**Miskolci Egyetem**

Gépészmérnöki és Informatikai Kar

Alkalmazott Matematikai Intézeti Tanszék

**Szakdolgozat**



**Interaktív megjelenítő eszköz pénzügyi adatok elemzéséhez**

**Készítette:**

Bencze Zsombor

LP5J4B

BSc programtervező informatikus hallgató

**Témavezető:**

Piller Imre

Miskolc, 2023

|  |  |  |
| --- | --- | --- |
| ***Miskolci Egyetem***  ***Gépészmérnöki és Informatikai Kar*** |  | ***Alkalmazott Matematikai  Intézeti Tanszék***  *3515 Miskolc-Egyetemváros* |
| *Szak:* ***Programtervező informatikus*** *Szakirány: Webtechnológia* |  | ***Intézmény azonosító: FI 87515*** |

***Szakdolgozat feladat kiírás***

***Bencze Zsombor***

*Neptun-kód: LP5J4B*

*BSc programtervező informatikus hallgató részére*

|  |  |  |
| --- | --- | --- |
| *A tervezés tárgyköre:* | Webfejlesztés | |
| *A feladat címe:* | Interaktív megjelenítő eszköz pénzügyi adatok elemzéséhez | |
| *A feladat részletezése:*   1. A szakdolgozat célja olyan interaktív, dinamikus megjelenítési módok tervezésének, működésének és használatának a bemutatása, amelyekkel egyazon idősor különböző részei, különböző forrásból származó idősorok, az azokból származtatott értékek összehasonlíthatók, az aggregáláshoz használt paraméterek rugalmasan változtathatók. 2. A pénzügyi adatok (például tőzsdei árfolyamok) elemzéséhez elengedhetetlen, hogy a rendelkezésre álló információk a szakértők számára könnyen áttekinthető formában rendelkezésre álljanak. 3. Az alkalmazásnak webes környezetben, szerver-kliens architektúrának megfelelően kell elkészülnie. Ehhez szerver oldalon Node.JS programnyelvet fogok használni. Míg a kliens weboldal megvalósításához HTML5, CSS, JavaScript programnyelveket kell használni. Az alkalmazás adatai adatbázisban lesznek eltárolva | | |
|  | | |
|  | | |
| Tervezésvezető: Piller Imre | | egyetemi tanársegéd  Alkalmazott Matematikai Intézeti Tanszék |
| *Konzulens:* | |  |
| *A szakdolgozat kiadásának időpontja:* | | 2023.02.21. |
| *A szakdolgozat beadásának határideje:* | | 2023.05.26. |
| Miskolc, 2023.05.26. | | Piller Imre  *egyetemi tanársegéd* |

**Tartalomjegyzék**

1. Bevezetés 1
2. Elektronikus dokumentumok 2
   1. [Átvihetődokumentum formátumok . . . . . . . . . . . . . . . . . . . . 2](#_TOC_250021)
   2. [A dokumentum tartalma . . . . . . . . . . . . . . . . . . . . . . . . . . 3](#_TOC_250020)
   3. [Optikai karakterfelismerés . . . . . . . . . . . . . . . . . . . . . . . . . 3](#_TOC_250019)
   4. [Gépi tanulási módszerek . . . . . . . . . . . . . . . . . . . . . . . . . . 4](#_TOC_250018)
3. A dokumentum strukturális elemzése 5
   1. [A PDF dokumentum beolvasása . . . . . . . . . . . . . . . . . . . . . . 5](#_TOC_250017)
   2. [Margók becslése . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6](#_TOC_250016)
   3. [Paragrafusokra bontás . . . . . . . . . . . . . . . . . . . . . . . . . . . 8](#_TOC_250015)
   4. [Szavakra bontás 10](#_TOC_250014)
   5. [Karakter szintűelemzés 11](#_TOC_250013)
   6. [Bonyolultabb dokumentumok 12](#_TOC_250012)
   7. [Képek és szövegek megkülönböztetése 14](#_TOC_250011)
   8. [Karakterek felismerése 15](#_TOC_250010)
   9. [Küszöbértékek meghatározása 18](#_TOC_250009)
   10. [Képfelismerés neurális háló segítségével 19](#_TOC_250008)
   11. [Program tesztelése 19](#_TOC_250007)
       1. [Futási időbecslése 19](#_TOC_250006)
       2. [A paragrafusok feldolgozásának a helyessége és a szöveg helyes- ségének a viszonya 20](#_TOC_250005)
4. Partícionálás és klaszterezés 23
   1. [Oldalak partícionálása 23](#_TOC_250004)
   2. [Összefüggőképterületek detektálása 26](#_TOC_250003)
5. A vizsgálatokhoz készített programok 30
   1. [Képek vágása és további feldolgozása 30](#_TOC_250002)
   2. [Jupyter munkafüzetek használata 31](#_TOC_250001)
6. Összefoglalás 32

