Nyílt forráskódú OCR szoftver felhasználása igazolvány adatok ellenőrzésére

Pásztor Dániel
Automatizálási és Alkalmazott Informatikai Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem
danim1130@gmail.com
Konzulens: Ekler Péter

Abstract. Egy személy magát mind a valós, mind a virtuális világban valamely hivatalos szerv által kiállított igazolvány segítségével tudja igazolni. Ezeknek a szerepe a digitális megoldások folyamatos bővülésével szintén nőtt. Elsősorban a szerencsejáték, pénzügyi, vagy az állam által törvényileg szabályozott területeken működő szervezetek megkövetelik minden beregisztrálónál az azonosítást, ezzel kiszűrve a nem létező személynek kiadó, illetve a más adataival visszaélő felhasználókat.

A kártyák ellenőrzését először emberi erőforrásból oldották meg. Ez kisebb cégek esetén kivitelezhető, az igény növekedésével azonban nehezen lehet lépést tartani.

Az elmúlt évek technológiai fejlődései elérhetővé tették, hogy a nagy számítási komplexitásnak minősülő karakterfelismerést (OCR) szoftverek is el tudják végezni elfogadható időn belül. Ez történhet a felhőben, illetve lokálisan az adott szerveren.

Bár felhő alapú karakterfelismerő szoftverek már nagy számban találhatóak (pl. Google Cloud Vision, Microsoft Azure, ABBYY Cloud OCR, BlinkID), ezek azonban vagy fizetősek, vagy szolgáltatásaik nehezen szabhatók testre. Önállóan futó megoldás már kevesebb van, és ezek közül is csak egy ingyenes, nyílt forráskódú, jelenleg is fejlesztés alatt álló elterjedt OCR létezik, a (szintén Google által fenntartott) Tesseract.

Ezek a megoldások képesek egy képen található dokumentum beolvasására, az adatok ellenőrzésére azonban a naív megoldáson kívül, ahol összehasonlítjuk az OCR által kinyert szöveget a felhasználó által megadottal, egyik szoftver se ad lehetőséget.

Dolgozatomban erre a problémára adok egy nyílt forráskódú, könnyen használható megoldást. Első lépésként konkrétan az adóigazolványon található adatokat igazolom, azonban ez a későbbiekben könnyen általánosítható. Ennek megvalósításához az OpenCV képfeldolgozó könyvtárat fogom felhasználni a Tesseract karakterfelismerő programmal együttműködve, hogy minél pontosabb eredményeket kapjak.

Tartalomjegyzék

1. Bevezetés

Az elmúlt évtizedben az informatika, és ezen keresztül rengeteg egyéb szakterület óriási mértékű fejlődésen ment keresztül. Az Internet megjelenésével és elterjedésével egyre több és több eszközünk áll valamilyen kapcsolatban egymással. Ennek megjelenési formája az IoT (*Internet of Things*), melynek köszönhetően ma már szinte kihívás olyan elektromos eszközt találni, mely valamilyen formában ne lenne képes csatlakozni a hálózatra.

Az eszközökön keresztül az emberek mindennapjai is sok átalakuláson estek át az elmúlt években. Az interneten keresztül eddig még soha nem látott mennyiségű információ vált könnyen elérhetővé és kereshetővé. Rengeteg oldal, illetve alkalmazás jött létre annak érdekében, hogy a mindennapjaink bizonyos részeit (például vásárlás, kikapcsolódás, ismerősökkel való beszélés) a kedvenc eszközünkön keresztül elérhetővé tegyék.

Az interneten található rengeteg szolgáltatás használatához általában valamilyen szinten be kell regisztrálni a szolgáltatóhoz annak érdekében, hogy azonosított személlyé váljunk. Itt általában a felhasználó felelőssége és érdeke, hogy helyes adatokat adjon meg, különben nem tudja az adott szolgáltatást igénybe venni. Vannak azonban olyan területek, ahol a szolgáltató érdeke is, hogy meggyőződjön az ügyfele kilétéről.

Egyik példa erre a pénzügyi szolgáltatások, melynél a pénzmosás, illetve a más nevében történő hitel felvételének megakadályozása miatt szükséges a felhasználó egyértelmű azonosítása. Emellett szükséges az azonosítás akkor is, ha az adott területen valamilyen törvényi szigorítás áll érvényben (pl. szerencsejátékokon csak felnőttek vehetnek részt.)

2018 tavaszán a Magyar Országos Horgász Szövetség azzal a problémával kereste meg a tanszéket, hogy az újonnan bevezetett igazolványok regisztrációjánál szeretnék a felhasználókat az adóigazolványukon keresztül azonosítani. A megvalósítandó megoldás egyedi igényei, valamint a sokféle - és nem mindig jó minőségű - adóigazolvány hatékony kezelése miatt egy egyedi megoldást fejlesztettem ki. A feladat az igazolványok nagy pontosságú felismerése volt, valamint egy olyan architektúra kialakítása, mely nagy igazolvány szám esetén is hatékony és képes jól skálázódni. A kidolgozott algoritmus és archtiektúra általánosan is felhasználható, melyet szintén részletezek a dolgozatban.

A dolgozatom további része a következő struktúrát követi:

- A 2. fejezet a szoftver elkészítésében felhasznált technológiákat mutatja be.
- A 3. fejezet bemutatja az eddig elért eredményeket a témával kapcsolatban.
- A 4. fejezetben található az adóigazolvány ellenőrzésére elkészített megoldás részletes leírása.
- A 5. fejezet bemutatja, hogyan lehet az elkészült szoftvert optimalizálni a végrehajtási időre, illetve a több kép párhuzamos feldolgozására.

- A 6. fejezet kitekintést ad a dolgozat témáján túlra, milyen fejlesztések, általánosítások lehetségesek a projekttel kapcsolatban.
- A 7. fejezet lezárja a dolgozatot, összefoglalja a tanulságokat.

A dolgozatom nyelve a magyar, ez alól egyedül a kód-részletek képeznek kivételt, melyekben a megjegyzések, illetve a változó-elnevezések angolul történnek a további felhasználhatóság céljából.

2. Felhasznált technológiák

Ezen fejezetnek a célja, hogy felsorolja és röviden ismertesse a dolgozatom során használt főbb technológiákat.

2.1. Python

A Python egy magas szintű interpretált nyelv, melynek első verzióját 1991-ben adták ki. Általános célú nyelv, a tervezés fő szempontjai között volt a könnyen olvashatóság, illetve a nagy kifejezőerő. A nyelvhez azóta több frissítést, fejlesztést is kiadtak. 2000-ben jött ki a nyelv 2. verziója, majd 2008-ban a 3. nagyobb átalakítást is közzé tették.

Jelenleg két fő verzió van aktív használatban a Python felhasználók körében. A 2. verzióhoz tartozó utolsó fő frissítést 2010-ben adták ki 2.7-es verziószámmal, míg a 3. verzióhoz a 2016-ban kiadott 3.6-os verzió az elterjedt. Ennek fő oka, hogy a két fő verzió között olyan nyelvi változtatások történtek, melyek nem voltak kompatibilisek a régegbbi nyelvvel.

Manapság már a legtöbb Python könyvtár támogatja az újabb verziót, viszont így is rengeteg kód maradt, mely nem lett frissítve a 3-as verzió kiadása óta. (Pl. Ubuntu alapértelmezetten még mindig a 2.7-es verziót telepíti).

A C, illetve a C++ nyelven írt kódokat könnyű Python oldalról meghívni, így nagyon sok nagy teljesítményű könyvtár vált elérhetővé a nyelv számára. Ez, kombinálva a könnyű használatával és szintaxisával optimálissá tette az adatelemző feladatok elvégzésére, illetve a különböző könyvtárak közötti feladatmegosztás megoldására.

Én a dolgozatom során a 3.6-os verziót fogom bemutatni és használni. A nyár folyamán adtak ki egy kisebb frissítést, a 3.7-es verziót, ez azonban nem tartalmazott a projekt számára lényeges újításokat, illetve még nem volt alkalmam teljes mértékben letesztelni, hogy minden felhasznált könyvtár hibamentesen működik az új verzión is.

Az alábbiakban a szoftver megértéséhez szükséges nyelvi elemek lesznek bemutatva.

2.1.1. Változók deklarálása

A Python egy teljesen objektum orientált nyelv, minden változó egy adott objektumra mutat. Dinamikus nyelv, nincsen típusellenőrzés, ha egy művelet nem hajtható végre egy változón, akkor futáskor erről egy hibát kapunk.

Egy változó deklarálásakor egy létező objektumnak adunk egy nevet, mellyel később is elérhetjük. Külön utasítás nincs a változónév létrehozására, az első értékadás műveletnél létrejön magától, melyet ezután bárhonnan elérhetünk. Szükség esetén a *del* paranccsal törölhetjük az adott változónevet.

Ezekre a műveletekre mutat példát a 1. példakód.

1. Kód. Változó deklarálás

```
x = 1
```

```
x = "Test"
print(x) #"Test"
#print(y) #NameError: name 'y' is not defined
del x
#print(x) #NameError: name 'x' is not defined
```

2.1.2. Függvények

A változók mellett természetesen lehetőségünk van függvényeket is deklarálni. A függvények kódját az indentáció határozza meg: a függvény deklarálása után minden a függvényhez tartozó kódot valamilyen konstans mértékben (pl. 1 tab) indentálni kell. Egy függvénynek tetszőlegesen sok változója lehet, melyeknek lehet alapértelmezett értéke is. A függvényből a *return* kulcsszóval lehet visszatérni, melyben akár több visszatérési értéket is megadhatunk. Több visszatérési érték esetén az összes érték egy objektumba lesz becsomagolva, melyből indexeléssel érhetjük el az adott értékeket, vagy egyből betölthetjük az összes változót változónevekbe.

