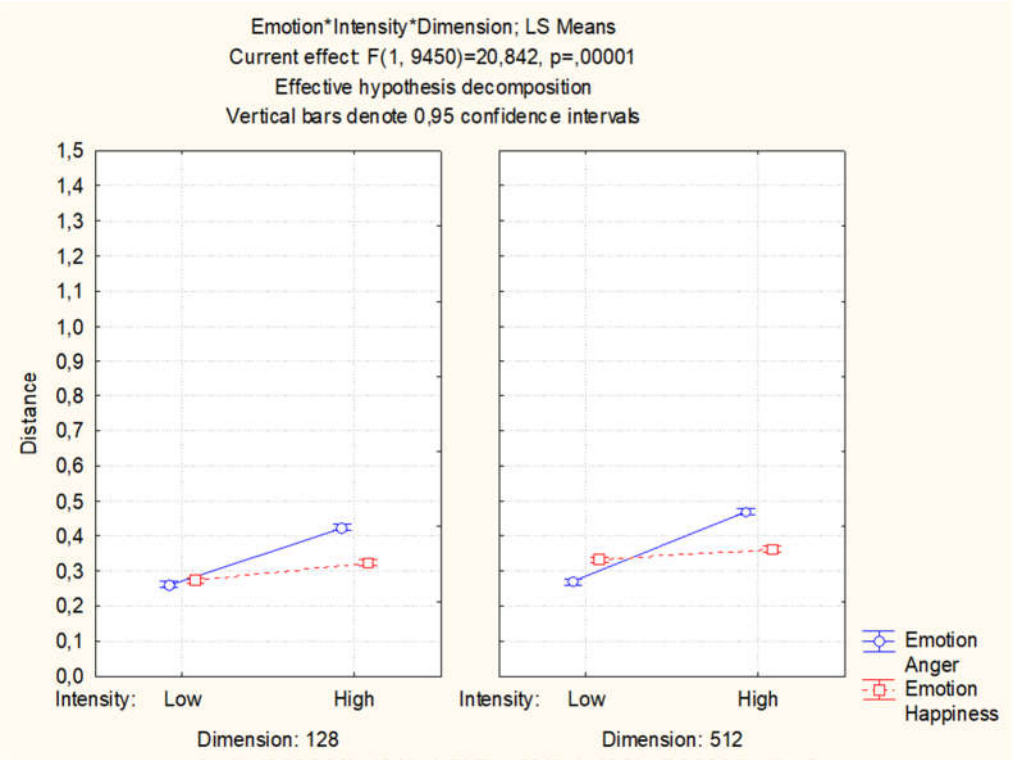
22. ábra: 512 dimenziós vektor, alanyok között távolságok a vízszintes tengelyen, a függőleges tengelyen pedig az adott távolság gyakorisága. A kontroll csoporthoz képeset a mérések közel 4/5-e a határérték felett helyezkedik el.