[Irodalomjegyzék 33](#_TOC_250000)

**Bevezetés**

A szakdolgozatom egy weboldal, amely alkalmas többféle pénzügyi adat elemzéséhez, ehhez különféle befektetési alapokat használok, továbbá egy grafikonrajzolót. Ennek az oldalnak a szervezeti felépítését, elkészítését és az alkalmazás működését mutatom be.

A pénzügyi adatok, vagy tőzsdei árfolyamok megismerése lehet nagyon egyszerű, de gyakran nehézséget jelent azoknak, akik előszőr próbálkoznak meg vele a hétköznapokban. Ezért a dolgozatom olyan lehetőségeket mutat be, amelyekkel átláthatóbban lehet megérteni és kezelni őket.

Ezek a lehetőségeket kétféle alternatívára osztottam szét. Az egyik opció a részletes leírása és ismertetése a befektetési alapoknak, kiegészítve egy táblázattal, amely ábrázolja a hozam-kockázat profilt, továbbá egy alap által elért éves nettó hozam sáv. A másik lehetőségként külön oldalon található grafikonrajzoló használható. Lehetőség van kezdési és végpontot megadni, így különböző intervallumokat kiválasztva tudjuk vizsgálni az adott befektetési alapot. A dolgozat az utóbbira fektet nagyobb hangsúlyt több okból kifolyólag is. Ezt elsősorban az indokolja, hogy az így nyert adatokat rugalmasabban tudja kezelni a felhasználó, mivel egyszerre több kiválasztási lehetőség áll rendelkezésre, amivel részletesebb adathalmazt nyerhető ki.

A megjelenítéshez és az algoritmusok fejlesztéséhez több programozási nyelvet is került választásra, többek között HTML5 képezi a weboldal vázát, CSS a weboldal megjelenítését képezi, Node.JS amellyel a weboldal szervere működik és végezetül JavaScript nyelven válik elérhetővé a grafikonrajzoló.

Több grafikonrajzoló és eszközkezelő weboldal elérhető a különböző befektetési vállalatoknak az interneten. Ebből kifolyólag felmerülhet a kérdés, hogy akkor miért volt szükség még egy weboldal elkészítésére? Többek között erre a kérdésre is választ kaphatunk a szakdolgozat elolvasása után.

**Interaktív megjelenítő oldalak**

A statisztikai elemzés egyik alapvetőmódszere az összehasonlítás. Ez történhet különböző számítások révén, illetve grafikus ábrázolás segítségével. A grafikus ábrázolás előnyei, hogy jól szemléltethetőek, arányokat is érzékeltet és

## Átvihetődokumentum formátumok

Az elektronikus dokumentumoknak különféle formátumai vannak. A dolgozat szem- pontjából kiemelt fontosságú a PDF (*Portable Document Format*), mivel az tekinthető a leginkább elfogadott, és mint a neve is mutatja, valószínűleg a legtöbb platformra átvihetődokumentum formátumnak. Az aktuálisan használt speciﬁkációját az*Adobe* nevűcég kezeli [7].

Gyakran használt dokumentum formátum az RTF (*Rich Text Format*), a DOC, a DOCX amelyet elsősorban a MicroSoft Oﬃce programcsomag tett népszerűvé. E mellett további nyílt dokumentumformátumokkal is találkozhatunk, mint például az ODT (*Open Document Text*).

Elterjedtségét illetően a PDF vetélytársának a HTML (*HyperText Markup Langu- age*) nyelven írt dokumentumokat tekinthetjük, viszont céljukban és kezelési módjuk- ban is jelentősen különböznek.

A konverzió HTML-ből PDF formátumba a megfelelőstílusbeállításokﬁgyelembe- vételével mindig elvégezhető. A nehézséget a másik irány jelenti, mivel az sosem tud tökéletesen működni, mivel a HTML nyelv a PDF eszközkészletének jelentős részét nem tartalmazza. Ehhez kapcsolódóan találhatunk kutatási eredményeket a szakirodalom- ban [9].

