**Szegedi Tudományegyetem**

**Informatikai Intézet**

# SZAKDOLGOZAT

**Marosi Márk Dániel**

**2022**

**Szegedi Tudományegyetem**

**Informatikai Intézet**

# Domain specifikus szöveg feldolgozása kép alapú dokumentumokon

Diplomamunka

*Készítette:*
**Marosi Márk Dániel**
Gazdaságinformatika szakos hallgató

*Témavezető:*
**Janurik Viktor Bálint**
Tanszéki mérnök

Szeged
2022

# 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

# Tartalomjegyzék

# 1. fejezet

# 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.

Ezen túlmenve, 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, viszont 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 kettő másodperc alatt minden adatom, ami a személyi igazolványomon volt, az 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 állampolgráság, és így tovább.

Szakdolgozatom célja, hogy ezt a témát körbejárjam, felkutassam a legújabb technológiákat és egy ilyen folyamatot be tudjak mutatni egy tetszőlegesen kiválasztott domainnel.

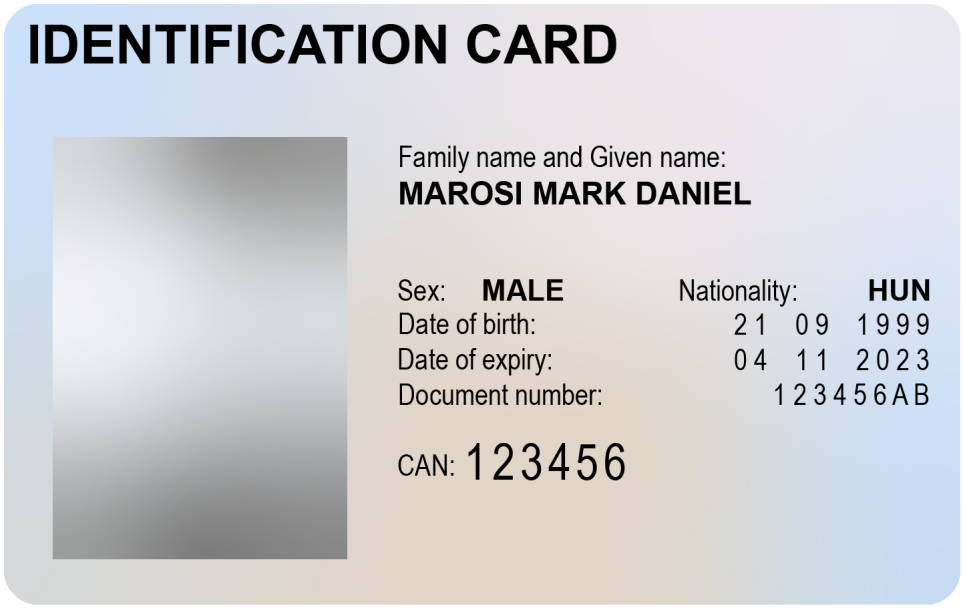
Ennek megfelelően szakdolgozatomat öt fő részre tagoltam. A dolgozat első részében a jelenleg legismertebb és legmodernebb technológiák utáni kutatásom eredményét fogom részletezni, mélyebben kifejteni azt, hogyan működnek azok a hasonló felhasználási célú redszerek, melyeket jelenleg használnak. A második részben rávilágítok a jelenleg bárki számára elérhető szövegfelismerő rendszerek hiányosságaira, továbbá bemutatom és összehasonlítom azokat a technológiákat, keretrendszereket, amelyeket érdemes lehet felhasználni a projektemben. Harmadik lépésben a projektem megvalósításának részleteit, komponenseit, felépítését fogom bemutatni, valamint megindoklom a domain választásom. Végül lépésről lépésre bemutatom a program működését egy konkrét esetre, valamint a program futásának elvárt eredményét.

# 2. fejezet

# Jelenlegi rendszerek

## 2.1. Jelenlegi rendszerek bemutatása

Egy átlagos felhasználónak kép alapú szövegbeolvasásra jelenleg telefonos vagy web alapú applikációk segítségével van lehetősége. Ez a legegyszerűbb módszer, mert könnyen kezelhető, nem igényel hozzáértést, ingyenesen igénybe vehető, gyors és megbízható. A következőkben egy általam készített, nem valós adatokat tartalmazó minta igazolványon fogom tesztelni az egyik applikációt.