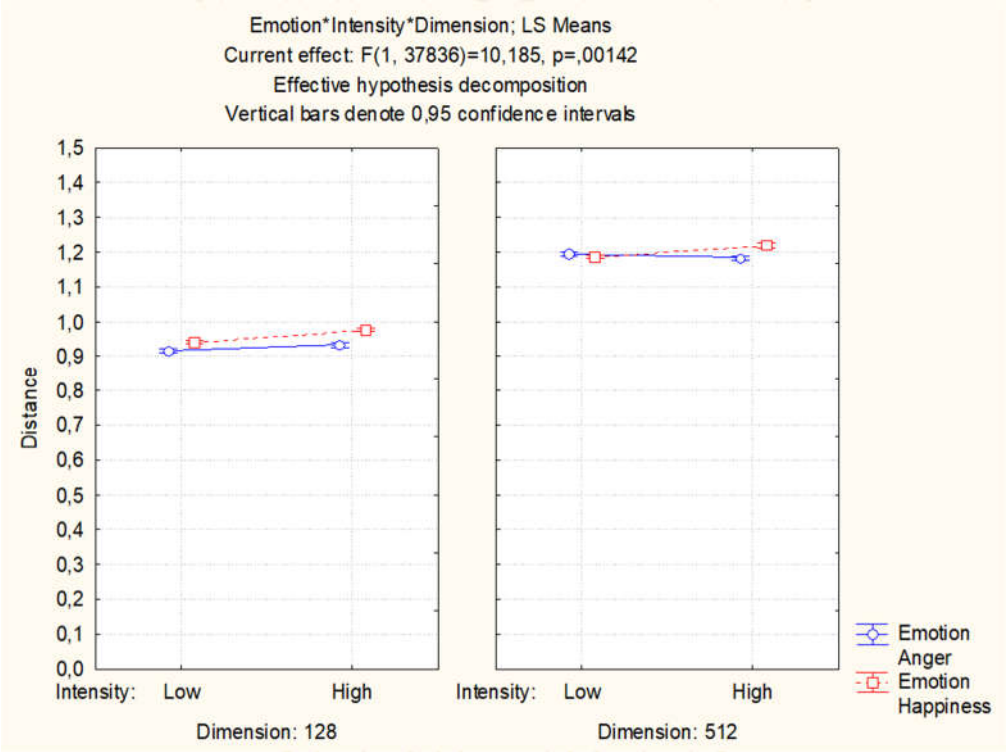
23. ábra: Alanyok közötti (Within = 0) és belüli (Within = 1), 128 és 512 dimenziós vektorok. Látható, hogy az alanyok közötti mérések átlaga a kontroll csoportnál (128 dimenziós) 1,1-es határérték alá esnek. Az 512 dimenziós vektorok ez esetben pedig határérték fölé esnek. Az alanyokon belüli képek esetén a mért értékekben nincs ilyen szignifikáns eltérés, a kontroll csoporthoz képest az 512 dimenziós változatnál minimálisan magasabb távolságok kerültek feljegyzésre. A p érték az adott esetben 0, azaz a mérés eredményei statisztikailag szignifikánsak.



24. ábra: Alanyok közötti és belüli mérések összefüggése a vektorok dimenziószámával és az érzelem intenzitásával. Alanyok közötti mérések esetén különböző érzelmi intenzitások nem mutatnak szignifikáns eltérést, azonban az alanyon belüli mérések igen. Itt is megfigyelhető, hogy azt 512 dimenziós vektorok esetében, magasabb értékek kerültek feljegyzésre. A p érték ebben az esetben 0,00854, így az eredmények ismét statisztikailag szignifikánsak.