A függvények használatáról a 2., illetve a 3. kód mutat példát.

2. Kód. Függvény, paraméterek

```
def my_func(x = 3):
    print(x)

my_func() #"3"
my_func(3.0) #"3.0"
my_func("Test") #"Test"
```

3. Kód. Függvény, visszatérési érték

```
def my_func():
    return 1, 2

x = my_func()
print(x) #"(1,2)"
print(x[0]) #"1"
x, y = my_func()
print(x) #"1"
print(y) #"2"
```

2.2. Elágazások

Az elágazásokat a legtöbb nyelvből ismert *if-elif-else* szerkezettel lehet elérni. A függvények deklarásához hasonlóan az indentáció határozza meg, hogy egy elágazásnak a blokkja meddig tart. Erre mutat példát a 4. kód.

4. Kód. Elágazások

```
x = 3
if x % 2 == 0:
    print("The number is even.")
elif x > 0:
    print("The number is positive odd.")
else:
    print("The number is negative odd.")

print("The number has been evaluated")
```

A többi nyelvtől eltérően elsőre kicsit szokatlan lehet, hogy az elágazásokban deklarált változók nincsenek az adott blokkhoz kötve, azok kívülről is olvashatóak. Ez sok nehezen észrevehető hiba forrása lehet, ezért oda kell figyelni, hogy lehetőleg minden elágazásban ugyanazokat a változókat deklaráljuk (illetve szükség esetén töröljük). Ezt mutatja be a 5. kód.

5. Kód. Elágazások

```
x = 2
if x % 2 == 0:
    y = 0
else:
    print("The number is odd.")

print(y) #Throws error if x is odd, runs fine if x is
    even.
```

A más nyelvekben megszokott *switch* struktúrának nincs Python megfelelője, azt csak az *if-elif* struktúrával lehet kiváltani.

2.2.1. Python tárolók

A Python nyelv is biztosítja a legtöbb nyelvben megtalálható tárolókat:

- Halmaz (*set*): A halmaz a matematikai definíciónak megfelelően egyedi, megkülönböztethető elemek együttese. Hatékonyan tudjuk tesztelni, hogy egy elem része-e a halmaznak, illetve szükség esetén végig tudunk iterálni a halmaz összes elemén. Létrehozása a *set()* operátorral történik.
- Lista (*list*): A lista tetszőleges elemek valamilyen sorrendben lévő összessége. A halmazokon végzett műveletek mellett támogatva van az index alapú elemkivétel. A Python lehetővé teszi a lista legkülönbözőbb indexelési módjait. Létrehozása a [ ] operátorral történik, melyen belül vesszővel elválasztva adjuk meg a kezdő értékeket.
- Szótár (*dictionary, map*): A szótár a programozó által megadott kulcs-objektumhoz rendelt érték-objektum párok összessége. Létrehozása a *{ }* operátorral történik.

Ezen beépített struktúrák mellett érdemes még megjegyezni a *numpy* könyvtár által biztosított ndarray struktúrát. Ez gyakorlatilag egy C-ben megvalósított, tetszőleges dimenziószámú és méretű tömb, mely a Python listáján definiált műveletek mellett egyéb segédműveleteket is megvalósít. Ennek következtében előre megadott típussal rendelkezik, cserébe viszont nagyságrendekkel gyorsabb, mint egy lista.

A különböző struktúrákra a 6. kód mutat példát.

6. Kód. Alapstruktúrák

```
#Example for lists
a = [0, 1, 2]
print(len(a)) #"3"
print(max(a)) #"2"
print(a[0]) #"0"
print(a[-1]) #"2" #Prints the last element inserted.
    Generally, a[-i] = a[len(a) - i]
print(a[0:2]) #"[0, 1]" #Slices the array

#Example for sets
b = set()
b.add(1)
b.add(2)

#Example for working with sets and lists
print(1 in b) #True
for x in a: print(x) #"0 1 2"

#Example for dicts
c = {}
c["key1"] = "value1"
c[a] = "value2"
c[b] = b
```

2.3. OpenCV

Az OpenCV [1] [2] fejlesztését az Intel kezdte el 1999-ben. A projekt fő célja a gépi látás fejlődésének elősegítése volt egy közös, optimalizált keretrendszer létrehozásával. Teljesen ingyenesen használható mindenféle megkötés nélkül, C/C++ nyelven lett fejlesztve, így gyakorlatilag tetszőleges platformra lehet vele fejleszteni. Elsősorban a C++ alapú fejlesztést támogatják, de egyéb nyelvekből is elérhetőek a különböző funckiók, mint például Python, Java, C#.

A projektnek sikerült elérnie a fő célját, jelenleg gyakorlatilag ez az egyetlen könyvtár, mely elérhetővé teszi szinte az összes képmanipulációs műveletet egy könnyen használható felületen keresztül, legyen az akár egy egyszerű képátméretezés, vagy egy bonyolultabb mintaillesztési feladat.

A projekt során a 3.4.0.12-es OpenCV verziót használtam, melyhez a Python 3.6 hivatalosan támogatott nyelv.

2.4. Tesseract OCR

A Tesseract OCR (*Optical Character Recognition*) egy karakterfelismerő program. Eredetileg 1985-1994 között fejlesztették a Hewlett-Packard cégnél. A projektet 2005-ben nyílt forráskódúvá alakították, majd 2006-ban a Google úgy döntött, szponzorálni fogják a fejlesztést. Ennek köszönhetően óriási fejlődésen ment át a szoftver mind teljesítményileg, mind pontosságilag. Várhatóan a közeljövőben kiadásra kerül a 4. verzió, melyben a hagyományos felismerő rendszer mellett egy LSTM alapú neurális háló is helyet kapott, ezzel is tovább növelve a beolvasóképességét a szoftvernek.

Bár a Tesseract-OCR funkciói elérhetőek egy C++ API felületen keresztül, mégis inkább az önálló, parancssorból meghívható alkalmazáson keresztül szokták használni, így ez gyakorlatilag tetszőleges programozási nyelvből elérhető.

3. Irodalomkutatás

A projekt során a Python nyelvet használom, mely kifejezőereje és nagy támogatottsága miatt ideális különböző könyvtárak illetve szoftvercsomagok közötti működés létrehozására. Ehhez elengedhetetlen a dokumentáció, mely részletes leírást tartalmaz az adott verzióhoz tartozó beépített függvényekről, szintaxisról [3].

A munka során hasznos volt az OpenCV különböző funkcióinak használata. Ennek egy rövid bemutatása megtalálható az alábbi cikkben [4]. Az alapvető elemek bemutatása után rövid bekezdésekben összefoglalják az általánosabb képmanipulációs műveleteket, melyek megtalálhatóak a könyvtárban. Bár eredetileg az OpenCV 2. verziójához készítették, eléggé általános fogalmakat tartalmaz ahhoz, hogy még most is hasznos olvasmány legyen. Bármikor, amikor elakadtam valami nem triviális problémán, ebből indultam ki a szükséges kulcsszavak beazonosításához.

Következő fontos forrás volt az OpenCV hivatalos dokumentációja [5]. Ez a bevezető után már könnyen követhető, kódokkal és képekkel illusztrált leírást ad az OpenCV legtöbb beépített funkciójáról, kezdve az alapoktól, mint például fájlműveletek, egészen az arcdetektálás, illetve objektum követés témakörökig.

A feladat megoldásának egyik kulcs lépése a *SIFT* algoritmus használata. Ennek bemutatása a dolgozatomban csak érintőlegesen történik, az alábbi cikkben viszont egy konkrét implementáció mellett maga az algoritmus is részeltesen le van írva [6].

A *SIFT* algoritmusra létezik szabadalom, mely egyelőre csak Amerikában érvényes, így Európában nincs jogi probléma annak felhasználásában. Ha szükséges lenne, léteznek ingyenesen felhasználható algoritmusok, mint például az *ORB* [7]. Ezt, illetve pár egyéb alternatívát a dolgozatom során kipróbáltam, és bár a feladatnak megfeleltek, a *SIFT* jobb eredményeket produkált.

Az igazolványon található adatok beolvasása során egy szükséges extra lépés volt, hogy meg kellett találni, pontosan hol is helyezkedik el a szöveg a kártyán. Ehhez remek kiinduló forrás volt a [8] cikk. A cikkben egy olyan detektálót hoznak létre, mely rengeteg különböző típusú szöveg pozícióját képes felismerni egy adott fényképen. Mivel az én esetemben tudtam előre, hogy körülbelül milyen szövegre számíthatok, ennek egy sokkal egyszerűbb verzióját elég volt implementálni.

A szoftver egy másik jelentős komponense a szövegdetektálásra használt Tesseract-OCR szoftver. Ennek bemutatására szolgál az alábbi cikk [9]. Ezt még a Tesseract 2. verziójának elkészítésekor írták, azóta jelentősebb fejlődéseken ment keresztül, de a cikkben leírt felismerés bizonyos szinten még a legfrissebb verzióban is benne van.

A Tesseract-OCR egy komplex szoftver, rengeteg opcionálisan konfigurálható tulajdonsággal, illetve optimalizációs lehetőségekkel. Ezek részletes megismerésében segít a Wiki oldaluk [10], melyen a legfrissebb verzióhoz tartozó beállítások is elérhetőek. Emellett itt jelennek meg a rövidebb összefoglalók a nagyobb változtatásokról.

Az OpenCV és a Tesseract kombinálása képi szövegfeldolgozás feladatra természetesen nem eredeti ötlet, előttem már sokan mások is használták ezt a két népszerű könyvtárat, például autók rendszámának detektálására [11]. Már régebben is arra az eredményre jutottak, hogy bár a Tesseract nem tökéletes karakterbeolvasó szoftver, de általában így is eléri a fizetős szoftverek pontosságát.

