Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Marosi Márk Dániel

2023

Szegedi Tudományegyetem

Informatikai Intézet

Domain specifikus szöveg feldolgozása kép alapú dokumentumokon

Szakdolgozat

Készítette:
Marosi Márk Dániel
Gazdaságinformatikus szakos hallgató

Belső Témavezető:
Janurik Viktor Bálint
Tanszéki mérnök

Külső Témavezető:
Gyikó Richárd
CTO

Szeged
2023

Feladatkiírás

A digitalizáció és az automatizáció terjedésével egyre nagyobb az igény olyan programokra, melyek kép alapú dokumentumokról beolvasott szöveget képesek domain függően feldolgozni és osztályozni predikciók, szövegkörnyezet és a dokumentumon elfoglalt pozíció alapján. A szakdolgozat célja egy ilyen program elkészítése egy tetszőlegesen választott domainnel.

Tartalmi összefoglaló

A téma megnevezése:
Domain specifikus szöveg feldolgozása kép alapú dokumentumokon

A megadott feladat megfogalmazása:
A szakdolgozat készítésének célja a kép alapú dokumentumok szövegfeldolgozásának témáját körbejárni, majd a gyűjtött információk felhasználásával elkészíteni egy alkalmazást, mely képes egy tetszőlegesen kiválasztott domain alapú dokumentumról szöveget kinyerni és azt feldolgozni.

A megoldási mód:
A kiválaszott domain egy személyi igazolvány kinézetére épülő, szöveg és szám típusú adatokat is tartalmazó dokumentum lett. Az adatvédelmi problémák elkerülése érdekében magam által szerkesztett, egyéni kinézettel készült igazolványokat mutatok be és használok fel a szakdolgozatban.

Alkalmazott eszközök, módszerek:
A projekt megvalósítása a legújabb, ingyenesen elérhető programcsomagok és függvénykönyvtárak (EasyOCR, OpenCV) segítségével Python programozási nyelven történt, egy általam írt algoritmussal kiegészítve, mely a kinyert adatok párosítását szolgálja.

Elért eredmények:
A fejlesztés végeredménye egy olyan asztali alkalmazás, mely a bemenetként kapott kép alapú személyi igazolványról képes szöveget kinyerni, az összetartozó adatokat párokba rendezni, majd ezt egy .csv kiterjesztésű fájlba exportálni.

Kulcsszavak:
szövegfeldolgozás, szövegkinyerés, képfeldolgozás, karakterfelismerés

Tartalomjegyzék

Bevezetés

A témaválaszatásomat az informatikai rendszerek elképesztő gyorsaságú fejlődésének gondolata alapozta meg. Egy olyan világban élünk, ahol az okos eszközök elkezdték kiváltani a manuális munkavégzési folyamatokat, vagy azokat egyszerűbbé tették. Minden nap a zsebünkben hordunk egy olyan kompakt eszközt, amely rendelkezik kamerával. Az okostelefonok kamerája, és egy erre fejlesztett applikáció együtt képes kiváltani egy hagyományos szkenner hardvert.

Emellett egyes felsőbb kategóriás telefonok már képesek fényképről felismerni szöveget, és opciót biztosítanak a szöveg kinyerésére is. Amennyiben előttünk van egy névjegykártya, és szeretnénk róla egy nevet vagy telefonszámot gyorsan kimásolni anélkül, hogy nekünk kelljen manuálisan begépelni, egyszerűen készítenünk kell róla egy fényképet, és amennyiben a telefon szöveget talál a képen, azt kimásolhatóvá és vágólapra illeszthetővé teszi a felhasználó számára, ezzel értékes időt spórova, továbbá a hibázás lehetőségét is csökkentve. Mivel ezek a szoftverek egyre elterjedtebbek, szerettem volna mélyebben belelátni ebbe a témába, és egy nemrég történt tapasztalat csak felerősítette szakmai érdeklődésem a szövegfelismerés és feldolgozás kapcsán.

Egy határátkelésnél történt, hogy a személyi igazolványomat egy kis méterű, kompakt szkennelőgépbe helyezték, és néhány másodperc alatt minden adatom, ami a személyi igazolványomon volt, a határrendészet szoftverébe került. Ez a folyamat egy olyan plusz lépést tartalmaz az előző, okostelefonos példához képest, hogy itt nem csak az igazolványon található szöveg került felismerésre és beolvasásra, mint egy nagy adathalmaz, hanem a szoftver képes volt ezt az adathalmazt megfelelően szétbontani és osztályozni, felismerte, hogy melyik adat a név, melyik adat az állampolgárság, és így tovább.

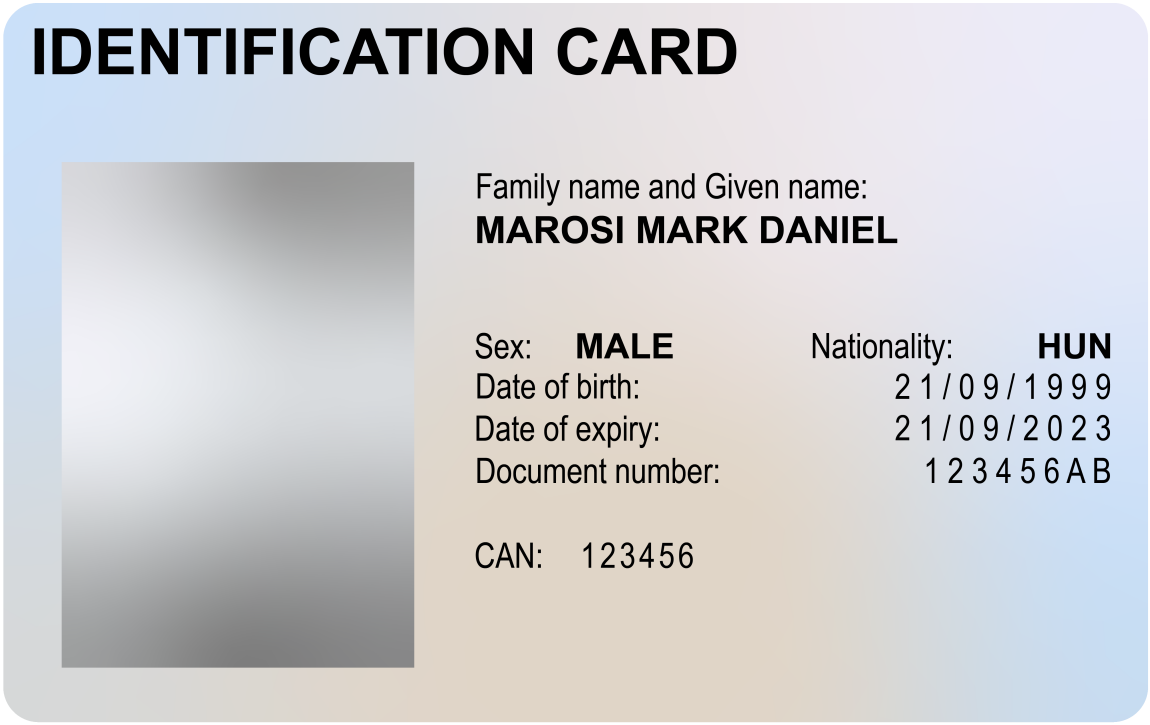
A szakdolgozatom készítése során a célom, hogy ezt a témát körbejárjam, felkutassam a legújabb technológiákat, és elkészítsek egy programot, mely lehetővé teszi a hatékony szövegfeldolgozást.

1. fejezet

Jelenlegi rendszerek

1.1. Jelenlegi rendszerek bemutatása

Manapság egy átlagfelhasználónak képről történő szövegkinyerésre telefonos vagy web alapú applikációk segítségével van lehetősége. Ez a legegyszerűbb módszer, mert könnyen kezelhető, nem igényel hozzáértést, ingyenesen használható, gyors és megbízható. A következőkben egy saját készítésű, nem valós adatokat tartalmazó igazolvány mintán fogok tesztelni két, általam választott web alapú applikációt.



1.1. ábra. igazolvány minta

Az első applikáció weboldalán [1] található egy fájlfeltöltésre alkalmas mező, ide kell tallózni a képet, melyről szeretnénk kinyerni a szöveget, ezután pedig el kell indítani a beolvasási folyamatot, mely körülbelül öt másodpercet vesz igénybe. A folyamat végén egy szövegmezőből kimásolhatóvá válik a képről kinyert szöveg. A webapplikáció az előbb bemutatott képről az alábbi szöveghalmazt nyerte ki:

> IDENTIFICATION CARD Family name and Given name: MAROSI MARK DANIEL Sex: MALE Nationality: Date of birth: Date of expiry: Document number: CAN: 123456 HUN 21/09/1999 04/11/2023 123456AB

Az eredményt megvizsgálva megállapítható, hogy a képből kinyert szöveg hibátlan, a betű és a szám alapú adatok is helyesen jelennek meg. Viszont azt is észrevehetjük, hogy az adatok rendezetlenek, kézi korrigálást igényelne, hogy az összetartozó szövegek egymás mellé kerüljenek.

A másik weboldal [2] felépítése nagyon hasonlít az előbb kipróbált oldal felépítésére, itt is egy fájl tallózására alkalmas beviteli mező segítségével tölthető fel a dokumentum, és egy szövegmezőből válik kimásolhatóvá a kinyert szöveghalmaz.

> IDENTIFICATION CARD
> Family name and Given name:
> MAROSI MARK DANIEL
> Sex: MALE
> Date of birth:
> Date of expiry:
> Document number:
> CAN: 123456
> Nationality:
> HUN
> 21/09/1999
> 21/09/2023
> 1 2 3 4 5 6 AB

1.2. Jelenlegi rendszerek hiányosságai

Tegyük fel, hogy egy olyan szituációban találjuk magunkat, ahol több személyi igazolványt kell digitalizálnunk, például egy Excel táblázatban eltárolni a kártyán olvasható adatokat. Erre három esetet fogok felvázolni. Az első a klasszikus módszer, ahol közvetlenül a személyi igazolványról írjuk át az adatokat a táblázatba, kézzel begépelve.

A második módszer az előbbiekben bemutatott webapplikáció segítségével történne, a harmadik eset pedig egy olyan program használatát feltételezi, melyet a szakdolgozatom keretében szeretnék elkészíteni.

Amennyiben az első, klasszikus módszert választjuk, kézzel kell beírnunk minden egyes adatot, betűről betűre, számról számra. A három módszer közül ez a leghosszabb és legtöbb emberi hibalehetőséget rejtő módszer. Hibázhatunk az adatok olvasásánál, azok begépelése során, valamint ott is hibázhatunk, hogy nem jó cellába visszük fel az értékeket, mert elcsúszunk valahol.