A PDF formátum elviekben megjelenítőeszköztől függetlenül nagyon hasonlóan je- lenik meg. (Két megjelenítőeszközön a dokumentumok megjelenítésének azonosságát a hagyományos tekintetben vesszük, ezért szerepel a megfogalmazásban az, hogy nagyon hasonlóak lesznek.) Ezt úgy képes elérni, hogy (szöveges dokumentum esetén) a szöve- ges tartalom mellett akár karakter vagy ékezetek szintén is tárolja, hogy a megjelenített alakzatok melyik részének hova kell majd kerülnie. Ennek ilyen formában megvannak az előnyei, amelyek egyúttal a dolgozat célkitűzésére, és a választott módszerekre is választ igyekszik adni.

A szövegelemek pontos helyének tárolása előnyös, mert így a szöveg tördelésével nem szükséges később már foglalkozni. Ez a megjelenítés sebessége és a dokumen- tum egysége szempontjából is lényeges. Azt mondhatjuk, hogy a dokumentum egy egységként tárolja a dokumentum helyes megjelenítéséhez szükséges adatok jelentős részét. Ez többek között azt is jelenti, hogy általában nem kell attól tar- tani, hogy egy adott betűtípus, vagy szimbólum a megjelenítőrendszeren nem lesz elérhető.

*•*

Mivel több adatról van szó, ezért hátránynak tekinthető, hogy maga a doku- mentum mérete nagyobb lesz. Ez indokolatlanul nagy fájlméretet is jelenthet, mivel ha csak a dokumentumban tárolt lényegi információkra van szükségünk, akkor egy PDF fájl mérete több ezerszerese lehet az egyszerűszöveges tárolási módénak.

*•*

A PDF fájlokban a szövegek kijelölése időnként körülményes lehet. Gyakran talál- kozni például a sortörésekből adódó másolási hibákkal, a lemaradt ékezetekkel, a karakterek indokolatlan sorrendjével vagy akár nem látható, helytelen karakterek megjelenésével.

*•*

Attól függően, hogy a PDF milyen eszközzel készült, tartalmazhat elválasztáso- kat, speciális tördelési módokat.

*•*

## A dokumentum tartalma

A dokumentumokban lévőadatok egy része a tartalmat, másik része pedig a hozzá szükséges formázást írja le. A dolgozat célja, hogy bemutasson olyan módszereket, amelyekkel ez a kettőelválasztható egymástól. Az optikai karakterfelismerőrendsze- rek (OCR,*Optical Character Recognition*) tulajdonképpen ezt a problémát oldják meg általánosan [11]. A szakirodalomban ehhez kapcsolódóan teljes dokumentumok struk- turális elemzésére is láthatunk példákat, ahol a dokumentum képének szegmentálása adja az elemzés alapját [4].

A PDF fájlformátum közismert, számos API is elérhetőhozzá, viszont elég bonyolult (a benne rejlőszámos lehetőség miatt). Ahogy korábban említésre került, a karakterek elrendezése, tördelése elválaszthatja az egyébként egymáshoz tartozó adatokat, így a szövegblokkok kiolvasása önmagában nem elegendő. Arra tehát nem tünt érdemesnek hagyatkozni, hogy a PDF fájlban lévőadatokat valamilyen függvénykönyvtár segítsé- gével közvetlenül ki lehessen nyerni. Helyette inkább az a feltételezés tünt megfelelőnek, hogy egy PDF dokumentum egy minimális mennyiségűzajjal terhelt bemeneti képnek tekinthető, amelyből adatokat a hagyományos képfeldolgozási módszerekkel ki lehet nyerni.

## Optikai karakterfelismerés

Az optikai karakterfelismerési problémának számos változatai és előfordulási módjai vannak. A módszereket például az alábbi szempontok alapján csoportosíthatjuk.

Az elemzett szöveg alapján el szoktak különíteni géppel- és kézzel írott karakte- rekre kifejlesztett karakterfelismerőmódszereket.

*•*

Jelentős különbségek vannak annak megfelelően, hogy mennyi szimbólumot tar- talmaz az az ábécé, amelyből a szöveg karakterei kikerültek. Különbségek adód- nak továbbá a szöveg elrendezésének változatosságából, a szövegképben előforduló hibákból, zajokból.