A meglévő megoldásokat és irodalmat áttekintve látható, hogy a téma aktívan vizsgált a tudományos életben, hasonló kutatásokon már dolgoznak [12] [13] [14]. A megoldásom egyedi jellege a többféle és nagyon eltérő igazolvány minőségből adódik, valamint a mellé épített skálázódó architektúrából.

4. Adóigazolvány ellenőrzés

A fejezet célja az adóigazolvány ellenőrzéséhez szükséges munka, illetve döntések bemutatása. Először a probléma ismertetése kerül sorra, majd a megoldást mutatom be lépésről lépésre.

A munka során sok példaigazolvány kerül bemutatásra. Minden olyan ábrán, ahol az egész kártya bemutatásra kerül, külön engedélyt kértem az adott igazolvány tulajdonosától. Azokban az esetekben, ahol a tulajdonost nem tudtam elérni, csak egy-egy mező kerül bemutatásra az adatok védelme miatt.

A teljes kód megtekinthető a publikusan elérhető GitHub projektben [15].

4.1. A probléma ismertetése

A Magyar Országos Horgász Szövetség az új igazolványrendszerük bevezetése miatt szükségesnek látta, hogy a horgászok a regisztrálás során beazonosítsák magukat. Az azonosításhoz ők az adóigazolvány mellett döntöttek, olyan érveléssel, hogy egyedül az adóigazolványszám az, ami egy ember élete során soha nem változik.

Mivel az új rendszerbe kötelező regisztrálnia a horgászoknak, rengeteg adóigazolványt kell beolvasni, és leellenőrizni, hogy valóban a felhasználó által megadott adatok szerepelnek-e rajta. Az ehhez szükséges erőforrás csökkentése érdekében keresték meg a tanszéket azzal a feladattal, hogy próbáljuk meg minél jobban automatizálni ezt az ellenőrzést.

A következő követelmények voltak kitűzve:

- A rendszernek fel kell ismernie az összes jelenleg hatályos adóigazolvány kártya típust.
- Az adatok ellenőrzésére egyetlen egy kép áll rendelkezésre, mely lehet kamerával készített, illetve szkennelt színes kép.
- A kártyán lévő összes adatot ellenőrizni kell.
- Egy kártyát mindenképp el kell utasítani, ha egyértelműen hamis adat van megadva azon.
- Kiemelten figyelni kell a hamis pozitív esetekre, vagyis mikor a rendszer úgy fogadja el a kártyát, hogy a felhasználó valójában hibás adatot adott meg. Ennek kiderülése esetén újra kell nyomtatni a horgász halászkártyáját, mely többletköltség a megrendelőnek.
- A rendszer az elutasítás és az elfogadás mellett dönthet további ellenőrzés szükségességéről, amikor egy emberi kezelőnek is szükséges ellenőriznie a kártyát.

Ezek mellett a következő egyedi igényei voltak a megrendelőnek, melyekre a fejlesztés során oda kellett figyelni:

- Kamerával készített kép esetén a fotós valószínűleg egy hétköznapi ember lesz, aki nem feltétlen figyel oda a jó világításra, szögre, vagy fókuszra.

- Szkennelt kép esetén előfordulhat, hogy más kártya is benne van a képben (pl. sokan egyszerre szkennelik be az összes igazolványukat).
- Lehetőleg érjük el a 80%-os elfogadási arányt a jó minőségű képek esetén.

4.2. Információgyűjtés

A probléma pontosabb megismerését az információgyűjtés követte.

Magyarországon jelenleg két adóigazolvány kártya típus hatályos. Pontos dátumot nem sikerült találnom arra, mikor vezették be az új kártyákat, de a tesztelt kártyák alapján ez valamikor 2012-ben történhetett meg.

A két kártyatípus:

- Régi (2012 előtti): Papírkártya, melyre külön rétegként vitték fel az ügyfél adatait. A régebbi kártyák elszíneződnek, illetve beszakadoznak. Egy jobb minőségű régi kártyára láthatunk példát a 1. ábrán.

1. ábra. Régi adóigazolvány példakép

- Új (2012 utáni): Plasztikkártya, melyen jó minőségben szerepelnek az ügyfél adatai. A NAV által biztosított példakártya látható a 2. ábrán.

Ahogy a képeken is látható, az igazolványok az alábbi adatokat tartalmazzák:

- Adóazonosító jel: 10 számjegy
- Név
- Születési idő

2. ábra. Új adóigazolvány példakép

- Születési hely
- Anyja neve
- Kiállítás kelte
- Vonalkód: pár próba után egyértelművé vált, hogy az adóazonosító jelet tartalmazza.

Ezek mellett az új adóigazolványokon még a következő adat található:

- Sorszám: Két karakter, melyet 6 számjegy követ.
- Adatmátrix: A sorszámot tartalmazza

Ezek alapján egyértelmű, hogy milyen adatokat kell beolvasni. A vonalkód, illetve az adatmátrix segít abban, hogy az adóazonosítót, illetve a sorszámot könnyen ki tudjuk olvasni, mindenféle OCR szoftver nélkül.

Ezután megvizsgáltam, hogy van-e valami rendszer az adóazonosító jelben. Erről a következő információt találtam:

- Mindenképp 10 számjegyből áll.
- Az első számjegy kötelezően 8
- A 2.-6. számjegy azt fejezi ki, hogy az adott személy hanyadik napon született 1867. január 1-től számolva. (Ekkor alakult meg az önálló magyar vámhatóság)

- A 7.-9. számjegy egy (szinte, lsd. következő pont) véletlen számhármas, mely az azonos napokon születetteket hivatott megkülönböztetni.

- Az utolsó, 10. számjegy egy ellenőrző számjegy. Ez úgy kapható meg, hogy az összes előtte álló számjegyet megszorozzuk a számjegy helyének indexével, ezeket összeadjuk, majd vesszük a 11-el vett maradékát. A 7.-9. számjegyet tilos úgy választani, hogy végén keletkező maradék 10 legyen.

A sorszámra nem találtam ehhez hasonló szabályt.

Ezek alapján tehát egy vonalkód-olvasással máris megtudhatjuk az adóazonosító jelet, illetve abból származtatva a születési időt. A maradék 4 mezőre viszont nincs más lehetőség, mint valamilyen karakterfelismerő programmal kiolvasni a szöveget, és így ellenőrizni.

A szöveges mezők ellenőrzésére két megoldást láttam kivitelezhetőnek a projekt elején:

- A felhasználó által megadott adatokból minden egyes mezőre kirajzolom azt a szöveget, aminek szerepelnie kéne, és megvizsgálom, ez mennyire egyezik meg a kártyán az adott mezőhöz tartozó területtel. Ehhez szükséges, hogy előre ismerjük a betű típusát, méretét, illetve pontosan tudjuk a helyet is, ahol kereshetjük.

- Az adott mezőhöz tartozó területet egy OCR szoftverrel kiolvasom, majd összehasonlítom a felhasználó által megadott adatokkal. Ha egyezik, akkor az adott mező elfogadásra kerül.

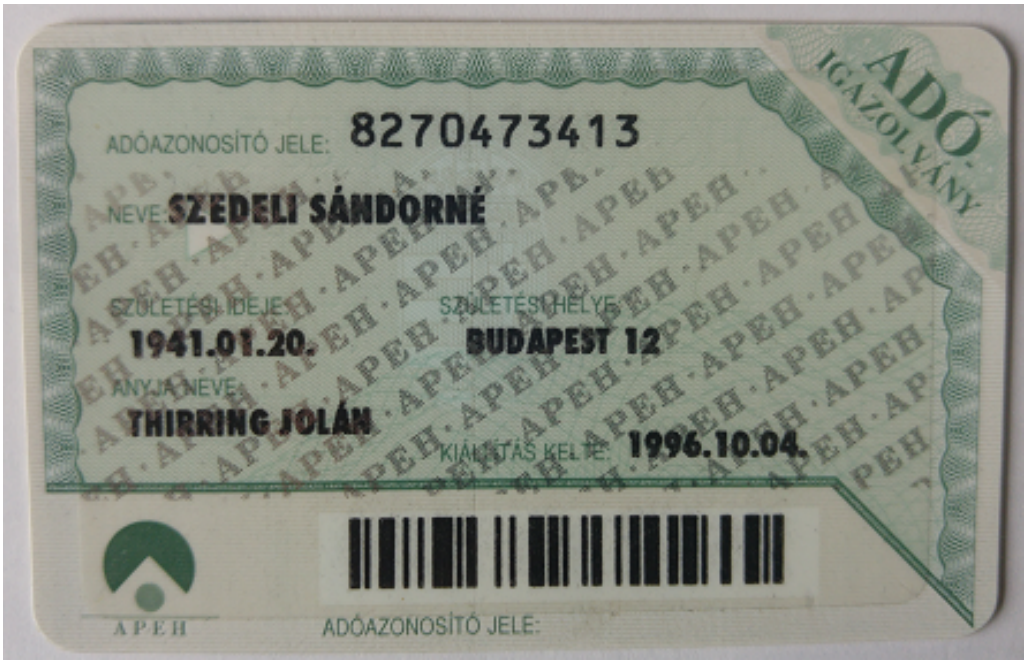
Teljesítményileg egyértelműen az első megoldás lenne optimális, ehhez viszont mindenképp ismerni kéne a betűtípust a kártyákon. Több példakártya begyűjtése után egyértelművé vált, hogy míg az új kártyákon mindenhol egységesítve van a betűtípus, addig a régi kártyákról ez nem mondható el.