A második módszer az első módszerben felvázolt három hibalehetőségből kettőt minimálisra csökkent, ezek a beolvasási és begépelési hibák. A szoftver a beolvasást egy olyan technológiára alapozva végzi, amit a szakdolgozat következő fejezeteiben fogok részletesebben bemutatni. Fontos megjegyezni, hogy az előzőleg bemutatott weboldalak lehetőséget biztosítanak egyszerre akár több kép feltöltésére is, ezzel megkönnyítve a dolgunkat. A képről történő szövegkinyerés rendkívül pontos eredményt képes szolgáltatni, emiatt eltekinthetünk attól, hogy beolvasási hiba történjen, azaz eleve rossz adat kerüljön a táblázatba. A begépelési hibalehetőséget kiküszöböli az, hogy a szoftver a beolvasott szöveget másolhatóvá teszi, így szimplán a másolás és beillesztés műveletek segítségével vihetjük fel az adatokat a kívánt cellába. Tehát amennyiben a szövegfelismerés hibátlan volt, úgy minimálisra csökken annak az esélye is, hogy a másolás során elrontunk valamit. Az egyetlen fennálló hiba továbbra is az, hogy a másolt adatot rossz cellába illesztjük be, összekeverünk két, látszólag hasonló alakiságú adatot, például a születési időt a kártya lejárati dátumával.

A harmadik módszer mind a három hibalehetőségre megoldást biztosítana. A program futása alatt végrehajtott folyamat első része teljes mértékben ugyan úgy történne, mint a második módszer esetében: a feltöltött képről vagy képekről kinyeri a szöveget ugyanazt a technológiát használva. A program ezután nem a nyers adathalmazt adja vissza a felhasználónak, mint a második esetben, hanem tovább dolgozik azon, és a futás eredménye egy olyan .csv vagy .xlsx kiterjesztésű fájl lenne, ahol minden adatkategória (név, állampolgárság, stb.) egy oszlop lenne, és az oszlopnév alatt helyezkedne el a hozzá tartozó adat, például az állampolgárság oszlopban a HUN szöveg. Ezzel teljesen megszűnne minden emberi hibalehetőség, hiszen a teljes folyamatot elvégezné helyettünk a szoftver. Ennek a módszernek további pozitív hozadéka, hogy az adatokat képes az elvárásainknak megfelelően formázni. Például a dátumot, amely a szövegkinyerés utáni állapotban 30 06 1979 alakot vesz fel, a kiolvasás után 1979.06.30 vagy egyéb tetszőleges formára tudná alakítani, tovább csökkentve a manuális teendőket.

2. fejezet

Az OCR technológia

2.1. Az OCR működési háttere

Az OCR, azaz Optical Character Recognition [3] (Optikai Karakterfelismerés) egy olyan technológia, amely képes szinte bármilyen képről vagy digitális dokumentumról az írott vagy nyomtatott szöveget felismerni és kinyerni. Az OCR egyik legnagyobb haszna, hogy képes kiváltani a kézi adatbevitelt, jelentős időt spórolva és megszüntetve az emberi hiba lehetőségét is.

Képzeljünk el egy könyvet, amit egy hagyományos szkennelővel beszkennelünk. A folyamat eredménye digitális képek sorozata lesz, melyeket könnyedén tudunk digitális eszközeink között mozgatni, vagy akár nyomtatóval sokszorosítani, de szerkeszteni, szöveget keresni vagy kimásolni belőle nem. Ha egy olyan szkenner állna rendelkezésünkre, amelyben OCR alapú szövegfelismerés is van, akkor a könyvet könnyedén digitalizálhatnánk, és nem képeket kapnánk eredményül, hanem egy olyan szöveges dokumentumot, amely teljesen szerkeszthető és kereshető.

Az Optikai Karakterfelismerés folyamata több lépésből tevődik össze. [4] Az első lépésben egy osztályozási folyamat történik, ahol a kiválasztott képet vizsgálva a világos területeket háttérnek, a sötét területeket pedig szövegnek minősíti a program. Ebből kifolyólag az OCR pontosságát tovább javíthatjuk azzal, ha már a karakterfelismerés előtt a kiválasztott képet fekete-fehérré alakítjuk.

Második lépésben a szövegnek minősített területeken egy olyan keresés indul, amelynek célja az alfabetikus karakterek (betűk) és numerikus karakterek (számok) azonosítása. Ez történhet karakterről karakterre, vagy szavanként (karakterlánconként) is. Az azonosítás egyik leggyakoribb módszere a mintaillesztésre alapul. Ennél a módszernél egy olyan adathalmazból dolgozik az algoritmus, amely sok különböző betűtípus- és szövegkép mintát tárol egy adatbázisban úgy, mint egy sablont. Mikor a képen alakzatot próbál felismerni, összehasonlítást végez a tárolt sablonok alakzatával, és a legnagyobb

egyezést mutató karakternek fogja minősíteni a képen látható alakzatot. Ez a módszer a legpontosabb, ha a bemenet egy olyan kép, melyen ismert betűtípusokkal jelenik meg gépelt/nyomtatott szöveg, mivel a lehetségesen előforduló betűtípusok száma végtelen, és lehetetlen minden típust az adatbázisban rögzíteni.

Amennyiben kézzel írt szöveget szeretnénk digitalizálni, akkor a pontos eredmény eléréséhez egy olyan algoritmusra van szükség, amely figyelembe veszi a karakterek bizonyos jellemzőit is. Ilyen jellemzők például a betű írására használt vonalak, azok irányai, elhelyezkedései, metszéspontjai, görbületei és hurkai. Ezen tulajdonságokat az algoritmus minden felismerhető karakterről eltárolja, majd a keresett alakzatot is felbontja ezekre a jellemzőkre, és megkeresi a tárolt karakterek közül azt, amellyel a legtöbb jellemző megegyezik. Ezt a folyamatot Intelligent Character Recognition (ICR), magyarul Intelligens Karakterfelismerés névvel illetik. [5]

Utolsó lépésben a dokumentum teljes szerkezeti képének és az OCR konfigurációja függvényében a felismert karakterek önmagukban, vagy pedig szavakba, mondatokba, szövegblokkokba rendezve kerülnek az eredménybe.

2.2. Az OCR pontosságának mérése

Az előzőekben bemutattam, hogyan képes az OCR egy szöveget tartalmazó képet gépi szöveggé alakítani, de felmerülhet bennünk a kérdés, hogy mennyire pontos az eredmény amit kapunk egy ilyen konverzió során. Az OCR csupán karakterről karakterre lépve végez műveleteket, arra viszont nem képes, hogy a dokumentum teljes kontextusát felismerve megállapítsa, hogy a szöveg, amit kinyert, helyesnek bizonyul-e az adott környezetben. Emiatt az OCR gyakran hibázhat, és ezek a hibák pont a szövegfelismerés által adott előnyöket csökkentik.

Legegyszerűbben úgy határozható meg a pontosság, hogy az OCR kimeneti eredményét összehasonlítjuk a képen szereplő szöveggel. Tegyük fel, hogy a képen szereplő szöveg 100 karakterből áll. Amennyiben az OCR által adott eredményben mind a 100 karakter egyezik az eredeti szövegben szereplővel, akkor azt mondhatjuk, hogy az OCR pontossága 100%. Amennyiben 99 karaktert sikerült helyesen felismerni, úgy ez az érték 99%. Tehát egyszerű arányosítással is kiszámolható egyfajta pontosság.

Most bemutatom a két leggyakrabban használt metrikát, melyek erre a logikára épülnek. [6]

2.2.1. CER - Character Error Rate (Karakter hibaarány)

A *CER* mutató azon karakterszintű műveletek minimális számát mutatja meg, amelyek szükségesek a bemeneti szöveg hibátlan kimenetté való korrigálásához. A *CER* számításához használt képlet:

$$CER = \frac{T}{T + C} * 100$$

A képletben található *T* az OCR eredményéből érkező karakterek a bemenettel azonos karakterekre való transzformációk számát jelöli (tehát ezek olyan karakterek, melyek helytelenül lettek felismerve), *C* pedig a helyesen felismert karakterek száma. Példa:

Felismerendő szöveg: abcdefg-123
OCR kimenet: abcdef9-1Z3
Mivel a g betűt 9-es számnak állapította meg, továbbá a 2-es számot Z betűnek, így két transzformációra lesz szükségünk a korrigáláshoz, tehát T = 2.
A helyesen felismert karakterek száma kilenc (a,b,c,d,e,f,-,1,3), ezért C = 9.

$$CER = \frac{2}{2 + 9} * 100 = 18.18$$

Ebben a példában 18%-os értéket vesz fel a *CER* mutató, természetesen ez a szám minél kisebb, annál jobb.

2.2.2. WER – Word Error Rate (Szóhibaarány)

A *CER*-rel ellentétben ennél a metrikánál azt vesszük figyelembe, hogy hány szó szintű műveletre van szükség ahhoz, hogy az OCR folyamat eredménye teljesen megegyezzen a bemeneti szöveggel. Bár a *WER* érték a szavak helyességét méri, nem a betűkét, de ha belátjuk, hogy ugyanazon betűk sorozatából kapjuk a szavakat, mint amiket a CER metrikával is vizsgáltunk, akkor jogosan feltételezhetjük, hogy a *WER* és a *CER* metrikák mutatószámai jól korrelálnak egymással.

A *WER* mérésére ugyan azt a képletet használjuk, mint a *CER*-hez, de a *T* paraméter a helyes szóra történő transzformációk számát, a *C* paraméter pedig nem a helyes karakereket, hanem a teljes terjedelmében helyesen felismert szavak számát jelöli.

2.2.3. További metrikák az OCR pontosságának megállapításához:

- SER - Symbol Error Rate (Szimbólum hibaarány):
  - A SER metrika segítségével vizsgálható, hogy a szövegben szereplő szimbólumok, különböző írásjelek milyen arányban kerültek helyesen felismerésre.
- Text-Based F1 Score (Szövegalapú F1-pontszám):
  - Ez a mérőszám a felismert szöveg helyes részarányának, illetve a helyesen felismert bemeneti szöveg részarányának a harmonikus átlagát számolja. [7]
- Keystroke Saving (Billentyűleütés megtakarítás):
  - Azt méri, hogy ha egy kézi bevitelen alapuló folyamatot egy OCR alapú rendszerrel váltunk ki, akkor hány billentyűleütést spórolunk meg.

Fontos megjegyezni, hogy az előbb bemutatott metrikák nem adnak minden esetben valós képet a különböző OCR modellek működéséről, hiszen a beolvasott dokumentumok minősége, valamint a tény, hogy kézírást vagy nyomtatott szöveget adunk bemenetként mind erősen befolyásoló tényezők az OCR kimenetének helyességében.