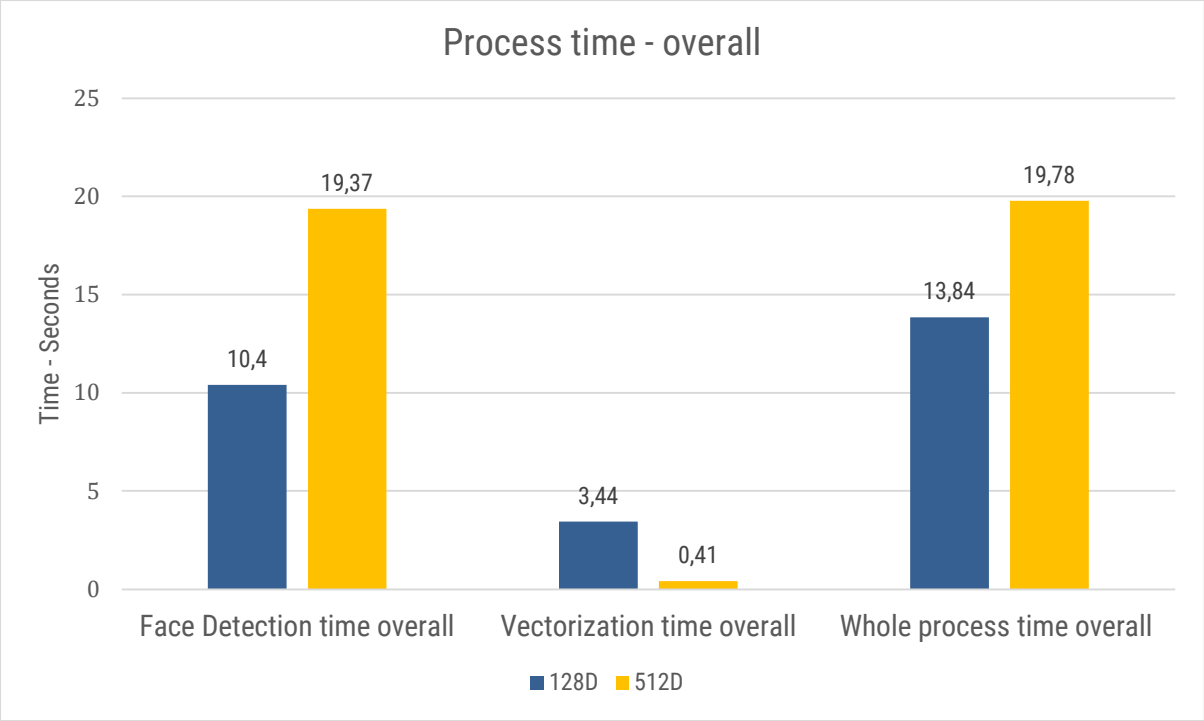
25. ábra: Érzelmek intenzitása és a vektorok dimenziószámának összefüggése alanyokon belüli mérések esetén. Mindkét csoportban megfigyelhető, hogy a düh nagyobb befolyással van a mérések során rögzített távolságok átlagára. A 128 dimenziós kontroll csoporthoz képest az 512 dimenziós vektorok esetén mindkét érzelemnél nőttek a távolságok. A p érték jelen esetben 0,00001, tehát az eredmények statisztikailag szignifikánsak.



26. ábra: Érzelmek intenzitása és a vektorok dimenziószámának összefüggése alanyok közötti mérések esetén. Itt is megfigyelhető, hogy a kontroll csoport (128 dimenziós vektorok) értékei nem érik el az 1,1-es érték határt, míg az 512 dimenziós vektorok által mért távolságok átlaga igen. Látható, hogy ebben az esetben nincsenek olyan nagy eltérések az alanyok átlagaiban, mint a 25. ábrán látható alanyokon belüli mérések esetén. A kontroll csoporthoz képest megfigyelhető, hogy az 512 dimenziós vektorok esetében a távolságok átlaga csökkent az olyan arcképeken végzett méréseknél, amikor magas intenzitású düh volt regisztrálva az alanyok arcán.