Egy másik, vastagabb betűtípusra mutat példát a 3. ábra. Ahogyan látható, itt teljesen más betűstílus van használva, mint az előző, 1. ábránál. A *B* karakter például gyakorlatilag egy egybeolvadt folt, a két lyuk csak nehezen vehető ki belőle.

Ezek mellett csak a kevés, családi körből begyűjtött régi adóigazolványok közül sikerült egy harmadik betűstílust is beazonosítani. Mivel a kevés tesztkártya közül is sikerült egyből 3 különböző betűstílust is találni, nem mertem a szövegkirajzolós megoldást választani. Itt kezdésből 3 különböző betűtípushoz kellett volna kigyűjtenem a teljes karakterkészletet, majd mindhárom verzióval le kellett volna tesztelni a régi kártyán található mezőt. Emellett nem garantálta semmi se, hogy például vidéken nem-e egy teljesen másik betűtípust használtak, ami csak a rendszer beindítása után derült volna ki.

Ezen indokok miatt egy OCR szoftvert használtam, mely nagyobb erőforrásigénye mellett általánosabb megoldást biztosít.

Ezeknek figyelembevételével a szoftvernek az alábbi lépéseket kell megtennie annak érdekében, hogy sikeresen leellenőrizze az igazolvány adatait:



3. ábra. Régi adóigazolvány vastag betűtípus

1. A felhasználó által beküldött képen meg kell keresni az adóigazolvány képét. Több kártya esetén biztosítani kell, hogy az adóigazolványt találjuk meg.

2. Az igazolványt be kell forgatni, illetve szükség esetén vissza kell alakítani a kamera miatt keletkező perspektív transzformációt annak érdekében, hogy a mezők helyét könnyen meg tudjam határozni.

3. Detektálni kell, hogy régi, vagy új adóigazolvány szerepel a képen.

4. A felhasználó által megadott adatokat ki kell olvasni a kártyáról, és azokat össze kell egyeztetni.

5. Az adott eredményeket ki kell értékelni, és a 3 megengedett válaszopció (elutasít, nem biztos, elfogadó) egyikével vissza kell térni.

4.2.1. Adóigazolvány detektálás, forgatás

Első lépésként tehát detektálni kell az igazolványt. Mivel a detektálás során nagy valószínűséggel a kártya mérete is kiderül, illetve az egyéb kártyák kiszűrése miatt a kártya típusa is, ezért az első három lépés gyakorlatilag összevonható.

A kártya megtalálására sok megoldást lehet elképzelni, én most röviden azokat ismertetném, melyeket ki tudtam próbálni:

- A kártya éleinek detektálása: Logikus gondolatnak tűnik, hogy keressük meg a kártya éleit a képen, majd ez alapján végezzük el a kép visszaforgatását. Ezzel a probléma az volt, hogy nagyon érzékeny volt a kártya hátterére: szkennelt képeknél a kártya éle sokszor beleolvadt a háttérbe, kamerával készített képeknél pedig sokszor bezavart a környezet. Emellett nem volt megoldva, hogy a kártyák közül csak az adóigazolványt szűrjük ki.

- A kártya különleges tulajdonságainak detektálása: Első sorban a vonalkód detektálása alapján próbáltam meg kiszűrni, hol is lehet a kártya. Ehhez egy külső könyvtárat, a *pyzbar*[16][17]-t használtam, mely többek között vonalkódbeolvasást is támogat. A tervem az volt, hogy a vonalkód detektálásából kiindulva visszafejtem, hol lehetnek a kártyák sarkai, viszont sajnos már a vonalkód detektálásánál a *pyzbar* sokszor nem adta vissza a vonalkód helyét, illetve nem is működött jól elforgatott kártyával.

Ezek után döntöttem úgy, hogy kipróbálok egy már létező, profi mintakereső és illesztő algoritmust.

Az első algoritmus, amiről rengeteg példakódot és információt találtam, a SIFT[6] volt (*Scale Invariant Feature Transform*). A SIFT egy olyan algoritmus, mely képes egy képen kikeresni a számára érdekes pontokat, és ezekről egy leírást készíteni. Érdekes pontnak van nevezve az a pont, melynek környezetében nagyobb változások történnek például a kép fényerejében (pl. sarkok, élek találkozása). Ezek a leírók bizonyos mértékben skála, illetve forgatás-invariánsak, így optimálisan össze lehet hasonlítani két kép tartalmát.

A terv tehát az, hogy elkészítek egy minta képet, mely helyesen be van forgatva, illetve transzformálva, és megpróbálom a bejövő képet ehhez a mintaképhez illeszteni.

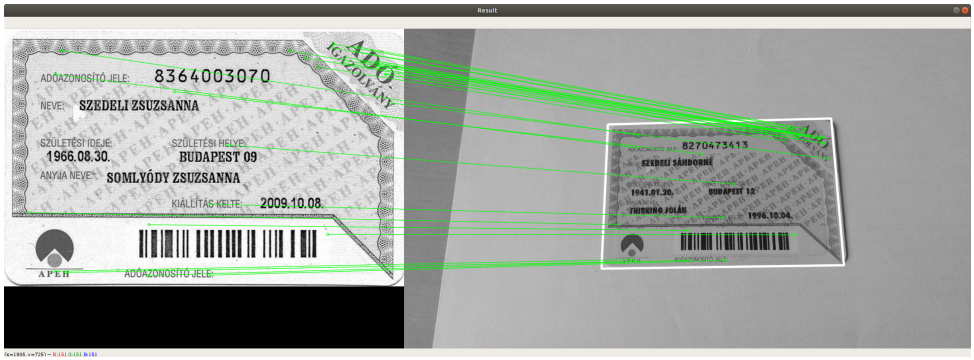
Annak tesztelésére, hogy ez milyen pontossággal működik, kipróbáltam az OpenCV által írt leírás mintakódját a saját képeimet behelyettesítve ([18]). Ennek eredményét mutatja meg a 4. ábra. Mint látható ezen, két különböző minőségű kép (szkennelt és fényképezett) között sikeresen megtalálta az ugyanolyan típusú (de szintén különböző) adóigazolványt.

Miután ez a pár teszt képpel tökéletesen működött, módosítottam a kódot úgy, hogy a megtalált igazolványt visszatranszformálja a mintakép helyére, melyet a 7. kód valósít meg.

7. Kód. Igazolvány keresés, transzformálás

```
import numpy as np
import cv2

MIN_MATCH_COUNT = 10
img1 = cv2.imread('...') # queryImage
img2 = cv2.imread('...') # trainImage
img2 = cv2.resize(img2, dsize=(0, 0), fx=1.0 / 4.0,
    fy=1.0 / 4.0, interpolation=cv2.INTER_LANCZOS4)
# Initiate SIFT detector
```



4. ábra. SIFT algoritmus alapú adóigazolvány keresés

```
sift = cv2.xfeatures2d.SIFT_create()
# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE,
    trees = 5)
search_params = dict(checks = 100)
flann = cv2.FlannBasedMatcher(index_params,
    search_params)
matches = flann.knnMatch(des1,des2,k=2)
# store all the good matches as per Lowe's ratio test.
good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append(m)

if len(good) > MIN_MATCH_COUNT:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in
        good]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in
        good]).reshape(-1, 1, 2)

    M, mask = cv2.findHomography(dst_pts, src_pts,
        cv2.RANSAC, 5.0)
    img2 = cv2.warpPerspective(img2, M, (img1.shape[1],
        img1.shape[0]))
    cv2.imshow("Transform", img2)
else:
    print("Not enough matches are found -
        {}/{}".format(len(good), MIN_MATCH_COUNT))
```

```
    matchesMask = None
```

Ennek eredményét a 5. ábra mutatja. A bal oldalon szerepel az eredeti fénykép, a jobb oldalon pedig a képből visszatranszformált igazolványkép. Mint látható, egész rossz körülmények között is sikeresen megtalálta az igazolványt. Az 1. kép sötétebb fényviszonyok mellett készült, a 2. képnél rosszul fókuszálva, szokatlan szögből lett fényképezve a kártya, míg a 3. képnél szándékosan lemaradt az igazolvány egyik sarka. Ezen hibák mellett is viszonylag pontosan sikerült mindhárom esetben megtalálni a képet.

5. ábra. SIFT algoritmus alapú adóigazolvány keresés és visszatranszformálás

Mivel láthatóan jól működik a legkülönbözőbb képekre és beállításokra, emellett az algoritmus mellett döntöttem. Ahhoz, hogy ezt felhasználhassam a szoftverben, még két helyen kellett módosítanom az algoritmust:

1. Az OpenCV által megvalósított SIFT algoritmus egyik paraméterén keresztül támogatja, hogy a bemenő kép bizonyos részeit kimaszkoljuk, vagyis ott nem keres érdekes pontokat. Ez ideális arra, hogy a minta igazolványképen a mezőket kitakarjuk, így ezek biztos nem fognak bezavarni a detektálásban.

2. A mintaigazolványra elég csak egyszer előre kiszámolni a SIFT algoritmus eredményét, hisz (mivel a mintakép nem változik), ez mindig ugyanazt a leíráslistát fogja eredményezni.

Ezeket figyelembe véve a mintaigazolvány leírólistájának kiszámolását a 8. kód, míg a visszaolvasását a 9. kód végzi.