2.3. Az OCR pontosságának javítása bemeneti oldalon

Az előbbiekben bemutattam, hogyan mérhető az OCR pontossága. Felmerülhet a kérdés, hogy milyen lehetőségek vannak a pontosság javítására. Mivel az OCR képről vagy kép alapú dokumentumról hivatott szöveget kinyerni, a pontos eredmény első és legfontosabb feltétele a megfelelő minőségű bemenet nyújtása. Az alábbiakban felsorolom, melyek a leggyakrabban előforduló körülmények, amik ronthatják az OCR pontosságát. [8]

- Az eredeti, szkennelésre váró dokumentum minőségére vonatkozóan:
  - Gyűrött, szakadt papír, vagy elmosódott szöveg a papíron
  - Sérült, lekopott kártya
  - Fakulás, elszíneződés
  - Fényes felület
  - Színes tintával nyomtatott vagy festett szöveg
  - Nem szokványos betűtípus használata
  - Kézírás

– A beszkennelt vagy kamerával elkészített kép minőségére vonatkozóan:

  - Homályos, életlen kép
  - Elmosódott, torz szélek
  - Alacsony képfelbontás
  - Zajosság, szemcsésség

Hogy az OCR munkáját elősegítsük, és ezzel javítsuk a pontosságot, az alábbi lépéseket tehetjük, mint felhasználók. [6]

– Megfelelő pontsűrűséggel rendelkező kép választása:

  - Általában egy 200-300 közötti pontsűrűséggel (DPI) rendelkező kép a legmegfelelőbb.
  - Kisebb értéknél bizonyos karaktereknél előfordulhat, hogy hibásan kerülnek felismerésre.
  - Nagyobb értékeknél szükségtelenül nagy méretű kép lesz a bemenet, az OCR pontossága ezen intervallum felett nem javul számottevően.

– Kontraszt növelése és színek eltüntetése:

  - Az OCR egyik lépése – ahogy azt a működésénél részletesebben kifejtettem – arra alapul, hogy a képen a világos részeket elválasztja a sötét részektől, és a sötét részeket jelöli meg szövegként, a világos részeket pedig háttérként. Ebből adódik a kép kontrasztjának és színvilágának szerepe a szövegfelismerésben, hiszen minél nagyobb a kontraszt, illetve minél kevesebb szín található a képen, annál pontosabban fogja tudni az OCR leválasztani a szöveget a háttérről.
  - Az OCR szempontjából egy jó bemeneti kép erősen kontrasztos és nem tartalmaz színeket. Kontrasztot ma már bármilyen képszerkesztő alkalmazással tudunk növelni, illetve filterek alkalmazásával szürkeárnyalatossá tudjuk alakítani a színes képeket.

- Ferdeségkorrekció:
  - A ferdén fotózott vagy szkennelt képek nagymértékben csökkentik az OCR hatékonyságát, hiszen a karaktereket meghatározott vonalakból és alakzatokból próbálja felismerni. Ha a képen ferdén vannak a karakterek, akkor nehezebben fog egyezést találni a saját adatbázisában szereplő karakterekkel.
  - A szkenneléskor vagy fotózáskor törekedni kell arra, hogy a kép minél kevésbé legyen ferde, de lehetőség van utólagos korrekcióra is képszerkesztő program segítségével.
- Zajeltávolítás:
  - Törekedni kell arra, hogy a képet megfelelő fényviszonyok mellett készítsük, hogy az minél kevésbé legyen zajos.
  - Bizonyos eljárásokkal csökkenthető az elkészített kép zajossága is, például simítási és zajmentesítési folyamatokkal, melyek szintén megtalálhatóak néhány képszerkesztő program funkciói között.

3. fejezet

A program megvalósítása

3.1. OCR könyvtárak összehasonlítása Pythonban

A megvalósítás megkezdése előtt mindenképpen szükséges feltérképezni a rendelkezésre álló OCR könyvtárakat. Ehhez szükséges azt is meghatározni, hogy milyen programozási nyelven fog a program elkészülni. Hosszas mérlegelés után a Python mellett döntöttem, mely egy olyan programozási nyelv, amely a népszerűségét többek között a rugalmasságának köszönheti, hiszen széles körben használható, legyen az webfejlesztés, adatbányászat, gépi tanulás, automatizálás vagy számítógépes grafika. A nyelv továbbá könnyen olvasható egyszerű szintaxisa miatt és támogatja az objektumorientált programozás alapelveit. Népszerűségének alappillérei közé tartozik a folyamatosan fejlődő és bővülő eszközkészlet és a számos ingyenes, nyílt forráskódú könyvtár [9], melyek a szakdolgozatban is fontos szerepet töltenek be.

A következő alfejezetekben két, általam választott ingyenes, nyílt forráskódú OCR könyvtár működését fogom bemutatni, rávilágítva a fő különbségekre.

3.1.1. A pytesseract könyvtár bemutatása

A Tesseract egy nyílt forráskódú OCR motor, mely számos programozási nyelvvel és keretrendszerrel kompatibilis. Ahhoz, hogy Pythonból használni tudjuk a Tesseract funkcióit, szükségünk van egy wrapperre, azaz egy olyan könyvtárra, amely Python nyelvből teszi lehetővé a Tesseract használatát. A pytesseract nevű könyvtár ezt a célt szolgálja. [10]

Néhány példa a pytesseract függvénykönyvtár importálása után rendelkezésünkre álló metódusokból: [11]

Az *image_to_string()* metódus, ahogy a neve is körülírja, a képről kinyert szöveget egy összefüggő szövegként adja vissza. Ennek a metódushívásnak az eredménye hasonlít a **jelenlegi rendszerek bemutatása** című fejezetben bemutatott internetes alkalmazások használatával kapott eredményre.

IDENTIFICATION CARD
Family name and Given name:
MAROSI MARK DANIEL
sex: MALE Nationality: HUN
Date of birth: 21/09/1999
Date of expiry: 21/09/2023
Document number: 123456AB
CAN: 123456

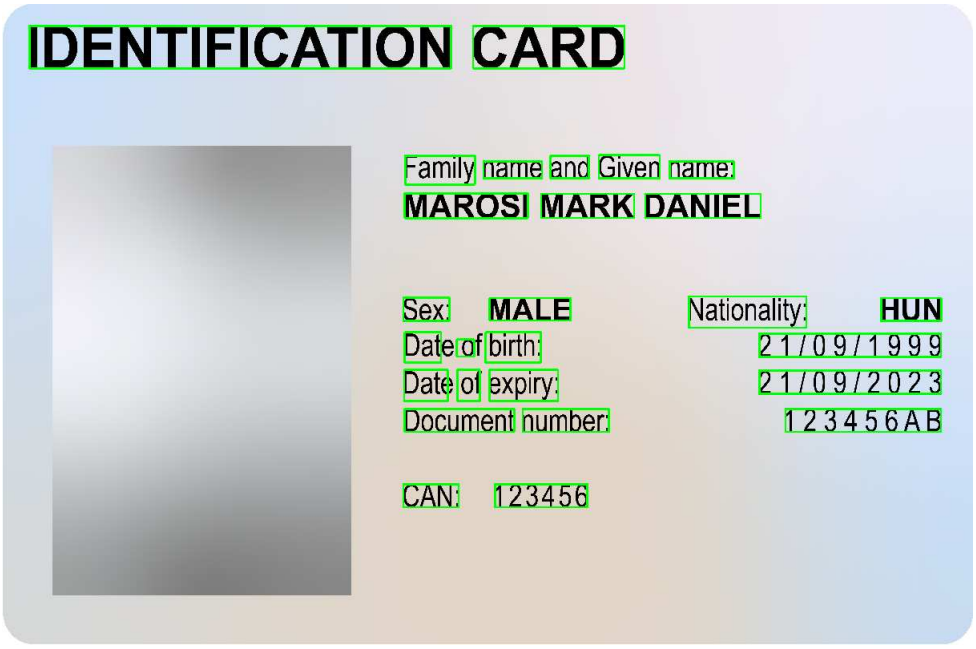
Az *image_to_boxes()* metódus minden egyes felismert karaktert egyesével, az őt körülhatároló négyzet bal felső sarkának koordinátáival, valamint a négyzet szélességével és magasságával adja vissza listába rendezve. Egy részlet az eredményhalmazból:

M 3085 2161 3202 2282
A 3213 2161 3334 2282
R 3347 2161 3456 2282
K 3470 2161 3579 2282

Az *image_to_data()* függvény a számunkra leghasznosabb, ugyanis ez figyelembe veszi a karakterek közelségét egymáshoz, ezáltal képes meghatározni szavakat, szorosan összefüggő szövegrészleteket. A szavak mellett visszadja a szót körülhatároló téglalap bal felső sarkának koordinátáját, szélességét, magasságát, valamint azt is, hogy az adott szó vagy szövegrészlet milyen pontossággal került meghatározásra százalékban kifejezve. Az, hogy a függvény milyen adatstruktúrában adja vissza a kinyert szavakat, konfigurálható az output_type paraméterrel. Választhatunk byte, string, dictionary illetve dataframe opciók közül.

top	left	width	height	confidence	text
956	2362	653	125	74	MAROSI
958	3085	494	121	95	MARK
958	3638	612	121	95	DANIEL

Az *image_to_data()* függvény eredményének szemléltetése céljából írtam egy függvényt, melyen a bemenetként adott igazolványképen a felismert szövegeket határoló téglalapokat kirajzoltattam az OpenCV nevű Python könyvtárban definiált metódusok segítségével [13]. Az OpenCV könyvárat a későbbiekben ismertetem. A függvény egy paraméterrel rendelkezik, amely az *image_to_data()* függvény eredményét várja dictionary struktúraként. A függvény kimenete egy új ablakban megnyíló kép, amelyen a bemeneti kép látható úgy, hogy a rajta található, egyben felismert szövegrészek körül megjelenik a szöveget körülhatároló téglalap.