Teljesítmény

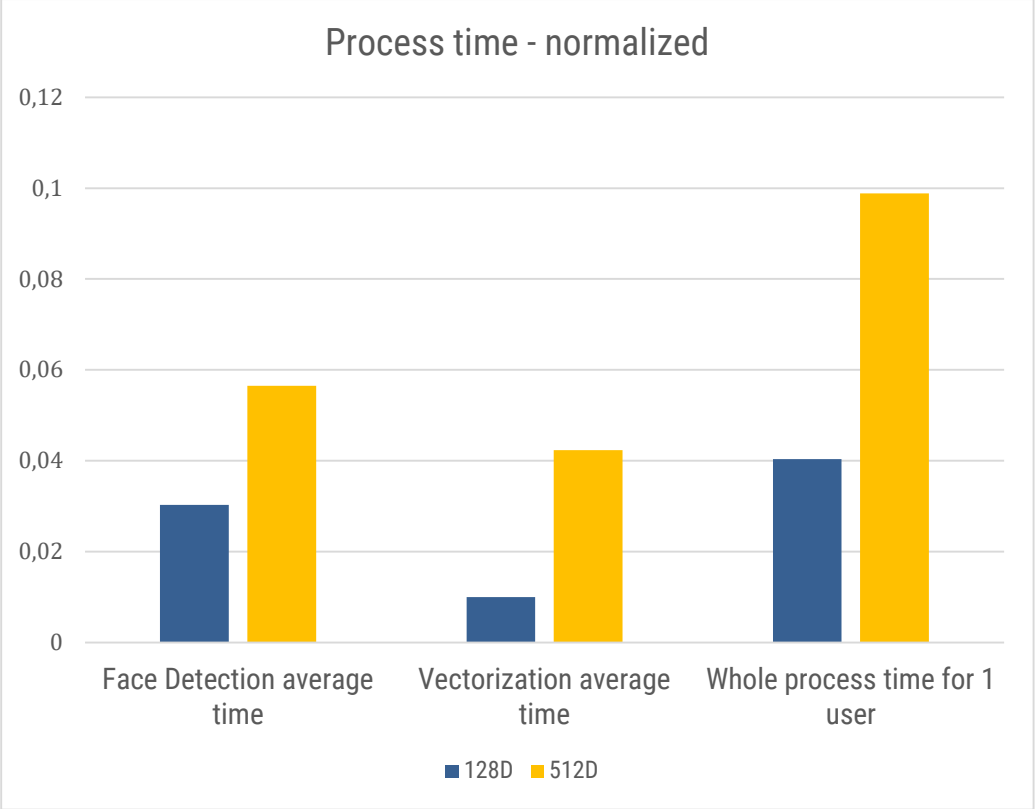
A dimenziók növelésével egyértelmű volt számunkra, hogy növekedni fog a számítási idő is, ezért alapos teljesítmény méréseknek vetetettük alá mindkét implementációt. A kutatási célra használt arc tárat vizsgálva az alábbi eredményeket kaptuk.



27. ábra: látható, hogy az MTCNN arc detektálás nagyjából kétszer annyi ideig tart a 330 képre nézve, mint a Haar Cascade megoldás, azonban a reprezentációs rétegben jól látszik a GPU feldolgozás előnye.