8. Kód. Mintaigazolvány leíró fájl létrehozása

```
import numpy as np
import cv2

img1 = cv2.imread('...') # queryImage
sift = cv2.xfeatures2d.SIFT_create()
mask = __get_template_old_image_mask(img1.shape[1],
    img1.shape[0]) # Get mask for image
kp, des = sift.detectAndCompute(img1, mask)

index = []
for point in kp:
    temp = (point.pt, point.size, point.angle,
        point.response, point.octave, point.class_id)
    index.append(temp)

pickle.dump((index, des),
    open("data/template_old_camera.id_data", "wb"))
```

9. Kód. Mintaigazolvány leíró fájl beolvasása

```
import numpy as np
import cv2

(kp_array, old_card_des1) =
    pickle.load(open("data/template_old_camera.id_data",
    "rb"))
old_card_kp1 = []
for point in kp_array:
    temp = cv2.KeyPoint(x=point[0][0], y=point[0][1],
        _size=point[1], _angle=point[2],
        _response=point[3], _octave=point[4],
        _class_id=point[5])
    old_card_kp1.append(temp)
```

Ezzel sikerült a régi adóigazolványt beforgatni a mintaigazolvány helyére, de még nincs megoldva az új igazolvány, illetve a kettő megkülönböztetése. Szerencsére erre az algoritmus ad egy nagyon egyszerű megoldást. Először megpróbáljuk a régi mintakártyát megkeresni a beküldött képen. Ha ez sikerül, akkor megtaláltuk a kártyát, és tudjuk, hogy régi kártya volt rajta. Ha nem sikerült, újra megpróbáljuk, de most egy új mintakártyával. Ha sikerült, akkor megfelelően be van forgatva, és tudjuk, hogy új adóigazolvány lett beküldve, különben pedig feltételezhetjük, hogy nincsen kártya a képen.

Ez az esetek nagy részében hibátlanul beforgatta az adóigazolványt, voltak azonban olyan esetek (elsősorban a régi típusú kártyáknál), melyeknél vagy nem találta meg a kártyát, vagy hibásan detektálta annak a helyét. Ennek a javítására két megoldást vezettem be:

1. A régi kártyák megtalálásának javítása érdekében felvettem a mintaképek közé még egy régi típusú, az eredetinél régebbi kártyát. Ha az első beolvasás nem sikerült, újra megpróbálom ezzel az új mintakártyával is.

2. A hibás kártyatalálat detektálásának kikerülése végett a beforgatott kártya azon részén, ahol a vonalkódnak kellett elhelyezkedni, lefuttatok egy vonalkód-beolvasó algoritmust. Ha talál vonalkódot, akkor feltételezem, hogy sikeresen megtalálta a kártyát, különben sikertelen volt.

4.2.2. Mezők beolvasása

Miután a képen megtaláltuk az adóigazolványt, és sikeresen beforgattuk azt a mintakép példájára, jöhet a mezők beolvasása a kártyáról.

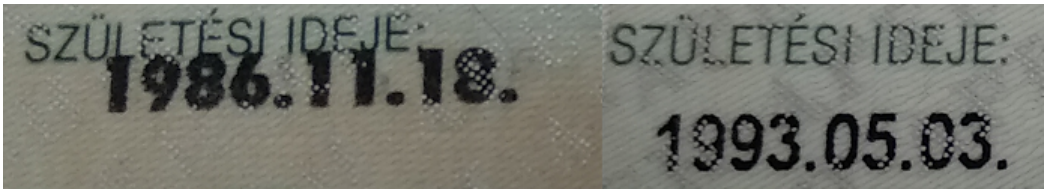
Első lépésként a már említett *pyzbar* könyvtárral beolvassuk a vonalkód értékét, majd ebből visszaszámoljuk a születési dátumot. Ha bármelyik mező eltér, biztosak lehetünk benne, hogy a felhasználó rosszul adta meg ezeket az adatokat, ezért ilyenkor egyértelműen elutasításra kerül a kérés.

Ezután jöhetnek a szöveges mezők. Először arra gondoltam, hogy mivel az adott mezők fix helyen vannak elhelyezve, elég ezeket a fix mezőket kivágni a képből, és átadni a Tesseractnak. A régi igazolványoknál sajnos bonyolódik a helyzet, itt ugyanis az anyja neve két helyen is szerepelhet. Ennek feloldására mindkét helyen megpróbáljuk beolvasni a nevet, és ha bármelyiknél egyezést találunk, akkor elfogadjuk az adatot.

Ez a tesztkártyákra jól működött, később azonban kiderült, hogy a régi adóigazolványok esetén akár egész szélsőségesen is el tudnak csúszni a mezők értékei. Erre mutat példát a 6. ábra.

Itt azt találtam, hogy az ilyen nagy mértékű eltérések miatt a fix mező kivágás a régi kártyáknál nem működik olyan jól, akkor se, ha ennek a mező méretének a lehető legnagyobb teret választom. Ezt két lépésben valósítottam meg.

Első lépésként detektálom, hol vannak az adóigazolványon szövegek. Itt először egy homályosítás után elvégzek az egész kártyán egy 5 csoportos klaszterezést, majd a legsötétebb klasztert feketére festem, míg a többit fehérre. Ezt mutatja be a 10. kód.



6. ábra. Régi adóigazolvány mező elcsúszás

10. Kód. Szöveg detektálás: klaszterezés

```
image_original = cv2.GaussianBlur(image, (3, 3),
    sigmaX=0.7)
image = image_original[40:580, 40:850, :]
Z = image.reshape((-1, 3))
Z = np.float32(Z)
criteria = (cv2.TERM_CRITERIA_EPS +
    cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 5
ret, label, center = cv2.kmeans(Z, K, None, criteria,
    10, cv2.KMEANS_RANDOM_CENTERS)
center = np.uint8(center)
minIndex = 0
for j in range(1, K):
    if (sum(center[j]) < sum(center[minIndex])):
        minIndex = j

for j in range(0, K):
    if minIndex == j:
        center[j][:] = 0
    else:
        center[j][:] = 255

res = center[label.flatten()]
image = res.reshape((image.shape))

if len(image.shape)==3:
    image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
image = cv2.copyMakeBorder(image, 40, 0, 40, 0,
    cv2.BORDER_CONSTANT, value=255)
```

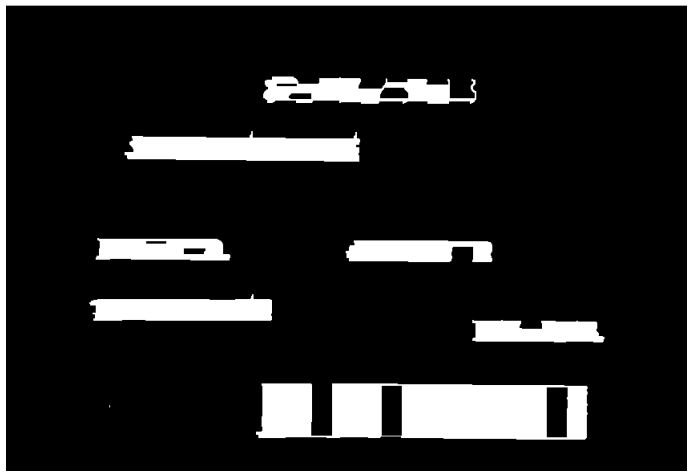
A klaszterezésnél figyelni kell, hogy ne a teljes kártyán végezzem azt el, ilyenkor ugyanis a kép sarkában lévő háttér bezavarhat a klaszterezés eredményébe. Ennek érdekében a klaszterezés során minden oldalról 40 pixelnyit belevágok a képbe, amit a kód végén egy ugyanilyen vastag fehér kerettel pótlok.

Ezt követően, mivel tudom, hogy minden szöveges mező, melyet keresek, vízszintesen helyezkedik el, ilyen irányban egy élkereséssel a betűk szélei ki fognak

ugrani. Ezeket a kiugrásokat egy zárás művelettel elsősorban vízszintes irányban elkenek, így ennek végére a szöveges mezők helyein fehér, körülbelül téglalapok lesznek. Ezt mutatja be a 11. kód, illetve ennek az eredménye látható a 7. ábrán.

11. Kód. Szöveg detektálás: élkeresés + zárás

```
ele_size = (25, 3)
img = cv2.Sobel(image,cv2.CV_8U,1,0)
img_threshold =
    cv2.threshold(img,0,255,cv2.THRESH_BINARY)
element =
    cv2.getStructuringElement(cv2.MORPH_RECT,ele_size)
img = cv2.morphologyEx
    (img_threshold[1],cv2.MORPH_CLOSE,element)
```



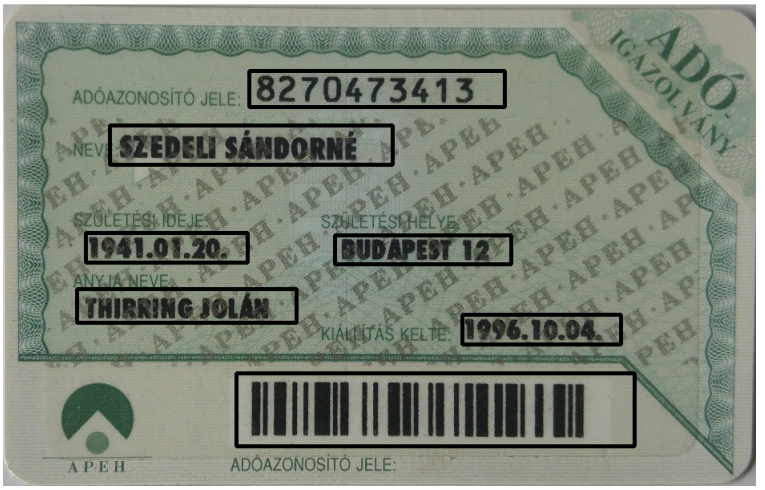
7. ábra. Szöveg detektálás - élkeresés + zárás

Végül kontúrkereséssel, majd a kontúrok befoglaló téglalapját véve megkapjuk azokat a téglalapokat, melyekben (többek között) a szöveges mezők is vannak. Ennek kódja látható a 12. kódon, illetve a detektált mezők téglalapjai egy példaképen a 8. ábra mutatja.