2.1. ábra. minta igazolvány

A weboldalon található egy fájl feltöltésre alkalmas mező, ide kell tallózni a képet, melyből szeretnénk kinyerni a szöveget, ezután pedig el kell indítani a beolvasási folyamatot, mely körülbelül 5 másodpercet vesz igénybe. A folyamat végén egy szövegmezőből kimásolhatóvá válik a képről kinyert szöveg. Az általam feltöltött képről kinyert szöveg:

> 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, nem minden adatról azonosítható be, hogy pontosan mit jelent.

## 2.2. 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 fogok 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, hibázhatunk az adatok begépelése során, valamint ott is hibázhatunk, hogy nem jó cellába visszük fel az értékeket, elcsúszunk valahol.

A második módszer az első módszerben felvázolt három hibalehetőségből kettőt minimálisra csökkent, ezek a beolvasási és begépelési hibák. A beolvasást egy olyan technológiára alapozva végzi a szoftver, amit a szakdolgozat következő fejezeteiben fogok részletesebben bemutatni. Fontos megjegyezni, hogy egyes weboldalak lehetőséget biztosítanak egyszerre akár több kép feltöltésére, ezzel is gyorsítva a folyamatot. A technológia rendkívül pontos, emiatt eltekinthetünk attól, hogy beolvasási hiba történjen, azaz eleve rossz adat kerüljön a táblázatba. A második, 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 elrontsunk 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 ugyan azon a technológián alapulva. 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 excel 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 30 06 1979 alakot vesz fel a kiolvasás után 1979.06.30 vagy egyéb tetszőleges formára tudná hozni, tovább csökkentve a manuális teendőket.

# 3. fejezet

# Az OCR technológia

## 3.1. Az OCR működése

Az OCR, azaz Optical Character Recognition (magyarul Optikai Karakterfelismerés) egy olyan technológia, amely képes 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 tudjuk, de szerkesztni, szöveget keresni vagy kimásolni belőle nem tudnánk. Ha egy olyan szkennerünk lenne, amelyben OCR alapú szövegfelismerés is lenne, 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. Az első lépésben egy osztályozási folyamat történik, ahol a kiválasztott képet vizsgálva a világos területeket háttérnek, a sötét területeket pedig szövegnek minősíti. Ebből kifolyólag az OCR pontosságát tovább javíthatjuk azzal, ha már a karakterfelismerés előtt az inputként választott képet fekete-fehérré alakítjuk.

Második lépésben a szövegként minősített területeken egy olyan keresés indul, amelynek célja az alfabetikus karakterek (betűk) és numerikus karakterek (számok) azonosítása, ez történet karakterről karakterre, vagy szavanként (karakter láncokként) is. Az azonosítás egyik leggyakoribb módszere a mintaillesztésre alapul. Ennél a módszernél egy olyan adathalmazból dolgozik az algoritmus, amely sok különböző betűtípus- és szövegképmintát 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 akkor igazán pontos, ha az input egy olyan kép, melyen ismert betűtípusokkal jelenik meg gépelt, nyomtatott szöveg, mivel a betűtípusok és kézírási stílusok száma végtelen, és lehetetlen minden típust az adatbázisban rögzíteni. Amennyiben kézzel írt szöveget adunk bemenetként, akkor a pontos eredmény eléréséhez egy olyan algoritmusra van szükség, amely figyelembe veszi a karakterek jellemzőit is. Ilyen jellemzők például a betű írására használt vonalak, azok irányai, elhelyezkedései, metszéspontjai, görbületei és hurkai. Ezen tulajdonságokat az algoritmus minden felismerhető karakterről tárolja, majd a keresett alakzatot is felbontja ugyan ezekre, és megkeresi a tárolt karakterek közül azt, amellyel a legtöbb jellemző megegyezik. Ezt a folyamatot Intelligent Character Recognition (ICR), magyarul Intelligens Karakterfelismerés névvel illetik.