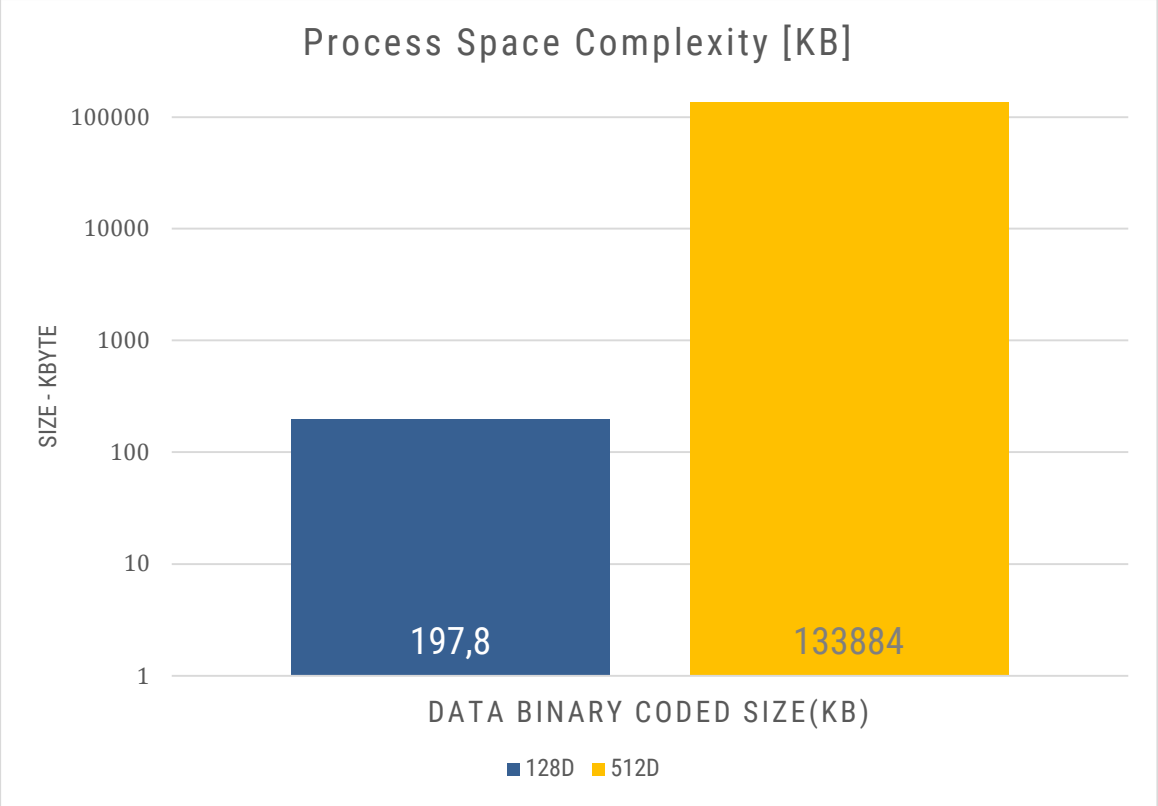
Több kép feldolgozása esetén (26. ábra) az új, CUDA gyorsítást használó reprezentációs rétegünk jelentős teljesítmény növekedést mutat, mivel képes mind a 330 képet egy batchben feldolgozni. Itt érdemes megemlíteni, hogy jelentősen nagyobb képszám esetén előfordulhat, hogy fel kell darabolni több batchre a feldolgozandó képeket, ugyanis a CUDA környezet nem rendelkezik olyan kifinomult swapping technológiával, mint a Windows vagy a Linux, így a programozónak figyelnie kell a memória használatot. Jelen esetben a tesztelést lebonyolító GPU rendelkezett elég memóriával ahhoz, hogy egy batchbe beleférjen az összes feldolgozandó kép.

Az MTCNN arc detektálás nagyjából kétszer annyi időt vesz igénybe, mint a Haar-Cascade, így a feldolgozás legnagyobb időszeletét ez teszi ki.



28. ábra: Egy kép feldolgozási idejére normalizált értékek.

Egy képre normalizálva (27. ábra) a következő eredményeket kaptuk. A kötegelt feldolgozást mellőzve a vektorizálás futási ideje közel négyszeresére nőtt, ami egyenesen arányos a dimenzió számmal. Ezzel együtt az egy képre jutó feldolgozási idő majdnem két és fél szeresére nőtt. Azonban a legnagyobb kiugró teljesítmény metrika az a tár komplexitás.



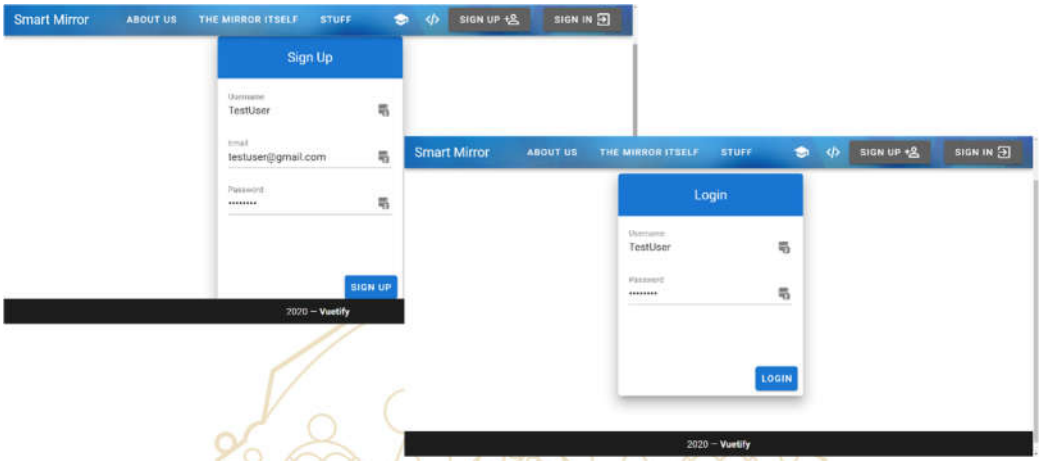
29. ábra: Tárhely komplexitás összehasonlítása. A nagy különbségekre való tekintettel logaritmikus skálát alkalmaztunk