*•*

Az alkalmazott heurisztika alapján beszélhetünk determinisztikus és sztochaszti- kus módszerekről. Utóbbi olyan esetekben fordul előgyakrabban, amikor a prob- lémát egy optimalizálási feladatra vezetik vissza, és a keresési tér mérete miatt véletlenszerű, futtatásonként különbözőeredményeket adhat az algoritmus.

*•*

A dolgozat arra törekszik, hogy a kifejezetten előnyösnek mondható képfeldolgozási problémát az elérhetőeszközökkel minél egyszerűbben, és elegánsabban oldja meg. A képfeldolgozással kapcsolatos problémákat az*OpenCV*(*Open Computer Vision*) nevű függvénykönyvtár segítségével oldottam meg [1].

## Gépi tanulási módszerek

A képfeldolgozási feladat is egy adatfeldolgozási probléma. A hagyományos módszer- tan szerint először egy előfeldolgozási, zajszűrési lépés szükséges, amit a jellegvektorok kinyerése követ majd. Mivel a PDF fájlokban fehér zajra, megvilágítási hibákra, folt- szerűzajokra nem kell számítanunk, ezért a feldolgozást gyakorlatilag kezdhetjük akár a jellegvektorok kinyerésével.

A Python-hoz szabadon elérhetők a*Tesseract*[13] és a*Keras*[5] függvénykönyv- tárak, amelyeket előszeretettel használnak optikai karakterfelismerési problémák meg- oldására. A gépi tanulás tanító- és tesztminták meglétét feltételezi. Mivel hosszabb terjedelműPDF dokumentumokat igen könnyűtalálni, ismert szövegből is egyszerűen létre lehetőket hozni, így a tanító és tesztminták előállítása ezek segítségével megold- ható.

A későbbiekben láthatjuk majd, hogy milyen módon és milyen eredményességgel lehet alkalmazni ezeket az aktuális problémára.

1. **fejezet**

**A dokumentum strukturális elemzése**

A strukturális elemzés elsőlépése a beolvasott, PDF formátumú dokumentumok PNG formátumú képekké való konverziója. A PDF formátum megválasztását az indokolta, hogy az tekinthetőa leginkább elterjedt, és a legtöbb esetben átvihetőformátumnak. Az átvihetőségből egyúttal az is adódik, hogy a dokumentumban a szövegek elrendezése karakter szintjén kötött lehet, a megjelenés ugyan egységes, viszont például az egybe- függőszövegrészek kijelölése, kimásolása már problémát jelenthet. A PNG formátum esetében szintén az elterjedtsége, szabványos kialakítása volt az egyik főszempont, de e mellett lényeges előny, hogy tömörített és veszteségmentes módon képes tárolni a képpontok intenzitásait.

## A PDF dokumentum beolvasása

A dokumentumok képpé konvertálásához apdf2imagefüggvénykönyvtárt használtam [8]. Ez paraméterként egyszerűbb esetben a PDF dokumentum elérési útvonalát és a kép méretét várja. (A méret megadásánál a képarány megtartása mellett opcionális lehet a kép magassága vagy szélessége.)

A beolvasást követően egy*Pillow*[3] függvénykönyvtár formátumának megfelelő képobjektum jön létre. A*Pillow*manapság már nem tekinthetőkorszerűeszköznek, ezért a beolvasott képeket*NumPy*[12] mátrixokká konvertálom. A beolvasás és az

átalakítás atest2.pdfesetében az alábbi módon valósítható meg.

☛ ✟

import cv2

import numpy as np

from pdf2image import convert\_from\_path

pil\_images = convert\_from\_path(’samples/ test2pdf.pdf’, size=(2500 , None)) images = [

cv2. cvtColor(np.array( pil\_image), cv2 . COLOR\_BGR2GRAY)

for pil\_image in pil\_images

]

✡

✠

A listára és aforciklusra itt azért van szükség, mert a PDF több oldalt is tartalmaz (és nyilván általában tartalmazhat).

AcvtColoregy*OpenCV*könyvtárhoz tartozó függvény, amelyik segítségével a kép szürkeárnyalatos képpé konvertálását végzem. A strukturális elemzés alapvetően olyan

* 1. *Margók becslése*

kontrasztos képekre készült, amelyeknél a színek nem jelentenek lényegi információt a feldolgozás szempontjából.