3.1. ábra. A függvényhívás eredménye, a Pytesseract által kinyert adatok

```
def generate_img_with_bounding_boxes_on_words(ocr_result):
    img = cv2.imread(ID_CARD_PATH)

    for i in range(len(ocr_result['text'])):
        if float(ocr_result['conf'][i]) >= CONF_LEVEL:
            (x, y, w, h) = (ocr_result['left'][i],
                            ocr_result['top'][i],
                            ocr_result['width'][i],
                            ocr_result['height'][i])
            img = cv2.rectangle(img, (x, y),
                (x + w, y + h), (0, 255, 0), 10)

    output_img_to_window(img)
```

```
def output_img_to_window(img):
    cv2.namedWindow("output", cv2.WINDOW_NORMAL)
    cv2.imshow("output", img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

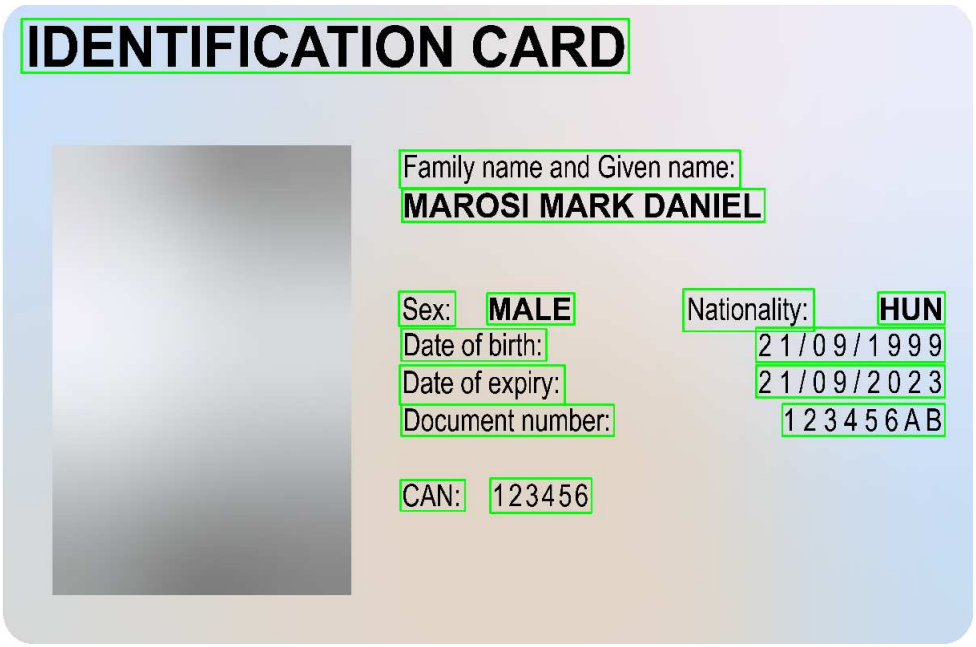
3.1.2. Az EasyOCR könyvtár bemutatása

Az EasyOCR könyvtár is nyílt forráskódú [12], ingyenesen használható OCR könyvtár. A Tesseracttól független, eltérő az implementációja, összességében nagyobb pontosságot képes biztosítani bizonyos esetekben. Azonban hátulütője, hogy nagyobb fájlméretű bemenetekre lassabb, mint a Tesseract.

Az EasyOCR importálása után először példányosítani kell egy objektumot az *easyocr.Reader()* konstruktorhívással, ezután rendelkezésünkre állnak az EasyOCR metódusai. Ezek közül a szakdolgozat szempontjából a *readtext()* metódus a legfontosabb, ennek egyetlen paramétere a kép elérési útvonala, melyről szöveget szeretnénk kinyerni. A metódus egy listával tér vissza, melyben minden listaelem egy szó vagy szövegrészlet, melyet az OCR a karakterek egymáshoz viszonyított pozíciója alapján összefüggőnek ítélt. Minden egyes listaelem további elemeket tartalmaz. Első helyen egy listát, ami az adott szöveget körülvevő téglalap négy sarkának koordináta-párjait tárolja, második helyen magát a felismert szöveget, a harmadik pozíción a pedig tizedestört alakban kifejezve azt, hogy az OCR hány százalékban biztos abban, hogy a kinyert szöveg egyezik a képen látható szöveggel.

bounding box coordinates	text	confidence
[[329, 78], [3551, 78], [3551, 350], [329, 350]]	Nationality:	0.97

Ahogyan azt a pytesseract könyvtár eredményén megtettem, itt is szemléltetésképpen írtam egy metódust, ami a bemenetként adott képen megjeleníti a felismert szavakat, összefüggőnek vélt karakterláncokat.



3.2. ábra. A függvényhívás eredménye, az EasyOCR által kinyert adatok

```
def generate_img_with_bounding_boxes_on_words(ocr_result):
    img = cv2.imread(ID_CARD_PATH)

    for text_row in ocr_result:
        bottom_left = tuple(text_row[0][0])
        top_right = tuple(text_row[0][2])
        img = cv2.rectangle(img, bottom_left, top_right,
                            (0, 255, 0), 10)

    cv2.namedWindow("output", cv2.WINDOW_NORMAL)
    cv2.imshow("output", img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Ha összehasonlítjuk a két bemutatott könyvtár által adott eredményeket, és összevetjük az utóbbi két képet, megfigyelhető, hogy valóban van eltérés a két metódus eredménye között. A fő különbség a szövegrészletek szegmentáltságából adódik: míg az EasyOCR az egymáshoz közel álló szavakat, karakterláncokat sokkal inkább egy egységként dolgozza fel, addig a Pytesseract karakter alapon vizsgálja a dokumentumot, kevésbé érzékeny a kontextusra, ebből adódóan szavakra bontva dolgozza fel a képen olvasható szöveget.

3.2. A dokumentum előfeldolgozása

Ahogyan "Az OCR pontosságának javítása bemeneti oldalon" című fejezetben kifejtettem, a lehető legpontosabb OCR kimenet eléréséhez szükségünk van a bemeneti kép előfeldolgozására pár lépésben az OpenCV nevű ingyenes könyvtár különböző metódusainak használatával. [14] Mivel a bemeneti dokumentum lehet színes, így először azok eltüntetésével kezdtem, hiszen a színek a szövegkinyerés szempontjából felesleges információt hordoznak. Ezt az OpenCV *cvtColor()* nevű metódusával értem el, melynek paraméterei egy kép, és az a színtér, melyre konvertálni szeretnénk. Esetünkben ez a színtér a *COLOR_RGB2GRAY*, ami az RGB színeket a szürke árnyalataivá konvertálja.

```
def convert_to_grayscale_image(image):
    return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
```

Következő lépésként a zajeltávolítás volt a célom. Egy zajos kép nagymértékben ronthatja a kimenet pontosságát, mivel az OCR a zajos, szemcsés részeket is az adott karakter részének határozhatja meg. A zajeltávolítás legegyszerűbb módja a kép kis mértékben történő elmosása, mivel így a zajos részeket összemossuk, és jobban beleolvad a háttérbe. Cserébe, ha tényleg csak kevés elmosást alkalmaztunk, a karakterek szélei csak kis mértékben veszítenek az élességükből, ezzel továbbra is fenntartva a felismerhetőséget. Erre a célra a *medianBlur()* metódust hívtam meg az OpenCV könyvtárból, melynek paraméterei a kép és egy 1-nél nagyobb páratlan szám, mely az elmosáshoz használt digitális rekesz lineáris méretét adja meg.

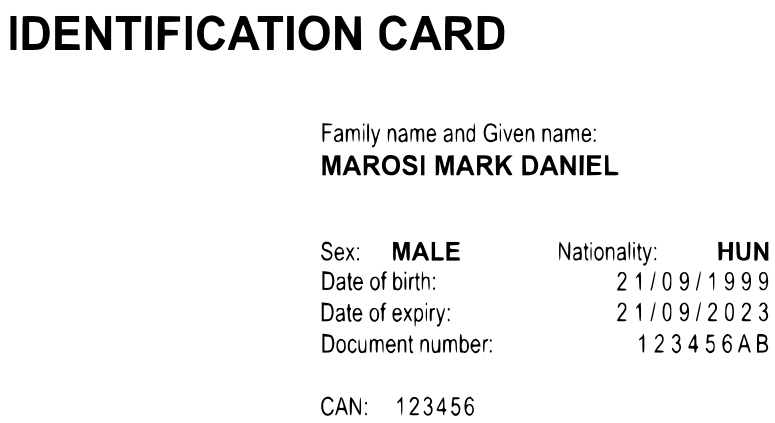
```
def remove_noise(image):
    return cv2.medianBlur(image, 5)
```

Utolsó lépésben a képen található szöveget elkülönítjük a háttértől, amennyire csak lehet. Ezt az OpenCV *threshold()* nevű metódusával érhetjük el, melyet általában arra használnak, hogy egy szürkeárnyalatos képet két színből álló képpé konvertáljanak. A függvény paraméterei a kép, egy küszöbérték (*threshold*), egy maximum érték (*maxval*), és a küszöbérték típusa. Különböző célokra különböző küszöbérték típusok léteznek [15], számomra a *THRESH_BINARY* a legmegfelelőbb. A *THRESH_BINARY* küszöbérték típus számítási elve:

$$output(x,y) = \begin{cases} maxval, & \text{ha } input(x,y) > threshold \\ 0, & \text{egyébként} \end{cases} \tag{3.1}$$

A képletből kiolvasható, hogy amennyiben a vizsgált pixel színe a küszöbértéknél nagyobb, akkor a szín átírásra kerül, és a maxvalue értékét fogja felvenni, egyéb esetben az

új értéke 0 lesz. Ahhoz, hogy a szöveg jól elkülönüljön a háttértől, a lehető legszűkebb küszöbértéket kell megadnunk, így a *threshold* paraméter a mi esetünkben 0 lesz. A *max-value* az a szín, amit a küszöbértéknél nagyobb, tehát az összes kiszűrni kívánt képpont (ebben az esetben a dokumentum háttere) fog felvenni, esetünkben 255 lesz, azaz fehér.



3.3. ábra. A három előfeldolgozó metódus futása utáni állapot

A három képfeldolgozó metódus lefuttatása után elértem, hogy a szöveg jól elkülönüljön a háttértől, a karakterek könnyen felismerhetőek legyenek, és ne maradjon a szövegfelismerés szempontjából irreleváns jelentést hordozó adat a dokumentumon.

3.3. Adatstruktúra felállítása és feltöltése

Ahhoz, hogy a képről kinyert szöveget hatékony módon, könnyen kereshető és osztályozható formában tudjam tárolni, az objektum orientált programozás egyik alappillérét, az osztályokat hívtam segítségül. Egy osztály többek között rendelkezhet tulajdonságokkal és definiálhat metódusokat. A dokumentumról kinyert adatoknak sok közös tulajdonságuk van. Mindegyik magával hordoz egy karakterláncot, valamint a dokumentumon való elhelyezkedést leíró koordinátákat. Pythonban az adattagok definiálására és az értékeik kezdetleges beállítására a konstruktor szolgál.

```
class CardTextItem:
    def __init__(self, top_left, top_right, btm_left,
                 btm_right, text, conf):
        self.top_left = top_left
        self.top_right = top_right
        self.btm_left = btm_left
        self.btm_right = btm_right
        self.text = text
```

A *top_left*, *top_right*, *btm_left* és *btm_right* adattagok rendre annak az alakzatnak a bal felső, jobb felső, bal alsó és jobb alsó pontjainak koordináta párjait jelölik, mely pontosan körbehatárolja az egybefüggően kinyert karakterláncot.

Azzal, hogy elkészült az adatstruktúra, már csak fel kell tölteni az OCR által kinyert adatokkal. Ehhez vissza kell emlékezni az **EasyOCR könyvtár bemutatása** című alfejezet során szemléltetett adatstruktúrára, melyben rendelkezésünkre áll a dokumentumról felismert összes karakterlánc. Ez egy olyan lista, melyben minden elem egy további struktúrát tartalmaz, benne a koordinátákkal, a szöveggel és a szöveg pontosságát becslő számmal. Ezt a listát szeretném úgy szétbontani, hogy minden elemét leképezem egy olyan objektummá, ami az előbb felállított adatstruktúra egy példánya, azaz minden egyes karakterlánc egy önálló objektum lenne.

```
def get_text_items_from_ocr_data(ocr_data):
    card_text_items = list()

    for i in range(len(ocr_data)):
        item_bl = ocr_data[i][0][0]
        item_br = ocr_data[i][0][1]
        item_tr = ocr_data[i][0][2]
        item_tl = ocr_data[i][0][3]
        item_text = ocr_data[i][1]

        text_item = cti.CardTextItem(item_tl, item_tr,
                                     item_bl, item_br, item_text)
        card_text_items.append(text_item)
```

A metódusban végigiterálunk az OCR által kinyert adatokon, és az aktuális elemet tovább bontva a megfelelő adattaghoz rendeljük. Ezután létrehozunk belőle egy példányt, majd hozzáadjuk a kész, feltöltött objektumokat tároló listához. A metódus futása után a lista minden eleme egy OCR által kinyert karakterláncot tartalmazó objektum. Példaképpen, az általam készített minta személyi igazolványról kinyert adatok listájának első eleme a metódus futása után:

Text: IDENTIFICATION CARD
top_left: [333, 350], top_right: [3551, 350]
btm_left: [333, 78], btm_right: [3551, 78]

3.4. Kulcs-érték párok azonosítása

A program írása során ezen a ponton már nem volt triviális az, hogy milyen irányban folytatssam a fejlesztést. A dokumentumról az EasyOCR segítségével sikerült kinyerni minden szöveget, és bár sikerült ezeknek egy adatstruktúrát felállítani, de a végeredményt tekintve ezen a ponton még csak azt sikerült elérni, amire az interneten található, ingyenes applikációk is képesek. A következő, és egyben a szakdolgozat szempontjából legfontosabb lépés az adatok osztályozása, az összetartozó adatpárok megkeresése.