Az 512 dimenziós megoldásunk tárhely igénye több mint hatszázszoros a kisebbik testvéréhez képest. Habár hatalmas a különbség, 130 MB még így sem számít soknak 330 vektorhoz, valamint a tárolt adatok nagy mértékben tömöríthetők, LZMA2 algoritmus Ultra beállítása 611 KB fájlméretet eredményezett.

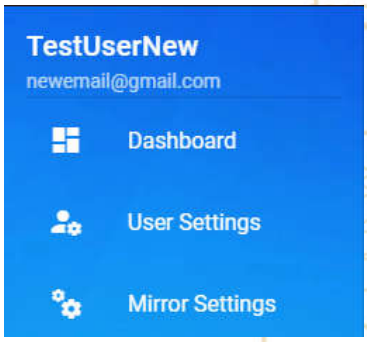
Habár az 512 dimenziót használó rendszer egy felhasználóra nézett teljesítménye futási idő szerint közel két és félszer rosszabb a 128 dimenziós társához képest, a pontosság terén nyert előrelépés miatt jobb választás ez a rendszer, mivel hiába rosszabb a teljesítménye, még mindig bőven elegendő egy felhasználó gyors beléptetéséhez, hiszen mindössze 0,1 másodperc a teljes felismerés ideje. Tárkomplexitás szerint kissé aggasztó lehet a 600 szoros tárhely igény, azonban realisztikus, 3-4 felhasználós környezetben ez nem probléma, valamint több felhasználó esetén van kilátás tömörített formában tárolni az adatokat.

Weboldal Működése

Ahogy fentebb bemutattuk, az első használatot követően a felhasználó a regisztrációs weboldalra lesz átirányítva. Itt a felhasználó létrehozza a profilját, majd az oldal felkéri a bejelentkezésre (29. ábra).

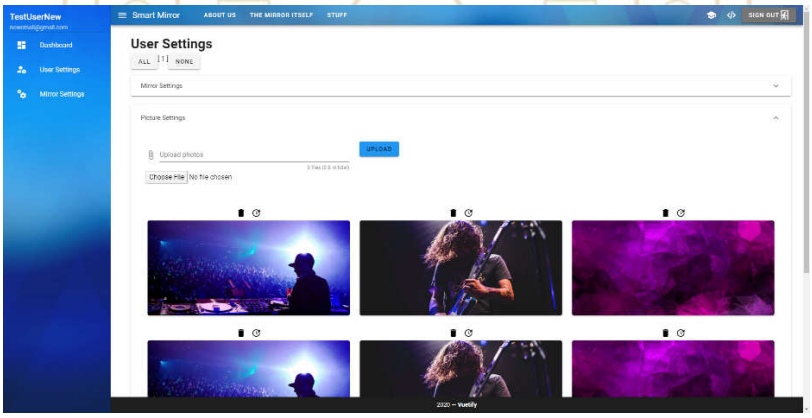


30. ábra: A regisztrációs és bejelentkező oldal



31. ábra A navigációs menü

Bejelentkezés után bal oldalt megjelenik a navigációs menü. (30. ábra) Itt a felhasználó az alábbi menük között tud mozogni. A *Dashboard* fül alatt találhatóak a felhasználó adatai, továbbá a felhasználó itt találkozhat esetleges szolgáltatás frissítésekkel és hírekkel. Abban az esetben, ha a felhasználó módosítani szeretné személyes adatait, a *User Settings* menüpontban találja a releváns opciókat. A *Mirror Settings* fül alatt pedig az arcfelismeréshez kötődő képeket tudja módosítani.