Utolsó lépésben a dokumentum teljes szerekezeti képének függvényében a felismert karakterek önmagukban, szavakba, mondatokba vagy szövegblokkokba rendezve kerülnek tárolásra.

## 3.2. Az OCR pontosságának mérése

Az előzőekben bemutattam, hogyan képes az OCR egy szöveget tartalmazó képet gépi szöveggé alakítani, de felmerülhet bennünk a kérdés, hogy mégis mennyire pontos az eredmény, amit kapunk egy ilyen konverzió során. A karakterfelismerés csupán képpontról képpontra vizsgálja a képet, és a betűk alakjából vonja le a végső következtetést, arra viszont nem képes, hogy a dokumentum teljes kontextusát felismerve megállapítsa, hogy a szöveg, amit kinyert, az pontosan mit is jelent, és 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 eltalálnia az szövegfelismerőnek, úgy a pontosság 99%. Tehát egyszerű arányosítással is kiszámolható egyfajta pontosság.

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

### 3.2.1. CER - Character Error Rate (Karakter hibaarány)

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

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

Ahol 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 számkarakternek állapította meg, továbbá a 2-es számot Z betűnek, így 2 transzformációra lesz szükségünk, tehát T=2.
A helyesen felismert karakterek száma 9 (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.

### 3.2.2. WER – Word Error Rate (Szóhibaarány)

Hasonlóan a CER-hez, 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 metrikáját méri, nem a betűkét, de ha belátjuk, hogy ugyan azon betűk sorozatából kapjuk a szavakat, akkor jogosan feltételezhetjük, hogy a WER és a CER metrikák jól korrelálnak egymással.

A WER mérésére ugyan azt a képletet használjuk, mint a CER érték méréséhez, 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.

### 3.2.3. További metrikák az OCR pontosságának megállapításához:

- SER - Symbol Error Rate (Szimbólum hibaarány):
  - Ez a metrika kifejezetten azt vizsgálja, 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.
- 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 előbbiekben 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.

## 3.3. 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 egy képről vagy kép alapú dokumentumról hivatott szöveget kinyerni, így a pontos eredmény első és legfontosabb feltétele a megfelelő minőségű bemenet nyújtása. Nézzük, mik azok a leggyakrabban előforduló körülmények, amik rontják az OCR pontosságát.

- 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
  - Emberi 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.

– A kép méretének és felbontásának helyes megválasztása:

  - A karakterfelismerés pontossága nagyban függ a bemeneti kép pontsűrűségétől (DPI).
  - Általában egy 200-300 közötti DPI-vel rendelkező kép a legmegfelelőbb, ennél kisebb értéknél bizonyos karaktereknél előfordulhat, hogy hibásan kerülnek felismerésre, nagyobb értékeknél pedig szükségtelenül nagy méretű kép lesz a bemenet, az OCR pontossága ezen intervallum felett nem javul számottevően.

– Kontraszt növelése és színek eltüntetése:

  - Mivel az OCR egyik lépésre – 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 a kontraszt minél nagyobb, 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 csak fekete-fehér színeket tartalmaz. Kontrasztot ma már bármilyen képszerkesztő alkalmazással tudunk növelni, illetve filterek alkalmazásával fekete-fehérré tudjuk alakítani a színes képeket.

- Ferdeségkorrekció:
  - Amennyiben ferdén fotózott vagy szkennelt képek nagy mértékben csökkentik az OCR hatékonyságát, hiszen a karaktereket meghatározott vonalakból és alakzatokból próbálja felismerni, és 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 simítási, zajmentesítési folyamatokkal, melyek szintén megtalálhatóak a leggyakoribb képszerkesztő programok funkciói között.