12. Kód. Szöveg detektálás: kontúrkeresés

```
Rect_all = [cv2.boundingRect(i) for i in contours[1] if
    i.shape[0] > 40]
Rect = [x for x in Rect_all if x[2] >= 40 and 20 <= x[3]
    <= 90]
```

```
RectP = [(int(i[0]-i[2]*0.05), int(i[1]-i[3]*0.15),
    int(i[0]+i[2]*1.15), int(i[1]+i[3]*1.15)) for i in
    Rect]
```



8. ábra. Szöveg detektálás végeredmény

Ezután hátramaradó feladat, hogy a téglalapok listájából kiszűrjük azokat a téglalapokat, melyek az adott adat mezőjéhez tartoznak. Ehhez minden adathoz felvettem több tesztpontot, majd sorban végigmentem a téglalapokon, és amelyik téglalapban benne volt legalább egy tesztpont, hozzárendeltem a megfelelő mezőhöz.

Végül egy utolsó lépésben összeolvasztottam a szétszakadó mezőket. Ritkán, de előfordul, hogy egy adott mező két, esetleg három különálló téglalapba kerül. Ekkor a tesztpontok alapján megtalált téglalapokhoz megvizsgálom, hogy van-e a közelükben velük körülbelül egy magasságban egy másik téglalap is, és ha igen, a kettőt összeolvasztom egy téglalappá.

Ennek az utolsó két lépésnek az implementálása nem jelent nehézséget, ezért külön itt most nem emelem ki őket. Itt még megjegyezném, hogy ezt a szöveg detektálást csak a régi kártyáknál szükséges lefuttatni, az új kártyákon az egyes mezők eléggé pontosan vannak elhelyezve ahhoz, hogy itt a fix kivágott téglalapokban mindig benne legyen a keresett szöveg is.

A tesztek alapján sajnos a Tesseract nem volt elég pontos ahhoz, hogy csak így egy beolvasás után minden esetben pontosan visszaadja a kártyán található szöveget. Ennek javításáért különböző szűréseket végzünk el a kártyákon. Ezek a szűrések az alábbiak valamilyen keverékéből jönnek elő:

1. *Binarizálás*: Ebben a lépésben a színes képet először átalakítjuk szürkeárnyalatossá, majd azt valamilyen módon binarizáljuk (fekete-fehér képpé alakítjuk). Ez általában úgy történik, hogy minden pixel, mely fényessége egy t határérték alatt van, fekete, míg a többi fehér színű lesz. Ez lehet

egy, az egész képre jellemző érték, illetve lehet az adott pixel környezetétől függő (adaptív érték). A binarizálásra több algoritmus létezik, én az Otsu féle binarizálást választottam, mert ez jól alkalmazkodik a különböző fényviszonyokhoz. Ez az algoritmus úgy választja meg a t határ értékét, hogy a következő értéket maximalizálja:

$$\theta_b^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$$

ahol ω(t) jelöli a t határértékkel vett fekete (0), illetve fehér (1) pixelek számának arányát az összeshez képest, a µ(t) érték pedig hasonlóan a fekete, illetve fehér csoporthoz tartozó átlag-fényerősségérték.

2. *Klaszterezés*: A klaszterezés gyakorlatilag a binarizálás kiterjesztésének tekinthető. A kép minden egyes pixeljét a színe alapján besorol egy csoportba, majd a pixel az adott csoport színeinek átlagát kapja meg. Én a K-Means algoritmust használtam, melynek egyik paramétere a csoportok száma. Ezután a legsötétebb csoportot véve megkapjuk a fekete betűket. Optimális a régi kártyákon a háttérben lévő *APEH* felirat eltüntetésére, azok ugyanis egy másik, önálló kategóriába fognak kerülni.

3. *Homályosítás*: A lépés során a kép bizonyos mértékben homályosodik. Ideális a különböző zajok eltüntetésére, túlzott használata azonban a kép fontos részeit is eltüntetheti. Az algoritmus során Gauss-blurt használtam, 3x3-as kernelmérettel, illetve 0.7-es szigma értékkel.

Ezekre a szűrésekre mutat példát a 9. ábra. Az utolsó két kép egy 5 csoportos klaszterezés eredményét mutatja. Ebből az első kép megtartotta a csoportok színeit, a második képben pedig a fent már említett műveletet elvégezve a legsötétebb színt feketével, a többit pedig fehérrel helyettesítettem.

Ezeknek a szűréseknek valamilyen kombinációját egy eset kivételével mindig a kivágott mezőre alkalmazom, ezzel is gyorsítva a végrehajtást.

Annak kiderítésére, hogy milyen szűrőkombinációk a legoptimálisabbak, létrehoztam egy kis teszt szkriptet, mely rögzítette, hogy a különböző esetekben hány mezőt tudott a Tesseract jól beolvasni. Ezt elindítva lefuttattam 30 alapvetően régi adóigazolványon (mindegyiknél csak a szükséges 4 mezőt olvastam be). Ennek eredményeit a 1. táblázat mutajta.

Ennek tudatában a beolvasás során a következő logikát követem:

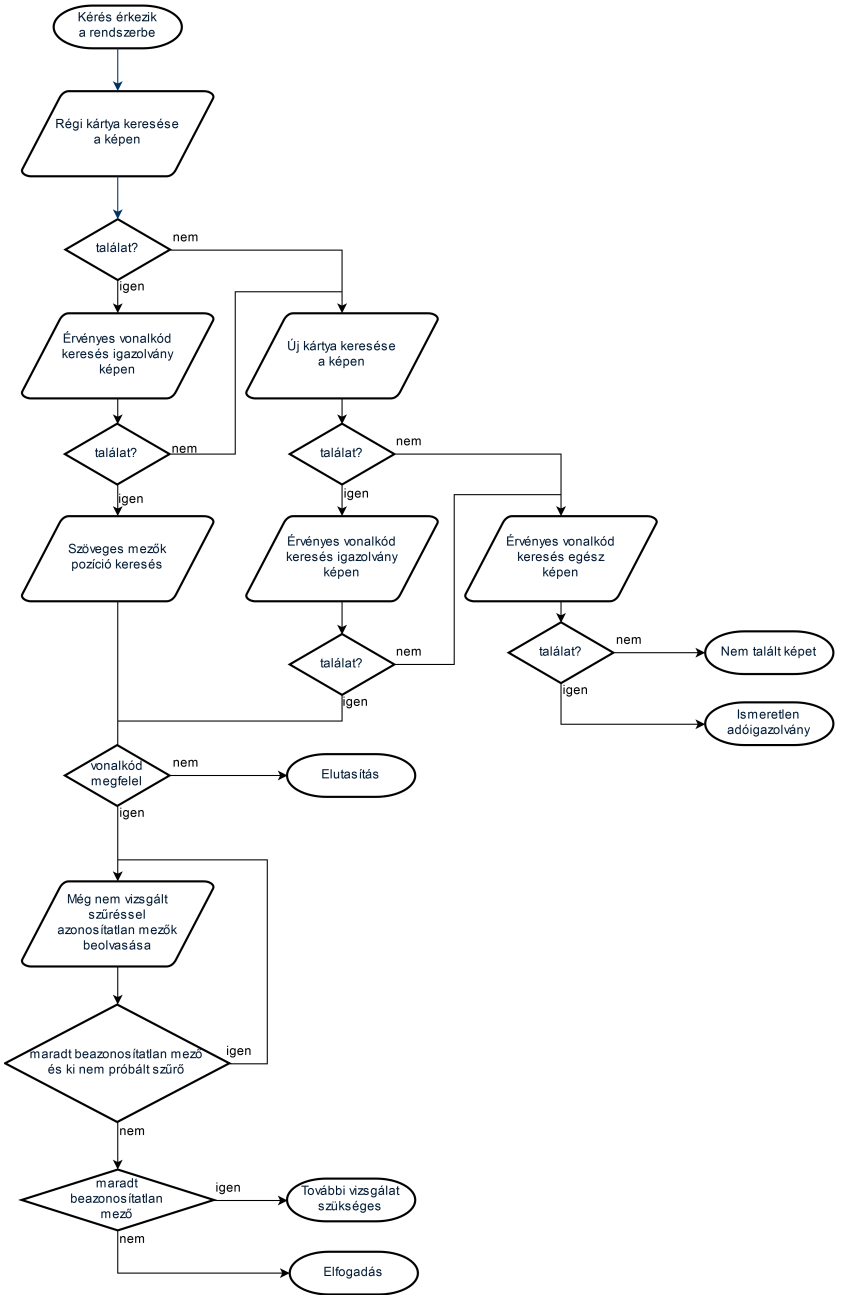
1. Kezdetben egy mező sincs sikeresen leellenőrizve a kártyáról.
2. Megpróbálom a legsikeresebb szűrőkombinációval beolvasni a 4 mezőt.
3. Ha van olyan mező, melyet nem sikerült beolvasni, akkor csak ezeken megpróbálom a következő legjobb szűrőt.
4. Ha sikerült az összes mezőt sikeresen leellenőrizni, elfogadjuk az adatokat.
5. Különben jelezzük, hogy ezt a kártyát szükséges extra ellenőrzéseknek alávetni.

A 10. ábra ismerteti a megvalósított algoritmus főbb lépéseinek sorrendjeit.

1. táblázat. Különböző szűrőkombinációk beolvasási eredménye

Szűrő	Sikeres beolvasás
homályosítás + 5 csoportos klaszterezés egész képen	85
homályosítás + 5 csoportos klaszterezés	79
homályosítás + binarizálás	79
homályosítás	77
binarizálás	75
5 csoportos klaszterezés	71
6 csoportos klaszterezés	66
Eredeti kép	65

9. ábra. Különböző szűrések eredménye egy adóigazolványon



10. ábra. Folyamatábra a szoftver magas szintű működéséről

5. Optimalizálás

Az eddig leírtak alapján sikerült megvalósítani egy működő szoftvert, azonban ez a nagy mennyiségű beérkező képek kezelésére még nem volt alkalmas. Ez a fejezet a különböző technikákat mutatja be, mellyel a teljesítményt sikerült növelni.