A dokumentumra tekintve megállapíthatjuk, hogy a legtöbb adat tulajdonképpen egy adatpár, ahol az egyik nevezhető kulcsnak, a másik pedig értéknek. Például a *Document Number* szöveg az egy kulcs, mivel egyértelműen azonosítható belőle, hogy milyen jelentést hordoz. Az *123456AB* karakterlánc pedig ennek a kulcsnak az értéke. Azért érték, mert önmagában nem hordoz jelentést, de ha hozzákapcsoljuk a kulcsához, onnantól tudjuk, hogy mit jelent. Tehát a kulcs az felfogható egy egyedi azonosítóként, amelynek az értéke az azonosított adat. Viszont a kulcs-érték párokba történő rendezés nem egy triviális feladat, és több módszer is felmerült a probléma megoldására.

Az egyik irány az a reguláris kifejezésekre épült volna, a felismert szövegeket bizonyos tulajdonságok alapján próbálta volna a program besorolni egy előre definiált kulcshalmazhoz. Például igazolványszámot a szabvány alapján (adott mennyiségű betűk és számok meghatározott sorrendben), vagy dátumokat a benne szereplő számok és elválasztójelek alapján. De ehhez előre definiálni kellett volna, hogy milyen kulcsok szerepelhetnek a dokumentumon, ami eléggé megkötötte volna a felhasználási lehetőségeket, továbbá a reguláris kifejezésekben kevés kihívást, kisebb rugalmasságot és pontosságot véltem felfedezni. Ennek ellenére a reguláris kifejezések használatának ötletét nem felejtettem el, és a későbbiekben felhasználtam.

Mindenképpen szerettem volna felhasználni az OCR által kinyert szövegekhez tartozó koordinátákat, így a következő ötletem az volt, hogy egy előre meghatározott sablon segítségével felismerhetőek legyenek az összetartozó adatok. A sablon úgy működött volna, mint egy generikus verzió a vizsgált dokumentumból, amin előre megadtam volna, hogy milyen koordináták között milyen adatot kell keresni. Feltételezve, hogy a vizsgált dokumentum illeszkedik a sablonra, könnyen meghatározható lett volna, hogy melyik karakterlánc milyen jelentést hordoz. Viszont ez is egyfajta kötöttséget vont volna maga után, hiszen ha kicsit változik a dokumentum felépítése, akkor a sablonon is módosítani kellene, hogy a kinyert adatok jelentése továbbra is meghatározhatóak maradjanak. Emiatt ezt a lehetőséget is kizártam.

Végül egy saját magam által írt algoritmus mellett döntöttem, ahol nincsenek előre meghatározva a kulcsok, és az értékek elhelyezkedése sincsen megkötve, mint egy sab-

lonnál. Ebből kifolyólag az algoritmusnak képesnek kell lennie megállapítania minden adatról, hogy az kulcs, vagy érték. Ez természetesen nem egy egyszerűen megállapítható és eldönthető kérdés, meg kellett határoznom, hogy milyen feltételek mellett számít valami kulcsnak vagy értéknek. Sok dokumentum típust végignézve megfigyeltem, hogy a kulcsok általában vagy az értéktől balra, vagy az érték felett helyezkednek el. Ez természetesen nem fed le minden esetet, de egy elég jó kiindulási pont volt az algoritmus implementálásához.

Az algoritmus végigiterál a teljes listán, mely a kinyert szövegeket tároló objektumokat tartalmazza. Minden iteráció első lépése az, hogy keressünk az objektumok közül egyet, amire nagy biztossággal rámondható, hogy az egy kulcs. Ennek a megállapítására írtam a find_next_key() metódust, mely az összes objektumot összehasonlítva megkeresi azt, amelyik a koordinátái alapján a többihez képest leginkább balra és leginkább felül van, máshogy fogalmazva, megkeresi a bal felső sarokhoz legközelebb eső elemet. Mivel feltételeztem, hogy a kulcsok minden esetben vagy az értkékek bal oldalán, vagy az értékek felett vannak, így elméleti szinten az algoritmus nem találhat értéket, hiszen annak kulcsa mindenképpen közelebb lesz a bal felső sarokhoz.

Először a *CardTextItem* objektumba fel kellett vennem 4 új adattagot: *is_examined*, *assigned_to*, *is_key*, *is_value*. Mind a négy adattag kezdetben 0-ra inicializálódik. Az is_examined egy logikai változóként viselkedik, és azt jelzi, hogy a kulcs-érték pár keresés lefutott-e már rá. Erre azért volt szükség, hogy ha az adott szövegre futó keresés nem talált hozzá tartozó párt, akkor jelezze, hogy ez nagy eséllyel egy olyan szöveg, mely csak tájékoztató jellegű, nem pedig egy fontos adat. Ilyen például az igazolvány mintán az *IDENTIFICATION CARD* szöveg, mely nem lesz kulcs, hiszen érték sem tartozik hozzá. Az *assigned_to adattag* segítségével lesznek összekapcsolva azok az objektumok, amik a kulcs-érték pár keresés során egymáshoz lettek rendelve. Maga az adattag hozzá kapcsolt másik objektumra fog mutatni. Az *is_key* és az *is_value* adattagok pedig szintén logikai változókként funkcionálnak, jelezve, hogy az adott objektum kulcsként vagy értékként lett felismerve.

```
def find_next_key(items):
    next_key = next(item
                    for item
                    in items
                    if item.is_examined == 0)
```

```
    for item in items:
        if item.is_examined == 0:
            if item.top_left[1] < next_key.top_left[1]:
                next_key = item
```

A metódus paraméterben megkapja az objektumok listáját, és a *next()* beépített függvény és egy szűrés segítségével kiválasztja a listából az első olyan objektumot, amit még nem vizsgáltunk. Ezután végig iterálva az objektumokon összehasonlítást végez a bal felső sarok Y koordinátája alapján. Az iteráció végére megkapjuk a dokumentum tetejéhez legközelebb eső (az Y tengelyen legmagasabban lévő) objektumot. Ezen a ponton felmerült egy fontos kérdés. Amennyiben a legfelül elhelyezkedő objektummal egyező magasságban található még több is, akkor a leginkább balra esőt fogjuk kulcsként címkézni. De olyan szélsőséges eset is előfordulhat, hogy nem pont azonos Y koordinátán, de nagyon pici eltéréssel szintén található egy másik objektum, hozzá közel bal oldalon elhelyezkedve, akkor őt kell kulcsként címkézni. Mivel nem feltételezhetjük, hogy a dokumentumon egymás mellett elhelyezkedő szövegek 1 pixel pontosságra egy vonalban vannak, továbbá azt sem, hogy az egymás alatt lévő szövegek bal széle pontosan egy függőlegesen húzott vonalra illeszkednek, így be kellett vezetnem egy olyan tűréshatár értéket, melyet hozzáadva illetve kivonva az éppen vizsgált X vagy Y értékből megkapjuk a tűréshatár intervallumot. Amennyiben ezen az intervallumon belül találunk újabb kulcsokat, azokról meg kell állapítani, hogy közelebb vannak-e a dokumentum bal széléhez, mint amit eddig találtunk. A find_next_key metódus az alábbi sorokkal egészül ki:

```
upper_bound = next_key.top_left[1] + THRESHOLD
lower_bound = next_key.top_left[1] - THRESHOLD

for item in items:
    if item.is_examined == 0:
        if lower_bound <= item.top_left[1] <= upper_bound:
            if item.top_left[0] < next_key.top_left[0]:
                next_key = item
```

A *THRESHOLD* konstans értékének meghatározása próbálgatással történt. A 40-es érték bizonyult minden esetben jónak, ez még az a tűréshatár, amibe az egymás mellett és alatt elhelyezkedő objektumok beleesnek, de azok már nem, amelyek a keresés eredménye szempontjából helytelenek lennének. Ezt a konstanst többször is használni fogom a kulcs-érték párok felderítése során.

A *find_next_key()* metódus visszatérési értéke az az objektum lesz, amely Y tengelyen a legmagasabban helyezkedik el. Amennyiben ezen objektum X koordinátájának a

tűréshatár intervallumán belül találunk egy vagy több másik, tőle balra eső objektumot, úgy azzal térünk vissza, amelyik ezek közül a legközelebb van a dokumentum bal széléhez. Tehát a metódus visszatér egy objektummal, melyről innentől feltételezzük, hogy kulcs. Azért nem jelenthető ki biztosan, mert kizárólag onnantól címkézünk egy objektumot kulcsként, ha megbizonyosodtunk róla, hogy tartozik hozzá érték. Ezen a ponton ebben még nem lehetünk biztosak.

A folyamat következő lépése a *find_value_for_key()* nevű metódus, melynek paraméterei az imént talált objektum, mely feltételezhetőleg kulcs, valamint a teljes objektumok listája. A futás rögtön azzal kezdődik, hogy az objektum *is_examined* adattagját 0-ról 1-re állítja, ezzel jelölve, hogy a párkeresési algoritmus már meg volt hívva a kulcsra. Ennek célja, hogy ha nem találunk hozzá tartozó értéket, akkor se vizsgáljuk többet. A metódus ezután két fő lépcsőből áll. Az első, hogy elindít egy jobbra történő keresést, melynek, ha van eredménye, akkor az a kulcstól jobbra talált legközelebbi objektum. Innentől tudjuk, hogy az eddig kulcsnak vélt objektum valóban kulcs, hiszen találtunk egy objektumot mellette, mely feltételezhetően a hozzá tartozó érték. Amennyiben a jobbra keresésnek nem lett eredménye, úgy egy lefelé történő keresés indul. Amennyiben a kulcsnak vélt objektum alatt sem találtunk másik objektumot, akkor a metódus futása véget ér anélkül, hogy a kulcsnak vélt objektumot valóban kulcsként címkéztük volna, és értéket rendeltünk volna hozzá. Ebből megállapítható, hogy az objektumhoz valószínűleg nem tartozik érték. Ilyen például az *IDENTIFICATION CARD* szöveg, se tőle jobbra, se alatta nem helyezkedik el hozzá kapcsolható érték. Mivel a kulcskeresés mindig a bal felső objektumokból indul ki, így teljesen felesleges lenne a balra illetve a felfelé történő keresés, hiszen ott már nem találhatunk objektumot. Ez természetesen hátrány egy olyan dokumentumnál, melyről nem mondható el, hogy a kulcsok minden esetben az értékek bal oldalán vagy felettük találhatóak.