# 4. fejezet

# A program megvalósítása

## 4.1. OCR könyvtárak összehasonlítása Pythonban

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

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

### 4.1.1. 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.

Néhány példa a pytesseract importálása után rendelkezésünkre álló metódusok közül:

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 leginkább a jelenlegi rendszerek bemutatása fejezetben egy internetes alkalmazás 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

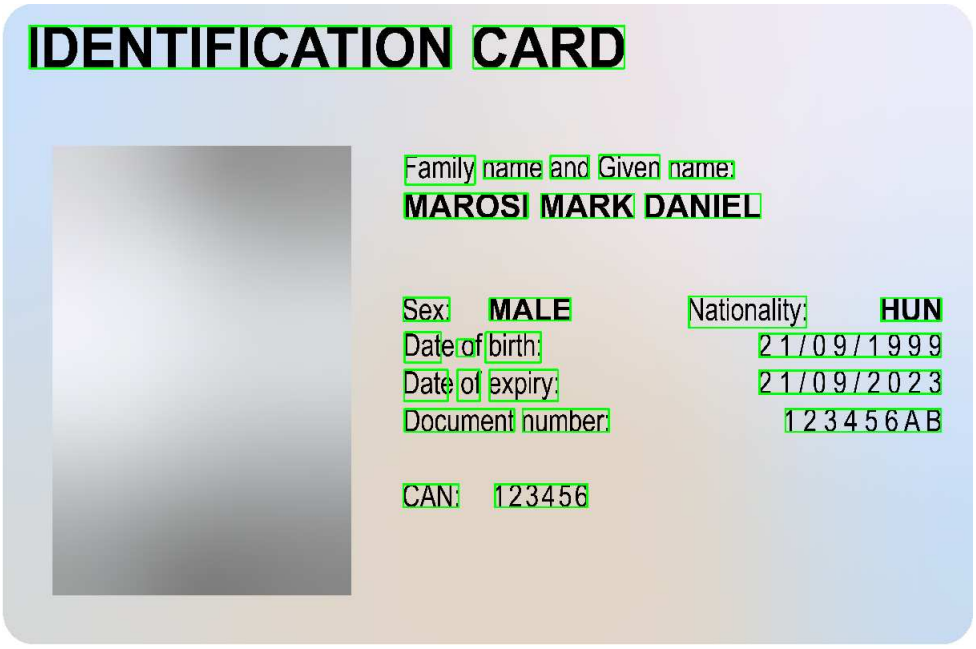
Az *image_to_boxes* metódus minden egyes felismert karaktert egyesével, az őt körülhatároló négyzet bal felső sarkának koordinátáival, valamint a négyzet szélességével és magasságával adja vissza listába rendezve. Egy részlet az eredményhalmazból:

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, az 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, amely a bemenetként adott igazolványképen a felismert szövegeket határoló téglalapokat kirajzoltattam az OpenCV nevű python könyvtárban definiált metódusok segítségével. Az OpenCV könyvárat a későbbiekben ismertetem. A függvény egy paraméterrel rendelkezik, amely az *image_to_data* függvény eredményét várja dictionary struktúrában. A függvény kimenete egy új ablakban megnyíló kép, amelyen a bemeneti kép látható úgy, hogy a rajta található, egyben felismert szövegrészek körül vannak rajzolva a szöveget határoló téglalap élei mentén.




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

```
def generate_image_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()
```

### 4.1.2. Az EasyOCR könyvtár bemutatása

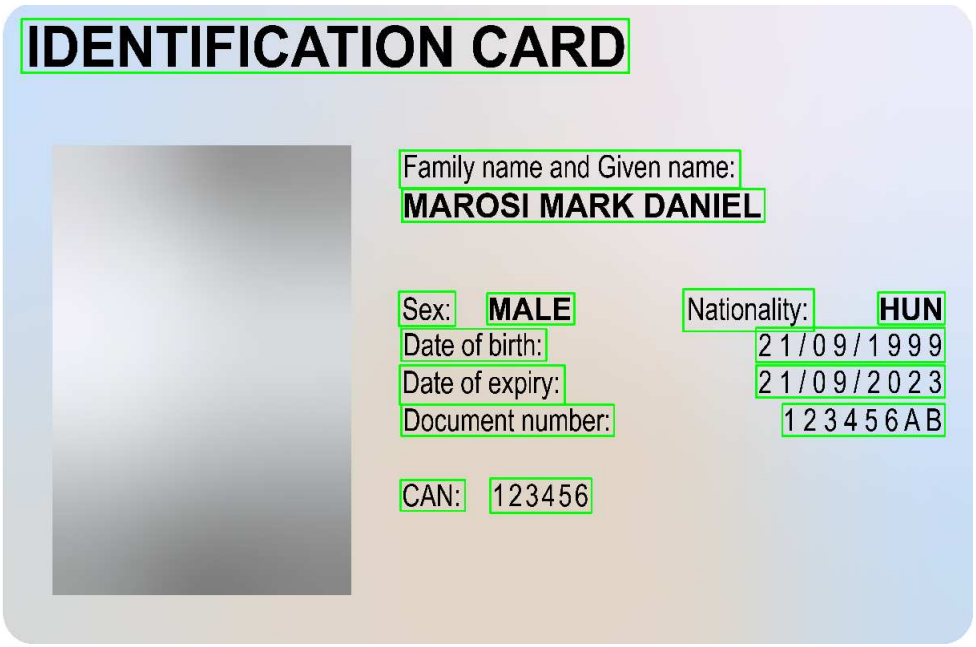
Az EasyOCR könyvtár is egy nyílt forráskódú, ingyenesen használható OCR könyvtár. A Tesseracttól független, eltérő az implementációja, összességében nagyobb pontosságot képes biztosítani bizonyos esetekben, hátulütője az, hogy nagyobb inputokra lassabb, mint a Tesseract.