Első lépésként megmértem az algoritmus egyes lépéseiben eltöltött időt. Ez alapján egyértelművé vált, hogy (nem meglepően) a Tesseract futása tart a legtovább. Emellett a másik számottevő komponens az igazolványdetektálás során a *SIFT* algoritmus, illetve annak testreszabása jelentette.

5.1. OpenCV változtatások

Az OpenCV kódjánál a *SIFT* algoritmus gyorsaságán kellett javítani. Ezt a bemeneti kép skálázásával oldottam meg, melyet a 13. kód mutat.

13. Kód. Szöveg detektálás: élkeresés + zárás

```
img_height, img_width = input_img.shape[0:2]
scale = (target_width / img_width)
img2 = cv2.resize(input_img, (0, 0), fx=scale, fy=scale,
    interpolation=cv2.INTER_LANCZOS4)

# ...
# Later, at the inverse-transformation, scale back the
    points to the original positions
dst_pts /= scale
M, mask = cv2.findHomography(dst_pts, src_pts,
    cv2.RANSAC, 5.0)
img2 = cv2.warpPerspective(input_img, M,
    card_sizes[card_type])
```

Itt a skálázott kép szélessége egy paramétere a függvénynek. Tapasztalatok alapján abban az esetben, amikor az igazolvány a kép nagy részét töltötte ki, 320 pixel széles skálázott képen is képes volt megtalálni az igazolványt. A gyakorlatban feltöltött képek esetén ritkán teljesült ez, ezért ezt a méretet megnöveltem 640 pixelre.

5.2. Tesseract futás optimalizálás

A szoftver az idő nagy részét a Tesseract futtatásával töltötte, ezért a fő optimalizálásnak itt kellett történnie. Ezt több lépésben tettem meg.

Ahogyan már írtam, a Tesseract 4. verziója óta elérhető egy neurális háló is a szövegek beolvasására. Ennek a hálónak a működését, illetve a súlyozásokat tartalmazza a *.tessdata* fájl, mely minden nyelvre különbözik, ezzel is alkalmazkodva az adott nyelv sajátosságaihoz. Egy nyelven belül is több különböző verzió elérhető, ebből az egyik a pontosság kárára a gyorsaságra fókuszál [19]. A tapasztalatok alapján valójában ez nem ront számottevően a beolvasás

hatékonyságán, főleg ha figyelembe vesszük, hogy több különböző szűréssel is végrehajtjuk a beolvasást.

Egy másik fontos optimalizáció volt a Tesseract hívások számának csökkentése. A megvalósított szoftver minden egyes mezőre elindított egy Tesseract folyamatot, majd megvárta, hogy ez befejeződjön. Tekintve, hogy egy igazolványon 4 mezőt kell beolvasni, ez legjobb esetben is 4 folyamatindítást jelentett, az újrapróbálások miatt viszont legrosszabb esetben ez a szám akár 64 is lehetett.

A Tesseract ad arra lehetőséget, hogy egy folyamatindítás során több képet olvasson be, majd ennek eredményét egy táblázatban adja vissza, melyben az oldalszám segítségével lehet megkülönböztetni őket. Ehhez a képek címét be kell írni egy fájlba, majd ennek a fájlnak az elérhetőségét kell átadni a Tesseractnak. Ezt a lépést végzi el a 14. kód, míg az eredmény szétválogatását a 15. kód végzi.

14. Kód. Tesseract több kép beolvasása egyszerre

```
def run_multiple_and_get_output(images,
                                extension,
                                extension_configs,
                                lang=None,
                                config='',
                                nice=0,
                                return_bytes=False):

    image_names = []
    for image in images:
        temp_name, img_extension = save_image(image)
        image_names.append(temp_name + os.extsep +
            img_extension)

    temp_list_name = tempfile.mktemp(prefix='tess_')
    try:
        with open(temp_list_name, 'w') as temp_list_file:
            for image_path in image_names:
                temp_list_file.write("%s\n" % image_path)

        kwargs = {
            'input_filename': temp_list_name,
            'output_filename_base': temp_list_name +
                '_out',
            'extension': extension,
            'extension_configs': extension_configs,
            'lang': lang,
            'config': config,
            'nice': nice
        }
```

```
        run_tesseract(**kwargs)
        filename = kwargs['output_filename_base'] +
            os.extsep + extension
        with open(filename, 'rb') as output_file:
            if return_bytes:
                return output_file.read()
            return
                output_file.read().decode('utf-8').strip()
    finally:
        for image_name in image_names:
            cleanup(image_name)
        cleanup(temp_list_name)
```

15. Kód. Tesseract több kép eredményének szétválogatása

```
read_values = []
lines = read_str.split("\n")#
for line in lines:
    read_values.append(line.split("\t"))

page_index = read_values[0].index("page_num")
base_row = read_values.pop(0)
ret_list = [[]]
ret_list[0].append(base_row)
prev_page = '1'
while len(read_values) != 0:
    next_row = read_values.pop(0)
    if next_row[page_index] != prev_page:
        ret_list.append([])
        ret_list[-1].append(base_row)
        prev_page = next_row[page_index]

    ret_list[-1].append(next_row)
```

Ezen optimalizációk kombinálásával a gépemen egy kép ellenőrzését sikerült átlagosan 3 másodperc alatt elvégezni, amit már sikeresnek tartottunk a projekt során. A rendszer kiélesítése során viszont hamar problémába futott a megrendelő. Valószínűleg nem volt terhelés-tesztelve ez a része a rendszernek (mi pedig nem kaptunk arról adatot, mennyi kérést terveznek a rendszer felé küldeni), ezért első körben túlterhelődött a rendszer.

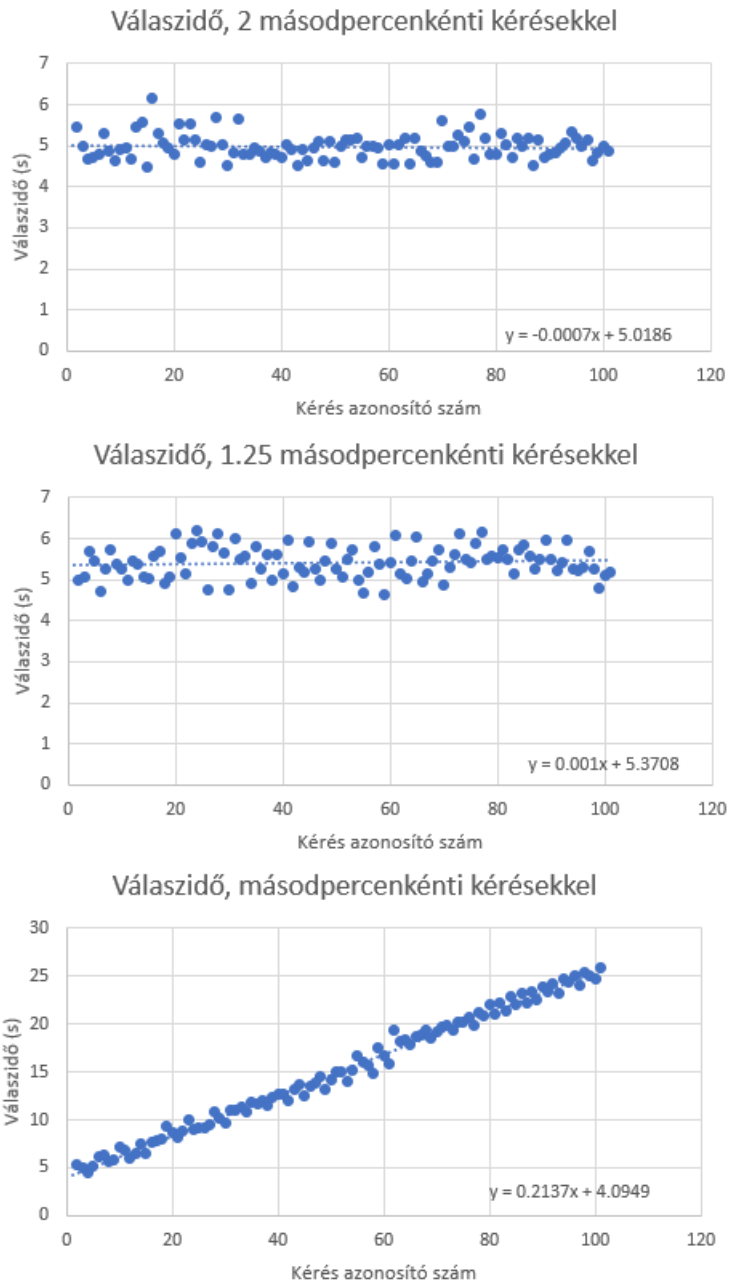
A probléma természetesen abból eredt, hogy a karakterfelismerés, illetve képfeldolgozás nagyon CPU igényes feladat, ezek már így is képesek teljesen kihasználni a gépemben található CPU-t. A probléma gyors kezelésére a szoftver 6 nagyon erős, 16 magos szervergépre került áthelyezésre, mely képes volt már kielégíteni az igényeket, de óriási plusz költséget is jelentett a megrendelőnek, ezért ezen a téren is mindenképp kellett javítani.