A *search_right()* metódus ugyanazokkal a paraméterekkel rendelkezik, mint a *find_value_for_key()*. Futása során először kiszámolja a tűréshatár intervallumot, majd ezen belül keres egyéb olyan objektumokat, melyek nem voltak még vizsgálva, és nincsenek más objektumhoz rendelve. Továbbá feltétel az is, hogy a kulcshoz képest nagyobb X koordinátával rendelkezzen, azaz tőle jobbra helyezkedjen el a dokumentumon. A talált objektumokat egy listában gyűjtjük, és a jobbra keresés végén, amennyiben a listában nincs elem, akkor tudjuk, hogy nem volt objektum a kulcstól jobbra. Ebben az esetben egy lefelé történő keresés indul. Amennyiben csak egy objektum van a találatok listájában, úgy visszatérünk azzal az objektummal. Ha a találatok listája több elemet is tartalmaz, akkor a *get_closest_item_right()* függvény fog hívodni, mely a listát kapja paraméterül, és a koordináták alapján visszaadja azt, amelyik jobbról a legközelebb található a kulcshoz.

A *search_below()* metódus ugyan úgy működik, mint a search_right, csak nem jobbra irányul a keresés, hanem a kulcstól lefelé. Ha a lefelé történő keresés során több objektumot is találunk, akkor hasonlóan, mint a search_right metódusban, megkeressük azt, ami a legközelebb van a kulcshoz, csak nem az X tengelyről, hanem az Y-ról, a *get_closest_item_below()* metódus segítségével.

A kulcs-érték keresési folyamat belépési pontja a *find_key_value_pairs()* metódus. Ebben egy while ciklus és egy listaszűrés felel azért, hogy egy objektumon legfeljebb egyszer menjen végig a párkereső algoritmus.

```
def find_key_value_pairs(items):
    while any(item.is_examined == 0 for item in items):
        next_key = find_next_key(items)
        find_value_for_key(next_key, items)
```

Az *any()* beépített listaszűrő metódus igazzal tér vissza, ha a lista legalább egy olyan objektumot tartalmaz még, ami nem volt vizsgálva. Ennek a helyességét a korábban tárgyalt *find_value_for_key()* metódus biztosítja. Tehát addig, amíg van olyan objektum, melyet nem vizsgáltunk, keresünk egy kulcsot a dokumentumon a *find_next_key()* metódussal, majd a kulcshoz megkeressük a hozzá tartozó értéket a *find_value_for_key()* függvénnyel. Mivel Pythonban alapértelmezetten referencia szerinti paraméterátadás történik, így nem tér vissza semmivel ez a metódus, mert minden módosítást az eredeti objektumokon végzünk. Ha már nincs több vizsgálatlan objektum, akkor visszatérünk a main függvénybe. Ezen a ponton a feladat el lett végezve, hiszen minden objektumot megvizsgáltunk, megtaláltuk a kulcs-érték párokat. Viszont az eddig használt adatstruktúra innentől értelmét veszti, a *CardTextItem* osztályból egyedül a *text* adattag releváns számunka. Az egyszerűbb kezelés érdekében egy kulcs-érték párokat tároló adatstruktúra mellett döntöttem.

```
def items_to_dict(card_text_items):
    result_dict = dict()

    for cti in card_text_items:
        if cti.is_key:
            result_dict[cti] = cti.assigned_to
        elif not cti.is_key and not cti.is_value:
            result_dict[cti.text] = "N/A"

    return result_dict
```

Az *items_to_dict()* metódus átrendezi az osztálypéldányokat egy dictionary adatstruktúrába, melynek minden kulcsa egy általunk kulcsnak vélt objektum *text* adattagja (a dokumentumról kinyert karakterlánc), értéke pedig a kulcshoz tartozó érték *text* adattagja. Abban az esetben, ha egy adott példány se nem kulcs, se nem érték, azaz egy olyan elem, melyhez nem találtunk párt, akkor az átmásolás során őt felvesszük a dictionary egy kulcsaként, és az "N/A" szöveg kerül hozzá értékként, ezzel jelezve, hogy a szöveghez nem találtunk hozzá köthető másik objektumot. A példa igazolványon ilyen az *IDENTIFICATION CARD* szöveg. Ezekről meggyőződhetünk, ha végigiterálunk rajta és kiíratjuk minden elemét:

```
Key: IDENTIFICATION CARD, Value: N/A
Key: Family name and Given name:, Value: MAROSI MARK DANIEL
Key: Sex:, Value: MALE
Key: Nationality:, Value: HUN
Key: Date of birth:, Value: 2 1/0 9 /1 9 9 9
Key: Date of expiry:, Value: 2 1/0 9 /2 0 2 3
Key: Document number:, Value: 1234 5 6 A B
Key: CAN:, Value: 123456
```

A minta igazolványról az adatok hibátlanul kerültek kinyerésre és a kulcs-érték párok megtalálása és párosítása is kiválóan sikerült. Viszont megfigyelhetjük, hogy bizonyos esetekben a szövegek formázásra, utólagos korrigálásra szorulnak.

3.5. Adatok utófeldolgozása és exportálása

Ha végignézzük a kulcs-érték párok kinyerésének eredményét, láthatjuk, hogy a kulcsok végén kettőspont szerepel. Ennek csak a dokumentumon volt szerepe, ez segített az emberi szemnek a kulcsokat összekötni az értékekkel. Viszont digitalizálva már szükségtelenek, így eltávolírásra kerülhetnek a Python függvénykönyvtárába beépített *replace()* metódus segítségével.

A kulcsokhoz kapcsolt értékeket végigfutva néhány esetben azt vehetjük észre, hogy felesleges szóközök szerepelnek a karakterek között. Ezek még az OCR folyamat során kerültek ilyen formában kinyerésre. Ebből látszódik, hogy az egy gyenge pontja az EasyOCR könyvtárnak, ha egy karakterláncban az általánosnál nagyobb a betűköz. Utólagos javításra a legegyszerűbb módszer szintén a *replace()* metódus lenne, de fontos észrevenni, hogy nem minden érték esetében tehetjük ezt meg. A név esetében például elrontanánk az adat helyességét azzal, hogy a vezetéknév és a keresztnév vagy keresztnevek közül eltűntetnénk a szóközöket. Ezért valahogyan ki kell szűrni azokat az adatokat,

melyekről eltávolíthatóak a felesleges szóközök anélkül, hogy az a helyességük rovására menne.

Erre a célra mintaillesztést használtam. A mintaillesztés reguláris kifejezések segítségével történik, melyben megszabjuk, hogy milyen szabályokat és mintákat kell követnie az általunk vizsgált szövegnek. A mintaillesztés kimenete lehet elfogadó, amennyiben a szöveg teljes egészében a mintára illeszkedik, és megfelel a követelményeinknek, egyébként elutasító. Tudjuk, hogy az olyan adatok, mint a nemzetiség kódja, a nem, a dátumok és a dokumentumszámok általános esetekben nem tartalmazhatnak szóközt. Így ezekre az adat kategóriákra írtam egy-egy reguláris kifejezést.

A dátumformátum a legbonyolultabb, hiszen a nap, hónap és év sorrendje változó, továbbá az elválasztójelek is lehetnek eltérőek. Ehhez az interneten kerestem már kész reguláris kifejezéseket, [16] és azt kicsit átalakítva a programomba három konstansként felvettem. Az első az *év*, *hónap*, *nap*, a második a *nap*, *hónap*, *év*, a harmadik pedig a *hónap*, *nap*, *év* sorrendű megfelelőjét tárolja a reguláris kifejezésnek. Mind a három esetben az évet négy számjegy, a hónapot és a napot pedig két-két számjegy jelöli. Egyéb eseteket nem vettem figyelembe, mivel ez a három nagyon nagy százalékban lefedi a különböző igazolványokon található formátumokat.

Az nemzetiséget jelölő ország kódjának felismerésére saját reguláris kifejezést írtam. Az országok kódjai az ISO-3166 szabvány alapján kettő vagy három betűből állhatnak. [17] Az erre írt reguláris kifejezés így néz ki: *^[a-zA-Z]{2,3}$*. Ez egy egyszerű kifejezés, mely a kis- és nagybetűk között különbséget nem téve korlátozza, hogy a betűk számának kettő és három között kell lennie, semmilyen egyéb karakter nem megengedett a szövegen belül, de előtte és mögött sem.

A dokumentumszámoknál és egyéb azonsítóknál azt a feltételt szabtam meg, hogy a karakterláncnak tartalmaznia kell legalább négy számot. Mivel ebbe beleesnek a dátumok is, így a szűrésnél figyelni kell, hogy először a dátum reguláris kifejezésére történjen a próbaillesztés, és ha arra nem illeszkedett a szöveg, akkor lehet próbálni a dokumentumszámra is. A dokumentumszám reguláris kifejezése: *^(.*\d){4,}\S*$*. Bármilyen, bármennyi (akár nulla) karakter után kötelezően négy vagy több szám kell, hogy szerepeljen, ami után bármennyi egyéb karakter állhat, akár egy sem.

Az utófeldolgozási folyamat lépéseit a *post_process_text()* metódusban gyűjtöttem össze. A metódus végigmegy az összes szövegen, a kulcsokról leszedi a kettőspontot, az értékeket pedig megtisztítja a nem kívánatos szóközöktől, amennyiben a mintaillesztés eredménye alapján ez nem változtatja a szöveg helyességét.

```
def post_process_text(items_dict):
    new_items_dict = dict()

    for key, value in items_dict.items():
        cleaned = value.replace(' ', '')
        if matches_any_regex(cleaned):
            new_items_dict[key.replace(':', '')] = cleaned
        else:
            new_items_dict[key.replace(':', '')] = value

    return new_items_dict
```

Az adatok exportálására a *csv* nevű függvénykönyvár volt a segítségemre. A *with open() as* segítségével erőforrást allokálok egy fájl írására, majd példányosítok egy writer objektumot, mely a beolvasott dokumentum mappájába fog létrehozni egy .csv kiterjesztésű fájlt. A későbbi konfigurálhatóság érdekében a *FILE_EXT* konstansban tárolom a kiterjesztést leíró stringet. Az adatstruktúrán végigiterálva annak kulcsai és értékei beírásra kerülnek a fájlba pontosvesszővel elválasztva, sortöréssel lezárva. Ezzel lehetővé teszem, hogy Excelbe is könnyen importálható legyen a kinyert adatok összessége, de bármilyen szerkesztővel megnyitva is könnyen olvasható legyen az emberi szemnek is a fájl tartalma.

```
def export_data_to_csv(path, data):
    filename = f"{path}/{name}_result.{FILE_EXT}"

    with open(filename, 'w') as file:
        writer = csv.writer(file, delimiter=';')

        for key, value in data.items():
            writer.writerow([key, value])
```