Az EasyOCR importálása után először példányosítani kell egy objektumot az easyocr.Reader konstruktorhívással, majd rendelkezésünkre állnak az EasyOCR metódusai. Ezek közül a szakdolgozat szempontjából a *readtext* metódus a legfontosabb, ennek egyetlen paramétere az 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.




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

```
def generate_image_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 legfőbb 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.

## 4.2. 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. Mivel a bemeneti dokumentum lehet színes, így először a színek 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. A mi 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 nagy mértékben 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.

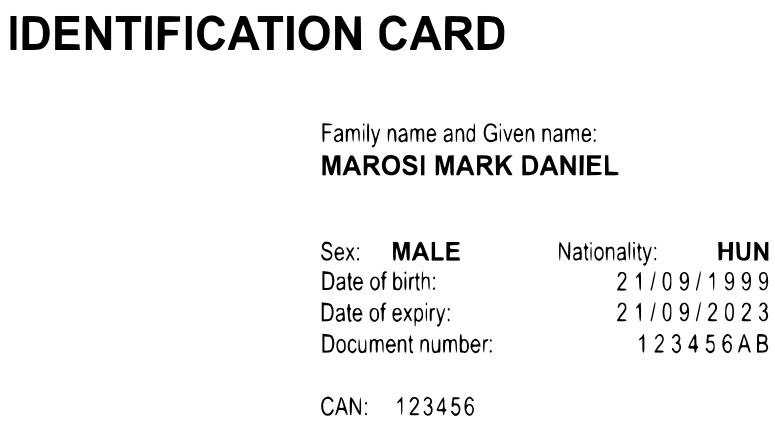
```
def remove_noise(image):
    return cv2.medianBlur(image, 5)
```

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 képbet kétszintű (bináris) képpé konvertáljanak. A függvény paraméterei a kép, egy küszöbérték (threshold), egy maximum érték (maxval), és a küszöbérték típusa. Különböző célokra különböző küszöbérték típusok léteznek, számomra a THRESH_BINARY a legmegfelelőbb. A THRESH_BINARY küszöbérték típus számítási elve:

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

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 maxvalue az a szín, amit a küszöbértéknél nagyobb, azaz kiszűrni kívánt képpontok (ebben az esetben a dokumentum háttere) fog felvenni, ez a mi esetünkben 255 lesz, azaz fehér.




4.3. 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 felismerhetőekek legyenek, és ne maradjon a szövegfelismerés szempontjából irreleváns jelentést hordozó adat a dokumentumon.

## 4.3. 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 végig iterálunk 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]

## 4.4. 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 ezt leszámítva, 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 igazából 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, aminek 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 kulcs halmazhoz. Például igazolványszámot a szabvány alapján (adott mennyiségű betűk és számok, meghatározott sorrendben), vagy dátumokat a benne szereplő számok és elválasztó jelek 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 még 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 meg lett volna adva, 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 sablonnál. 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 végig iterál 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 legjobban balra és legjobban 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 jelezve legyen, hogy ez nagy eséllyel egy olyan szöveg, mely csak tájékoztató jellegű, nem pedig egy fontos adat. Ilyen például a példa igazolványon 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, jelezvén, 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, tőle balrább 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 változó 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, amik 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, ami 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, hogy kulcs, mert csakis 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 előbb 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, hogyha nem találunk hozzá tartozó értéket, akkor se vizsgáljuk többet. A metódus ezután 2 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 kulcs keresés mindig a bal felső objektumokból indul ki, így teljesen felesleges lenne balra illetve felfelé történő keresés, hiszen ott már nem találhatunk semmit. 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 ob-

jektumokat 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 addíg, 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 már el volt 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

    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. Így végeredményben a dictionary csak az egymással párba állított szövegeket fogja tartalmazni. Erről meggyőződhetünk, ha végigiterálunk rajta és kiíratjuk minden elemét:

```
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.

## 4.5. 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. Utó-

lagos 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 egészébena mintára illeszkedik, és megfelel a követelményeinknek, egyébként elutasító. Tudjuk, hogy az olyan adatok, mint a nemzetiség kódja, a nem, a dátumok és a dokumentumszámok mind olyanok, melyek 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 elválasztó jelek is lehetnek eltérőek. Ehhez az interneten kerestem már kész reguláris kifejezéseket, és azt kicsit átalakítva a programomba három konstans változóba felvettem. Az egyik az év, hónap, nap, a másik a nap, hónap, év, a harmadik pedig a hónap, nap, év sorrendű megfelelőjét tárolja a reguláris kifejezésnek. Mind a három esetben az évet 4 számjegy, a hónapot és a napot pedig 2-2 számjegy jelöli. Egyéb eseteket nem vettem figyelembe, mivel ez a három nagyon nagy százalékban lefedi a különböző igazolványokon található formátumokat.

Az nemzetiséget jelölő ország kódjára saját reguláris kifejezést írtam. Az országok kódjai az ISO-3166 szabvány alapján 2 vagy 3 betűből állhatnak. 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 2 és 3 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 4 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 4 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_string = value.replace(' ', '')
        if matches_any_regex(cleaned_string):
            new_items_dict[key.replace(':', '')] = cleaned_string
        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 felhasználó által megadott mappába fog létrehozni egy csv kiterjesztésű fájlt. 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 olvasható legyen a fájl.

```
def export_data_to_csv(path, data):
    with open(path, 'w', newline='', encoding='utf-8') as file:
        writer = csv.writer(file, delimiter=';')

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

## 4.6. Összefoglalás

A szakdolgozatban leírt projekt lényege, hogy segítséget tudjunk nyújtani a hallgatóknak akár a Cisco hálózatépítő kurzusban, vagy akár a Cisco ipari vizsgára való készülésben. A fejlesztés a szakdolgozaton túl is folytatódik, célunk, hogy egy olyan rendszer álljon a hallgatók rendelkezésére, amivel produktívan lehet tanulni, kísérletezni.

A fejlesztés során lehetőségem nyílt nem csak csapatban, de új technológiákkal is dolgoznom. Köztük a RabbitMQ, aminek használata számomra újdonság és kihívás volt és a Python objektumorientált használata, amire ezelőtt még nem volt szükségem, lehetőségem. Illetve, eddigi tudásomat is lehetőségem nyílt kamatoztatni, a git verziókezelés, a

gitlab-runner és a hozzá tarozó CI/CD leírók használata, a docker és docker-compose, és a hyper-v virtualizáció terén.

# 5. fejezet

# Adatbázis sémák

## 5.1. Adatbázis sémák

A program a következő adatbázisokkal dolgozik:

- lab_tasks
- correct_commands
- running_configs
- startup_configs

A lab_tasks tábla tartalmazza az elérhető feladatok címét, a Cisco tananyag fejezet azonosítóját, amivel a hozzá tartozó tananyag könnyen megtalálható, és COM[1-6] oszlopokat, amelyeknek az értéke a feladat szerinti eszköz hostneve.

A correct_commands tábla azokat a parancsokat tartalmazza, amelyeket a különböző COM portokon ki kell adni, a feladat helyes elvégzéséhez.

A running_configs tábla azokat a "show running config" parancs kimeneteket tartalmazza, amelyeket a helyesen konfigurált eszközöknek produkálnia kell.

A startup_configs tábla pedig a hibaelhárítási feladatokhoz tartozó helytelen beállításokat előállító parancsokat tartalmazza, amelyeket a feladatmegoldás előtt a program futtat az eszközön.