Az egyszerű megoldás az lenne, hogy engedünk egyszerre több Tesseractot futni, és rábízzuk az operációs rendszerre, hogy ütemezze megfelelően a több folyamatot. Sajnos a Tesseract a több szál szinkronizálása, illetve a számításigényes feladatai miatt teljesen kifagy, ha egyszerre több példányt próbálunk meg egy CPU magon futtatni belőle [20]. Bár konkrétan nem tudjuk befolyásolni könnyen, melyik processzormagon fusson, azt feltételezhetjük, hogyha van rá elég mag, akkor az operációs rendszer nem fog ugyanarra a magra több példányt ütemezni. A Tesseract legfeljebb 4 párhuzamos szálat tud futtatni egyszerre, így tehát egy 16 magos gép esetén párhuzamosan 4 Tesseract folyamat gond nélkül tud futni, ami így már négyszeres teljesítménynövekedés.

Természetesen mivel a Tesseract párhuzamosan több szálat is futtat, meg kell oldani közöttük a szinkronizációt, ami helyzettől függően értékes processzor-időt is jelenthet. Ekkor felmerült bennem, hogy mi történne, ha a Tesseractot bekorlátoznám, hogy összesen egy szálon fusson egy példány. Szerencsére erre létezik egy könnyen állítható környezeti változó, mely pont ezt határozza meg. Egy szál esetén automatikusan átugrik minden szinkronizációs tevékenységet, így előfordulhat, hogy több képet tudunk párhuzamosan egységidő alatt feldolgozni. Több teszt után arra jutottam, hogy bár egy kép ellenőrzési ideje megnőtt 5-6 másodpercre, cserébe a 4 magon párhuzamosan 4 képet lehet feldolgozni, mely így egy körülbelül kétszeres teljesítménynövekedést jelent.

Ezeket a teszteket mutatja be a 11. ábra. Jól látható rajta, hogy ameddig a kérések gyakorisága nem érte el az 1.25 kérés/másodpercet, addig a 4 magos rendszer ki tudta szolgálni ezeket. Ez alatt viszont nem volt kapacitás már több kép feldolgozására, ezért a később érkező kéréseknek egyre többet és többet kellett várnia.

Ezek alkalmazásával kijelenthető tehát, hogy egy 16 magos gép esetén az eredeti (nem optimalizált) eset 3 másodperc/kép idejét sikerült 5-6 másodperc/16 kép-re javítani, amely kielégítette a kitűzött performancia igényeket és a megrendelő is elégedett volt vele.



11. ábra. Különböző szűrések eredménye egy adóigazolványon

6. Jövőbeli lehetőségek

Zárásként bemutatom, milyen jövőbeli fejlesztések képzelhetőek el a projekttel kapcsolatban.

6.1. Az algoritmus általánosítása

Az algoritmus és annak szoftveres implementációja úgy lett tervezve és megvalósítva, hogy könnyen általánosítható legyen a jelenlegi adóigazolványok kezelésén túl.

Ahhoz, hogy a jövőben tetszőleges kártyára be lehessen konfigurálni a megoldást, szükséges lenne az adóigazolvány-specifikus részeket lecserélni külsőleg megadott paraméterekre. Ide tartoznak többek között az alábbiak:

- A kártya detektálásához szükséges különleges pontok leíró listája.
- A kártyán található mezők pontos helyei.
- A kártyán található mezők tartalmának formaisága.

Ezek beállításához ajánlott lenne egy könnyen használható felhasználó felületet is tervezni (GUI) melyen keresztül ezeket be lehetne állítani.

6.2. Az OCR szoftver okosítása

Jelenleg a Tesseract-OCR csak az általa legvalószínűbbnek tekintett opciót adja vissza egy beolvasás során. Ez a mi helyzetünkben nem hatékony, hiszen tudjuk előre, hogy milyen szövegnek kéne szerepelnie. Sokkal hatékonyabb lenne, ha a Tesseract-OCR képes lenne például egy valószínűséget visszaadni, hogy az adott képen mekkora valószínűséggel található meg a bemenetként kapott szöveg.

Amennyiben ez nem oldható meg, akkor legalább azt bele lehetne fejleszteni, hogy karakterenként több lehetséges karaktert adjon vissza. Ezt a funkciót a régebbi verziókban sikerült megtalálnom, a legfrissebb verzióban viszont valamiért már hiányzik.

Ha ezek nem megoldhatóak, vagy jelentős nehézségekbe ütközik, érdemes lehet megpróbálkozni egy saját OCR szoftver fejlesztésével. A neurális hálók óriási tempóban voltak képesek fejlődni az elmúlt években, így ezek megteremtették annak a lehetőségét, hogy elég tanító kép segítségével automatizáltan lehessen egy ilyen rendszert betanítani. Bár valószínűleg nem lenne olyan pontos, mint a Tesseract, hatékonyabban ki tudnánk használni a megadott információt, így lehetséges, hogy végeredményben egy pontosabb rendszert kapnánk.

7. Összefoglaló

Dolgozatomban bemutattam egy adóigazolvány ellenőrző megoldás felépítését.

A Magyar Országos Horgász Szövetség egy valós igénnyel kereste meg a tanszéket. Azonosítottam és felmértem az ebben rejlő problémákat, megvizsgáltam a szakirodalmat, majd ennek tudatában egy egyedi megoldást dolgoztam ki a többféle igazolvány típus hatékony kezelésére. Optimalizáltam a rendszert nagy mennyiségű kártya kezelésére, melyhez skálázódó architektúrát terveztem. Ez a rendszer élesbe lett állítva, folyamatosan szolgálja ki az igényeket egy automatizált rendszer részeként.

Megoldásomban bemutattam a mintaillesztésre kiválóan alkalmas *SIFT* algoritmust, mellyel képes voltam a különböző igazolvány típusokat detektálni, illetve egy sablon helyére forgatni őket.

Adtam egy megoldást a nem fix szövegmező megtalálására, melyben él, illetve kontúrkereséssel könnyedén megkaphattuk a mezők befoglaló téglalapját.

Adtam egy módszert, mellyel a Tesseract pontosságán lehetett javítani, különböző szűrések használatával. Kimértem, milyen szűrések adják a legjobb eredményeket a felsoroltak közül.

Bemutattam, hogyan lehet ezt a megoldást optimalizálni nagyobb mennyiségű kérés teljesítésére.

Végül összefoglaltam, milyen jövőbeli lehetőségeket látok a projekttel kapcsolatban. Ide tartozik a megoldás kiterjesztése általános igazolványok beolvasására, illetve az OCR komponens továbbfejlesztése, mellyel képesek lennénk hatékonyabban ellenőrizni a felhasználó által megadott adatokat.

8. Hivatkozások

[1] N. Faruqui, *Open Source Computer Vision for Beginners: Learn OpenCV using C++ in fastest possible way (2nd Edition).*

[2] A. Kaehler, *Learning OpenCV: Computer Vision in C++ with the OpenCV Library.*

[3] Python, „Python documentation." https://docs.python.org/3.6/, October 2018.

[4] T. P. H. D. M. C. Ivan Culjak, David Abram, „A brief introduction to opencv." http://mipro-proceedings.com/sites/mipro-proceedings.com/files/upload/sp/sp$_0$08.*pdf*, *September*2012.

[5] OpenCV, „Opencv documentation." https://docs.opencv.org/3.0-beta/index.html, October 2018.

[6] R. Hess, „An open-source sift library." https://robwhess.github.io/opensift/siftlib-acmmm10.pdf, November 2010.

[7] K. K. G. B. Ethan Rublee, Vincent Rabaud, „Orb: An efficient alternative to sift or surf." https://docs.opencv.org/3.0-beta/index.html, November 2011.

[8] M. T. C. B. Marc Petter, Victor Fragoso, „Automatic text detection for mobile augmented reality translation." https://www.computer.org/csdl/proceedings/iccvw/2011/0063/00/06130221.pdf, November 2011.

[9] Google, „An overview of the tesseract ocr engine." https://research.google.com/pubs/archive/33418.pdf, September 2007.

[10] Google, „Tesseract wiki." https://github.com/tesseract-ocr/tesseract/wiki, October 2018.

[11] K. Sajjad, „Automatic license plate recognition using python and opencv." https://pdfs.semanticscholar.org/bddf/1200eb17f239e4dce2a9cec938eb8cf305f5.pdf, March 2010.

[12] N. H. Michael Ryan, „An examination of character recognition on id card using template matching approach." https://www.sciencedirect.com/science/article/pii/S1877050915020633, 2015.

[13] S. W. H. H. Y. Nooral S. Abtahi, Grady C. Shumate, „Method and apparatus for confirming the identity of an individual presenting an identification card." https://patents.google.com/patent/US5509083A/en, June 1994.

[14] J. C. G. D. F. S. G. B. W. V. R. Rodolfo Valiente, Marcelo T. Sadaike, „A process for text recognition of generic identification documents over cloud computing." https://search.proquest.com/openview/1d6c43922b81d4fb2a3e1675c8b3764e/1?pq-origsite=gscholarcbl=1976345, 2016.

[15] D. Pásztor, „Taxid software.” https://github.com/danim1130/TaxID, October 2018.

[16] Alex, „Zbar bar code reader.” https://github.com/NaturalHistoryMuseum/pyzbar, 2018.

[17] J. Brown, „Zbar bar code reader.” http://zbar.sourceforge.net/, 2007.

[18] OpenCV, „Feature matching + homography to find objects.” https://docs.opencv.org/3.0-beta/doc/py$_tutorials/py_feature2d/py_feature_homography/py_feature_homography.htmlpy-feature-homography, October$2018.

[19] Google, „Tesseract : Fast tessdata.” https://github.com/tesseract-ocr/tessdata$_fast, October$2018.

[20] Google, „Tesseract locks up when multiple programs attempt to process.” https://github.com/tesseract-ocr/tesseract/issues/984, October 2018.