4. fejezet

A kész alkalmazás bemutatása

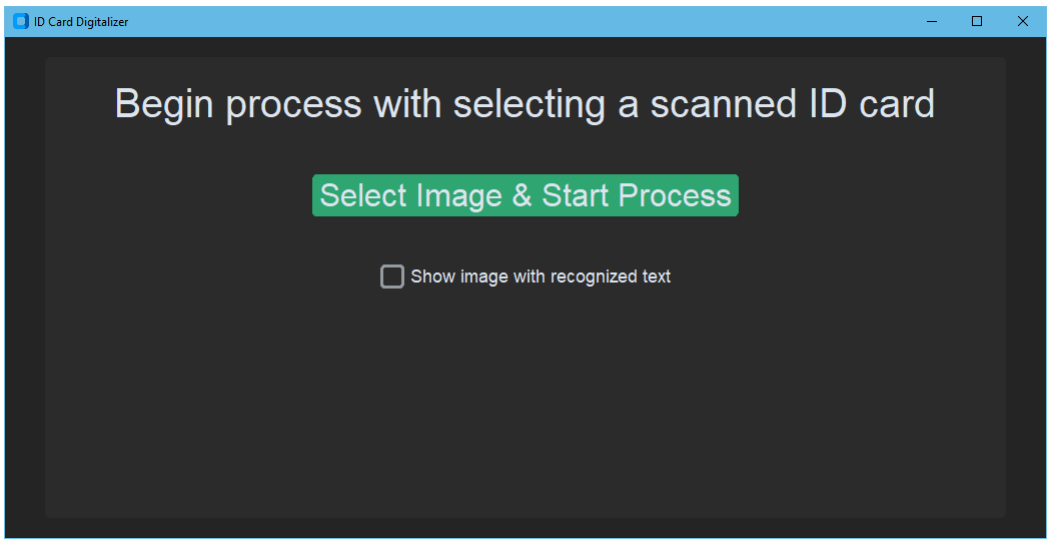
Mivel a program elkészítése során a fókusz a szövegfeldolgozáson és a kulcs-érték pár keresésen volt, így a felhasználói felület kapta a legkisebb figyelmet. Mivel ez egy olyan alkalmazás, melynek célzott, szűk körű közönsége van, melyek inkább cégek vagy szervezetek, és nem magánszemélyek, így a lehető legegyszerűbb, minimalista felületet kapott a program. Ennek megvalósítására a *tkinter* nevű Python könyvtár eszközkészlete tűnt a legjobbnak, mely a *Tcl/Tk* (Tool Command Language) [18] eszközkészlet Pythonra írt interfésze. A *tkinter* könyvtárnak is létezik néhány egyedi megvalósítása, ezek közül dizájn és színvilág szempontjából a *customtkinter* [19] tetszett a legjobban, így ezt a könyvtárat használtam a grafikus felhasználói felület elkészítésére.

4.1. Az alkalmazás használata

A felület az egyszerűség kedvéért mindössze egy gombot tartalmaz, mely kattintásra a fájlkezelőt nyitja meg, és várja, hogy a felhasználó tallózzon egy kép típusú fájlt. A fájl kiválasztása után a szövegkinyerési folyamat azonnal elindul, mely több másodpercet is igénybe vehet, erre a felhasználót egy "Process started, please wait..." üzenet figyelmezteti. Amint a szöveg kinyerése, feldolgozása és exportálása sikeresen megtörténik, a következő üzenet olvasható: "Process finished, CSV file saved to folder of the source image.". Ha ezen a ponton bemenetként adott kép mappájába tekintünk, találnunk kell egy .csv kiterjesztésű fájlt, mely az alábbi módon van elnevezve: originalFileName_result.csv (ahol originalFileName a bemenetként adott fájl neve). Ezzel biztosítottam azt, hogy ha ugyanabból a mappából több fájlra is futtatásra kerülne a program, akkor minden fájlhoz megtalálható legyen a hozzá tartozó eredmény, továbbá ne sokszorozódjon a csv fájl, hanem íródjon felül az előző verzió.

A képen látható, hogy a felhasználónak lehetősége van kijelölni a "Show image with recognized text" jelölőnégyzetet. Amennyiben bejelölésre kerül a beállítás, a kulcs-érték

párok megtalálása után felugró ablakban megjelenik az inputként adott kép, rajta zölddel bekeretezve minden felismert, egybefüggő karakterlánc, lilával pedig a kulcs-érték párok kerülnek összekötésre. Így a felhasználónak lehetősége van vizuálisan is ellenőriznie, hogy melyik szóhoz melyik érték tartozik, és melyek azok a szavak, amelyekhez nem talált a program hozzá köthető párt.

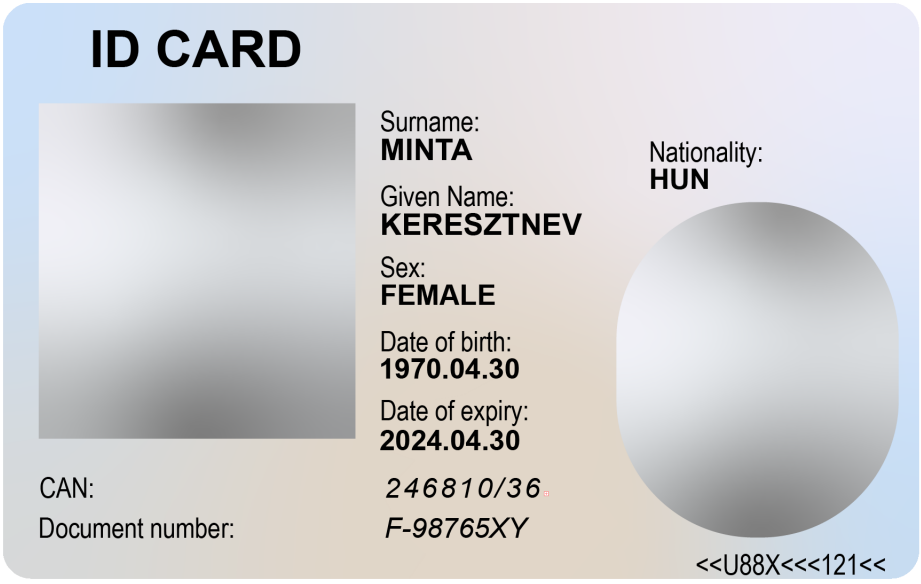


4.1. ábra. A program grafikus felhasználói felülete

A program futása során keletkezhetnek hibák. Amennyiben az "Error while preparing and reading data." hibaüzenettel találja szembe magát a felhasználó, akkor vagy a kép előfeldolgozása, vagy az OCR folyamat során keletezett kivétel. Ha az "Error while processing data." üzenet jelenik meg, akkor az OCR eredményének leképezésénél, vagy a kulcs-érték párok keresésénél történt végzetes hiba. A felhasználó találkozhat még az "Error while post-processing data." üzenettel, ami egyértelműen jelzi, hogy az utófeldolgozási lépés sikertelen volt egy nem várt hiba miatt. Amennyiben egy nem ennyire elkülöníthető folyamat során keletkezik hiba, akkor a "Something went wrong" üzenet jelenik meg. Ezek közül egyik hiba sem okozza a program leállását, így újra lehet próbálni a szövegkinyerési folyamatot.

4.2. Eredmény bemutatása több bemenetre

Az eddig használt példa igazolványról kinyert adatokat már bemutattam, de a program működésének validálásához több féle inputra is szükség van. Ezért készítettem két másik igazolvány mintát, melyeken a szöveg eltérő módon van elrendezve.

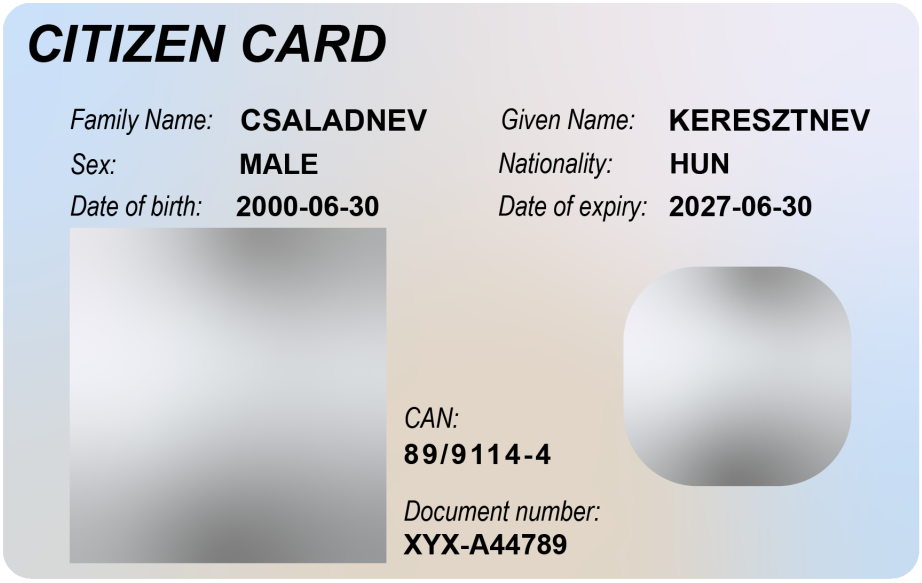


4.2. ábra. A második példa igazolvány

Amíg az első igazolványkép nagyrészt horizontálisan rendezett kulcs-érték párokat tartalmazott (kivéve a névnél), ennél a példánynál arra törekedtem, hogy a kulcs-érték párok egymás alá rendeződjenek, de hagytam vízszintes elrendezésű elemeket is, hogy teszteljem az algoritmus működését. A *CAN* és a *Document number* értékeit kitoltam a legtöbb adatot tartalmazó oszlopba, valamint egy pár nélküli karakterláncot is elhelyeztem a dokumentum jobb alsó sarkába. Hasonló célom volt a *Nationality* kulccsal, melyet jobbra eltolva, de az Y tengelyen két másik érték közé helyeztem el, de így sem sikerült a programot megzavarni. Ez már a csv fájl megnyitása előtt is ellenőrizhető, ha bejelölésre kerül a "Show image with recognized text" opció. Ezen látszódik, hogy a kulcs-érték párosítások helyesek, és az érték nélküli objektumok is párosítatlanul maradtak. A kimeneti csv fájl tartalma:

ID CARD	N/A
Surname	MINTA
Nationality	HUN
Given Name	KERESZTNEV
Sex	FEMALE
Date of birth	1970.04.30
Date of expiry	2024.04.30
CAN	246810/36
Document number	F-98765XY
«U88X«<121«	N/A

A harmadik dokumentumon, amit készítettem, szerettem volna más elrendezéssel is próbára tenni a programot, ezért újból a horizontális elrendezésből indultam ki, de két oszlopba rendeztem a kulcs-érték párokat. A teljesség igényének kielégítésére hagytam olyan objektumokat is, ahol a kulcs az érték felett helyezkedik el. Emellett változtattam a dátumok formátumán és az elválasztó jeleken, valamint a betűtípusokon és stílusokon is.