5.1. ábra. Séma relációs ábrázolása

# 6. fejezet

# Az alkalmazás működése

## 6.1. Az alkalmazás működése

A program belépési pontja a main.py, ami indulásakor példányosítja a MetaCommunication osztályt. Ez a példány becsatlakozik a RabbitMQ-ba, ahol egy message queue-n figyel, és vár egy felhasználó csatlakozására. A csatlakozást követően készen áll a használatra.

Egy felhasználó csatlakozásakor a MetaCommunication osztály user_connection() metódusa meghívódik, ezzel példányosítva a User osztályt. A User osztály konstruktora a szükséges adattagokat beállítja (felhaszáló azonosítója, kiválasztott feladat (ha van), helyes parancsok listája, százalékos inkremens).

6.1. ábra. A program indulása előtti és utáni message queue-k

A példányosítás után a vezérlés visszatér a MetaCommunication osztály user_connection() metódusához, ami a Thread típusú User példányt elindítja. Ezzel a User objektum run metódusa meghívásra kerül, ami a programszálat elindítja. Annyi felhasználóhoz tartozó Logger osztálypéldányt készít, ahány COM porton keresztül kommunikál az eszközökkel, továbbá ugyanennyi QueueListener osztálypéldány is létrejön, ha a felhasználó választott feladatot. Azaz, a jelenlegi felállás szerint egy felhsználóhoz – a hat COM porthoz – kétszer hat programszál tartozik, így a MetaCommunication és fő szállal együtt tizennégy vagy nyolc szálon fut a program.

6.2. ábra. A programszálak vizualizálva
(Készült a PyCharm Concurrent Activities Diagram segítségével)

A QueueListener példányok a hozzájuk tartozó message queue-kat figyelik, és az azon keresztül érkező parancsokat fogadják. Feldolgozásra meghívják rájuk a validate_commands() metódust, ami kiértékeli azt.

Amikor a felhasználó elvégezte a feladatot, a szintjelző 100%-ra vált, és üzenetet küld a felhasználónak erről. Ekkor a message queue-kat figyelő szálak feleslegessé válnak, ezért ezeket egyesítjük az őt példányosító szálakkal. Ez úgy érhető el, hogy a QueueListener példány is_waiting_for_join tulajdonságát 1-re állítjuk, ami megállítja a futtatását.

A szemétgyűjtést a Python beépített automatikus szemétgyűjtője végzi, ami referenciaszám alapján azokat az objektumokat, amikre már nem hivatkozik semmi, törli a memóriából. A dokumentációban leírtak alapján a szemétgyűjtés explicit meghívásra kerül, ha az újonnan létrehozott objektumok száma meghaladja a már jelenleg memóriában lévők 25%-át. Képletesen felírva:

$$current\_objects \times 0.25 < new\_objects \Rightarrow garbage\_collection \tag{6.1}$$

Ez a százalékos határ a program esetében ideális. A program indulásakor létrejön a MetaCommunication egy példánya. Egy felhasználó bejelentkezésekor kétszer hat objektum jön létre (hat QueueListener és hat Logger a hat COM porthoz). Amikor a felhasználó kijelentkezik, a következő felhasználó bejelentkezéséig két eset állhat fenn.

1. Az automatikus szemétgyűjtés nem fut le, tehát a következő felhasználó bejelentkezésekor az előző felhasználóhoz létrehozott objektumok még a memóriában vanank.

2. Az automatikus szemétgyűjtés lefutott és az előző felhasználó objektumai a következő bejelentkezésekor már nincsenek a memóriában.

Az első eset az, ami vizsgálatot igényel: Egy felhasználó aktív kapcsolata során 1 + 2 x 6 = 13 objektum van a memóriában, ezek pedig nem törlődtek. A következő felhasználó bejelentkezésekor 2 x 6 objektum keletkezik. A képletet alkalmazva:

$$13 \times 0.25 < 12$$
$$3.25 < 12$$

Az egyenlőtlenség igaz, tehát a szemétgyűjtés legkésőbb a következő felhasználó bejelentkezésekor meghívódik.