32. ábra: A Mirror Settings fül

Konklúzió

A projekt kitűzött célokat sikeresen teljesítette a csapat. A mérésekhez szükséges arc tár - arckép adatbázis létrehozásra került, az ezekkel folytatott mérések megtörténtek. Az arcfelismerő technológiákhoz kapcsolódó irodalomkutatás megtörtént. Ezek segítségével kiválasztásra került a számunkra megfelelő 512 dimenziós vektorokat és az MTCNN rendszert felhasználó arcfelismerő rendszer.

A szoftver architekturális változásokhoz ki lettek választva a megfelelő módszerek és technológiák, ezeket felhasználva pedig sikerült a három komponenst elkészíteni, ezeknek a kommunikációs interfészeinek egy részét összekötni.

Beszerzésre kerültek a rendszer háttérszolgáltatásaihoz szükséges hardverek, aminek segítségével az adatbázis és az alkalmazás szerver már üzemel.

A tükör kliens programja működőképes, de tovább bővítendő a személyre szabási lehetőségek száma.

A tükör kialakítása még nem végleges, a sérülékeny és vetemedésre hajlamos plexi tükörlapot hosszútávon érdemesebb lenne egy edzett üveglapra cserélni, illetve a tükörhöz készült kliensprogramot egy cél hardverre átültetni egy átlagos laptop használata helyett.

Projekt menedzsment szempontból a csapat működése és annak minden tagja hozta a tőle elvárható teljesítményt, a közel sem ideális körülmények ellenére a tagok az előző évhez képest több időt és energiát fektettek a kutatásba és fejlesztésbe.

A fenti eredmények alapján a projektről elmondható, hogy sikeresen lezárult ennek a dolgozatnak az elkészítésével.

Ábrajegyzék

Referenciák

[1] Singular Inversions Inc., “FaceGen Modeller.” Singular Inversions Inc., 2020, [Online]. Available: www.FaceGen.com.

[2] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, “Face recognition: A literature survey,” *ACM Comput. Surv.*, vol. 35, no. 4, pp. 399–458, 2003, doi: 10.1145/954339.954342.

[3] U. Jayaraman, P. Gupta, S. Gupta, G. Arora, and K. Tiwari, “Recent Development in Face Recognition,” *Neurocomputing*, Apr. 2020, doi: 10.1016/J.NEUCOM.2019.08.110.

[4] F. S. Samaria and A. C. Harter, “Parameterisation of a stochastic model for human face identification,” *IEEE Work. Appl. Comput. Vis. - Proc.*, pp. 138–142, 1994, doi: 10.1109/acv.1994.341300.

[5] L. J. Karam and T. Zhu, “Quality labeled faces in the wild (QLFW): a database for studying face recognition in real-world environments,” *Hum. Vis. Electron. Imaging XX*, vol. 9394, p. 93940B, 2015, doi: 10.1117/12.2080393.

[6] I. Kemelmacher-Shlizerman, S. M. Seitz, D. Miller, and E. Brossard, “The MegaFace benchmark: 1 million faces for recognition at scale,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 4873–4882, 2016, doi: 10.1109/CVPR.2016.527.

[7] N. Nixon, *The Brown Sisters Forty Years*. 2014.

[8] P. Viola and M. Jones, “Robust Real-time Face Detection,” vol. 20, p. 7695, 2001.

[9] P. Viola, M. Jones, and M. Energy, “Robust Real-Time Face Detection Intro to Face Detection,” *Int. J. Comput. Vis.*, vol. 57, no. 2, pp. 137–154, 2004.

[10] C. P. Papageorgiou, M. Oren, and T. Poggio, “A general framework for object detection,” *Comput. vision, 1998. sixth Int. Conf.*, vol. 0, no. January, pp. 555–562, 1998, doi: 10.1109/ICCV.1998.710772.