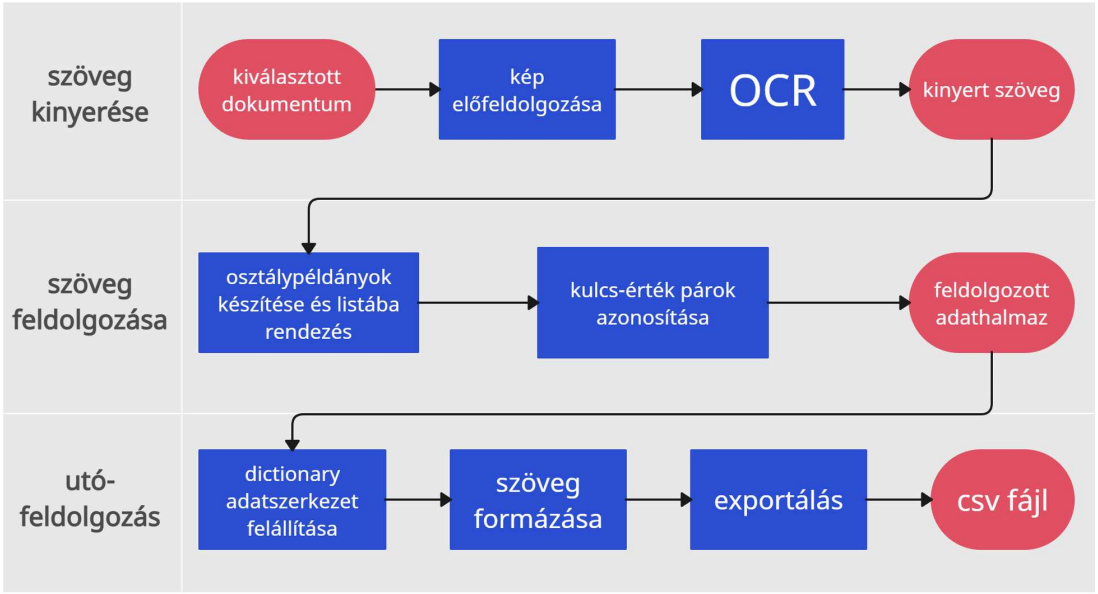
4.3. ábra. A harmadik példa igazolvány

A program futtatása során generált kép, és a az eredményt tartalmazó csv fájlt is igazolja a program helyes működését, ezzel az elrendezéssel is sikeres volt a szöveg kinyerés és feldolgozás.

CITIZEN CARD	N/A
Family Name	CSALADNEV
Given Name	KERESZTNEV
Sex	MALE
Nationality	HUN
Date of birth	2000-06-30
Date of expiry	2027-06-30
CAN	89/9114-4
Document number	XYX-A44789

4.3. Az alkalmazás felépítése

Strukturálisan az alkalmazás három fő egységre bontható. Az elsőbe sorolható minden olyan folyamat, amely a kép alapú dokumentumról való szövegkinyeréssel kapcsolatos. Ennek a folyamatnak az elején csupán egy fájl áll rendelkezésre, a végére viszont valamennyire rendezett formában, digitális formában már rendelkezünk a dokumentumról kinyert szövegről. A második egységben ezen az adathalmazon elindulva először az osztálypéldányok létrehozása, feltöltése, és azok listába rendezése zajlik. Ez a lista lesz az, amin már el tud indulni a kulcs-érték párok keresésére írt algoritmus, melynek kimenete az a rendezett adathalmaz, amely egyéb lépések nélkül is már a helyes eredményt tartalmazza. A harmadik egység az utófeldolgozás, mely célja, hogy a felesleges, idő közben fontosságukat elveszített adatokat ne vigyük magunkkal tovább, többek között ilyenek például a koordináta adatok. Ez a lépés egy dictionary adatszerkezetbe rendezi az összes szöveget a párjával (ha van), így válik az adathalmazunk teljesen rendezetté. Ezután a szebb kimenet érdekében a szöveg formázása történik, ahol a felesleges karakterek és szóközök eltávolításra kerülnek. Végül az adathalmaz készen áll az exportálásra, melyet a program .csv kiterjesztéssel fog elmenteni.



4.4. ábra. Folyamatábra a program fő lépéseiről

4.4. Továbbfejlesztési lehetőségek

A program számos ponton bővíthető új funkciókkal és beállítási lehetőségekkel. Az egyik legnagyobb értéket, de nem a legtöbb fejlesztés igénylő plusz beállítás a kimeneti fájl kiterjesztése lehetne. Előfordulhat olyan use-case, amikor nem csv kiterjesztésben szeretné a felhasználó az eredményt megkapni. Ez egy egyszerű legördülő menűből választható lenne, viszont mindegyik támogatott kiterjesztésnek saját export metódust kellene implementálni. Nagyobb fejlesztést igényelne, de rendkívül hasznos lenne, hogy ha egyszerre nem csak egy, hanem több kép alapú dokumentum is kiválasztható lenne. Ebben az esetben minden dokumentum kapna egy külön fájlt, melyben a kinyert és párosított adatok találhatóak, de az mégjobb lenne, ha az azonos kulcsokat tartalmazó dokumentumok adatai egy fájlba kerülnének írásra, úgy, hogy a kulcsok csak egyszer szerepelnének, mint oszlopnév, és alattuk az összes, adott kulcshoz kinyert adat a dokumentumokról. A felhasználó élményt javítaná, ha a tallózott dokumentum (vagy dokumentumok) előnézete látszódna a felületen, valamint, ha az eredeti képen megjelenő, színes kerettel ellátott kép is az alkalmazáson belül jelenne meg, a jelenlegi felugró ablak helyett. A beállítások halmaza bővíthető lenne még azzal is, hogy a program futásának eredménye a fájlba íráson túl akár azonnal kimásolható legyen egy szövegmezőből.

A kulcs-érték párokat kereső algoritmusban még rejtőzik potenciál. Jelenleg erősen korlátozott, hiszen a kulcshoz kapcsolódó érték keresése minden esetben először jobbra, majd lefelé történik a koordináta-rendszerben. Például egy igazolványképen, ahol olyan táblázatos elrendezés van, melyen a kulcsok az értékek felett helyezkendnek el, és a kulcstól jobbra egy másik kulcs áll, alatta a hozzá tartozó értékkel, az algoritmus jelen állapotában ezt helytelenül értékelné ki. Ezt a limitációt lehetne csökkenteni, ha egyszerre vizsgálna mindkét irányba, és a környező elemeket is vizsgálva el tudná dönteni, hogy pontosan melyik érték tartozik hozzá. A vizsgálatba be lehetne vonni az objektumok közötti távolságot is, mint heurisztika. Ez egyfajta mesterséges intelligencia bevezetése lenne, hiszen a környező objektumok helyzetét elemezve és értékelve hozna döntést, mely hatással lenne a következő kulcs-érték pár keresésére is.

Összefoglalás

A szakdolgozat elsődleges célja a szövegfeldolgozás témájának körbejárása volt egy olyan alkalmazás elkészítésével, mely az OCR technológiára alapozva képes igazolványokról készített képről szöveget kinyerni, majd ezeket egy saját algoritmus segítségével feldolgozni. A szakdolgozat segítségével betekintést nyerhettem a szövegfeldolgozás és karakterfelimserés világába, ám a kutató és fejlesztő munkák során megismertem számos egyéb képfeldolgozással kapcsolatos technológiát is. A program írása során törekedtem az egyetemen tanult fejlesztési módszerek használatára, és a Szkriptnyelvek nevű kurzuson tanult Python tudásom kamatoztatására. A modern fejlesztési irányelvek betartására is figyelemmel voltam, így hasznosítottam az objektum orientált programozás alapelveit is, a projekt komponenseinek átláthatóságáról pedig a programfájlok megfelelő szintű struktúráltsága gondoskodik.

Irodalomjegyzék

[1] Image to Text Converter

https://www.imagetotext.info/ (Utolsó megtekintés: 2023.04.23)

[2] Text Extractor Tool

https://brandfolder.com/workbench/extract-text-from-image (Utolsó megtekintés: 2023.05.11)

[3] What Is Optical Character Recognition (OCR)?

https://www.ibm.com/cloud/blog/optical-character-recognition

(Utolsó megtekintés: 2023.04.23)

[4] How does OCR work?

https://aws.amazon.com/what-is/ocr/ (Utolsó megtekintés: 2023.04.23)

[5] Field typing for improved recognition on heterogeneous handwritten forms

https://arxiv.org/pdf/1909.10120.pdf (Utolsó megtekintés: 2023.04.23)

[6] What affects OCR accuracy and how to improve it?

https://www.docsumo.com/blog/ocr-accuracy (Utolsó megtekintés: 2023.04.23)

[7] The F1 score

https://towardsdatascience.com/the-f1-score-bec2bbc38aa6

(Utolsó megtekintés: 2023.04.23)

[8] Improve OCR Accuracy With Advanced Image Preprocessing

https://docparser.com/blog/improve-ocr-accuracy/

(Utolsó megtekintés: 2023.04.23)

[9] Applications for Python

https://www.python.org/about/apps/ (Utolsó megtekintés: 2023.04.23)

[10] pytesseract 0.3.10

https://pypi.org/project/pytesseract/ (Utolsó megtekintés: 2023.04.23)

[11] How to OCR with Tesseract, OpenCV and Python

https://nanonets.com/blog/ocr-with-tesseract/ (Utolsó megtekintés: 2023.04.23)

[12] EasyOCR on GitHub

https://github.com/JaidedAI/EasyOCR (Utolsó megtekintés: 2023.04.23)

[13] opencv-python 4.7.0.72

https://pypi.org/project/opencv-python/ (Utolsó megtekintés: 2023.04.23)

[14] Image Processing in OpenCV

https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html

(Utolsó megtekintés: 2023.04.23)

[15] Threshold Types in OpenCV

https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html

(Utolsó megtekintés: 2023.04.23)

[16] Sample regular expressions

https://docs.opswat.com/mdcore/proactive-dlp/sample-regular-expressions

(Utolsó megtekintés: 2023.04.23)

[17] Country Codes ALPHA-2 ALPHA-3

https://www.iban.com/country-codes (Utolsó megtekintés: 2023.04.23)

[18] Tcl (Tool Command Language)

https://www.tcl.tk/ (Utolsó megtekintés: 2023.04.23)

[19] CustomTkinter on GitHub

https://github.com/TomSchimansky/CustomTkinter (Utolsó megtekintés: 2023.04.23)

Nyilatkozat

Alulírott Marosi Márk Dániel gazdaságinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, gazdaságinformatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2023. május 12. .....

aláíras

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Janurik Viktor Bálint témavezetőmnek, aki lehetővé tette a dolgozat elkészülését. Nagyra értékelem a türelmét és megértését, ami hozzájárult az eredmények eléréséhez.

Köszönet illeti azokat az oktatókat, tanárokat és szakembereket is, akik tanítottak és inspiráltak az egyetemi éveim során. Az általuk átadott tudás és tapasztalat nemcsak a szakdolgozatban, hanem az életemben is hasznos lesz.

Nem utolsó sorban köszönetet szeretnék mondani Gyikó Richárd külső témavezetőmnek is, aki szakmai hozzáértésével, támogatásával és iránymutatásával járult hozzá a szakdolgozat elkészüléséhez.