# 7. fejezet

# Továbbfejlesztési lehetőségek

## 7.1. Továbbfejlesztési lehetőségek

A szakdolgozat keretein belül implementálásra nem került funkcionalitások:

- A jelenlegi kód a hálózati erőforrásokhoz (MySQL adatbázis szerver, RabbitMQ) hardcode-oltan csatlakozik, ezért a kód átírása nélkül más fizikiai infrastruktúrára nem lehet telepíteni.
- Az adatbázisba leképezett CCNA kurzushoz kapcsolódó laborfeladatok száma igen alacsony, mivel a Word dokumentumokból és PDF fájlokból a konfigurációs lépések kiemelése nem szkriptelhető, ehhez nagy időbefektetés szükséges.
- Parancsok kiadása az eszközökön a modul által:
  - A felhasználó kijelentkezése után, az eszközök konfigurációjának törlése és az eszközök újraindítása az "indulási konfiguráció" (startup configuration) betöltésével.
  - A felhasználó által meghívható teljes rendszer visszaállítás arra az esetre, ha valamelyik felhasználó által kiadott parancs elérhetetlenné teszi a rendszert (pl.: jelszó beállításakor félreütött karakter miatti kizárás az eszközről).
- A projektben megírt osztályok és metódusaik, mind egy forrásfájlban találhatóak, amely nehezíti a kód olvasását. Ennek több forráskódra bontása, osztályok mentén, ezek csomagokba rendezése és névterek használata, nagyban javítaná a projekt átláthatóságát.

– Az idő előrehaladtával a Cisco tananyag verziózásának követése (a jelenlegi rendszer a 6-os verzión alapul). Ez megvalósulhat akár több tábla használatával, vagy akár több adatbázis séma létrehozásával. Ennek megvalósítása biztosítaná, hogy a modul a jövőben ne váljon elavulttá.

# Irodalomjegyzék


[1] Cisco Networking Academy,

v6 *Connecting Networks, Introduction to Networks, Routing and Switching Essentials & Scaling Networks* Instruktori forrásfájlok

[2] Python projekt struktúrálás és PEP

*https://docs.python-guide.org/writing/structure/*

[3] MySQL 8.0 Reference, SQL parancs szintaxisok,

*https://dev.mysql.com/doc/refman/8.0/en/*

[4] MySQL Cursor Objects,

*execute és fetch metódusok használata és paramétereik https://mysqlclient.readthedocs.io/user_guide.html*

[5] Docker Dokumentáció, parancsok szintaxisa és paramétereik,

*https://docs.docker.com/*

[6] Python 3 threading – Thread-based parallelism,

*https://docs.python.org/3/library/threading.html*

[7] Python 3 Classes,

*https://docs.python.org/3.9/tutorial/classes.html*

[8] MQTT protokoll

*https://mqtt.org/*

[9] RabbitMQ Tutorials,

*Work Queues, Publish/Subscribe https://www.rabbitmq.com/getstarted.html*

[10] Pika AMPQ protokoll implementációs dokumentáció

*https://pika.readthedocs.io/en/stable/*

# Nyilatkozat

Alulírott Kersmájer István szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztési Tanszékén készítettem, gazdaságinformatika 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 / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2023. április 17. .....

aláíras

# Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Janurik Viktor témavezetőmnek, aki a fejlesztés minden lépésében hasznos tanácsokkal látott el. A segítségéért a tesztkörnyezet összeállításában, a sok konzultációért és szakmai tanácsaiért, amelyek nélkül ez a projekt nem jutott volna tovább az "initial commit"-on.

Köszönöm továbbá két fejlesztőtársamnak, Orbán Veronikának és Csóti Zoltánnak, akikkel a fejlesztés produktívan tudott haladni, és kódváltozásaimhoz igazították saját munkájukat. A hallgatóknak, akik használni fogják az új rendszert, visszajelzéseikért az esetleges problémákról, és további fejlesztési ötleteikért.