[11] K. Zhang, Z. Zhang, Z. Li, S. Member, Y. Qiao, and S. Member, “(MTCNN) Multi-task Cascaded Convolutional Networks,” *IEEE Signal Process. Lett.*, vol. 23, no. 10, pp. 1499–1503, 2016, doi: 10.1109/LSP.2016.2603342.

[12] F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A unified embedding for face recognition and clustering,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07-12-June, pp. 815–823, 2015, doi: 10.1109/CVPR.2015.7298682.

[13] F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A unified embedding for face recognition and clustering,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Jun. 2015, vol. 07-12-June, pp. 815–823, doi: 10.1109/CVPR.2015.7298682.

[14] A. T. Tóth, P. S. Tóth, and S. Mészáros, “Okos Tükör Arcfelismeréssel,” in *XLIX. TUDOMÁNYOS DIÁKKÖRI KONFERENCIA*, 2019.

[15] P. S. Tóth, A. T. Tóth, and S. Mészáros, “Concept and implementation of a smart mirror,” in *14 th International Symposium on Applied Informatics and Related Areas organized in the frame of Hungarian Science Festival 2019 by Óbuda University*, 2019.

[16] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems 32 (NIPS 2019)*, 2019.

[17] M. S. Amos, Brandon, Bartosz Ludwiczuk, “OpenFace - Free and open source face recognition with deep neural networks.” .

[18] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[19] M. Greenblatt, “Chromium Embedded Framework.” 2008.
[20] S. H. Williams, “Eel.” 2018.
[21] J. F. Cohn, Z. Ambadar, and P. Ekman, “Observer-based measurement of facial expression with the Facial Action Coding System,” *Handb. Emot. elicitation Assess.*, no. January, pp. 203–221, 2007, doi: 10.1007/978-3-540-72348-6_1.
[22] E. B. Roesch, L. Tamarit, L. Reveret, D. Grandjean, D. Sander, and K. R. Scherer, “FACSGen: A Tool to Synthesize Emotional Facial Expressions Through Systematic Manipulation of Facial Action Units,” *J. Nonverbal Behav.*, vol. 35, no. 1, pp. 1–16, 2011, doi: 10.1007/s10919-010-0095-9.
[23] ShareX Team, “ShareX.” 2020, [Online]. Available: https://getsharex.com/.
[24] J. Willsher, “Bulk Rename Utility.” Acebrook Pty Ltd, 2019, [Online]. Available: https://www.bulkrenameutility.co.uk/.
[25] Apache Software Foundation, “Apache Maven.” Apache Software Foundation, 2020, [Online]. Available: http://maven.apache.org/.
[26] Jb. Inc., “Hibernate ORM.” Red Hat,JBoss Inc., 2020, [Online]. Available: http://hibernate.org/orm/.
[27] SonarSource S.A, “Sonarlint.” SonarSource S.A, 2020, [Online]. Available: https://www.sonarlint.org/.
[28] JetBrains s.r.o., “JetBrains Mono.” JetBrains s.r.o., [Online]. Available: https://www.jetbrains.com/lp/mono/.
[29] D. Knight, “DietPi.” Daniel Knight, [Online]. Available: https://dietpi.com/.
[30] G. Gregory, F. O. By, and L. Demichiel, *Java Persistence with Hibernate 2nd Edition*. Manning Publications, 2006.
[31] Apache Software Foundation, “Apache Tomcat.” Apache Software Foundation, 2020, [Online]. Available: http://tomcat.apache.org/.
[32] E. You, “Vue.js.” 2014.
[33] T. Holowaychuk and StrongLoop, “Express.js.” 2010.
[34] Google, “Angular.” 2016.
[35] K. Schwaber, “Scrum,” 2010. .
[36] C.-H. Hjortsjö, *Man’s face and mimic language*. Studentlitteratur, 1970.