## Margók becslése

Amint megvannak az oldalak, majd szürkeárnyalatos képpé alakításuk, a margók le- vágása következik. Az elemzés a nagyobb egységektől a kisebbek felé történik, vagyis ezt a paragrafusokra, sorokra, szavakra majd azokon belül a karakterekre vonatkozó elemzés követi majd.

Ahhoz, hogy meg tudjam állapítani, hogy hol kezdődik a szöveg, és hol található összefüggőfehér rész (margó, sorköz stb.), intenzitást kellett számolnom a kép minden sorára és oszlopára (255: fehér, 0: fekete). Ehhez átlagoltam a pixelek intenzitásának értékeit. Ezzel eredményül kaptam egy*x*és egy*y*tengely menti tömböt, amelyek

az oszlopok és a sorok menti intenzitás átlagokat tartalmazzák. A számításNumPy segítségével egyszerűen elvégezhető. Tegyük fel, hogy a vizsgált kép azimagenevű változóban található. Ekkor a sorokhoz és oszlopokhoz tartozó átlagok (proﬁlok) a

következőformában számolhatók.

☛ ✟

row\_profile = np.mean(image, axis=1)

column\_profile = np.mean(image, axis=0)

✡

✠

Ezt a két intenzitást a 3.1. ábrával szemlélteti a program.

Ezt a két tengely menti intenzitás tömböt felhasználva már meg lehet állapítani, melyik pixelnél ér véget a margó és hol kezdődik a szöveg. Egy egyszerűciklussal meg- vizsgálom a tömb elejétől indulva hogy meddig egyenlőek az intenzitások a 255 értékkel, majd lementem azt az indexet ahol megáll a programom. Ezt megismétlem a tömbön visszafelé haladva is, így lesz meg az alsó-felső, jobb-bal margó végpontja. Ezeket fel- használva a programom már tudja kezelni a paragrafusokat tartalmazó képterületet.

A változás helyének detektálásátfind\_first\_changeés afind\_last\_changefügg-

vények végzik. Ezek a következőképpen kerültek megvalósításra.

☛ ✟

def find\_first\_change(values):

*"""*

*Find the index of the f i rst changed value in the values .*

*:param values : an iterable array of comparable objects*

*"""*

*: return: the i index where values [ i−1] != values [ i ]*

i = 1

while i < len(values):

return i i += 1

raise ValueError(’All values are the same!’)

if values[i *−* 1] != values[i]:

✡

☛

✠

✟

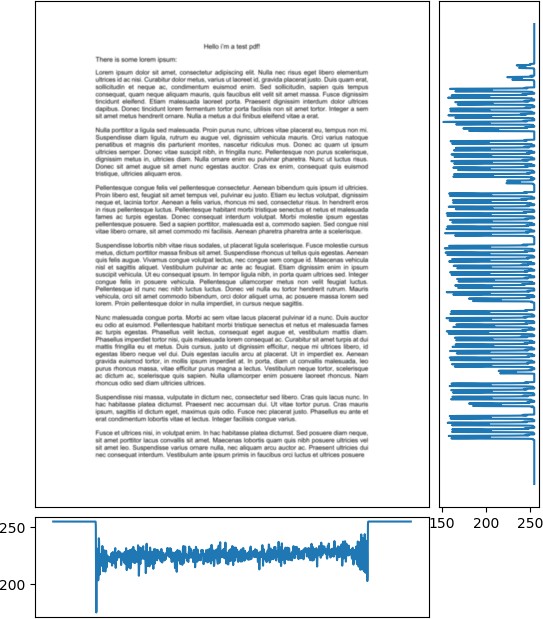
def find\_last\_change(values):

*"""*

*Find the index of the last changed value in the values .*

*:param values : an iterable array of comparable objects*

*: return: the i index where values [ i ] != values [ i + 1]*



* 1. ábra. A szegélyekre aggregált intenzitás értékek

✠

*"""*

i = len(values) *−* 2

while i >= 0:

if values[i] != values[i + 1]:

return i

raise ValueError(’All values are the same!’)

i *−*= 1

✡

A képek kiemelt fontosságú részét*Region of Interest*(röviden ROI) néven szokta emlegetni a szakirodalom. A programomban ezen területek adatainak kezeléséhez egy Regionosztályt deﬁniáltam, amely*(sor, oszlop, sorok száma, oszlopok száma)*négyessel jellemzi a téglalap alakú területet.

Ezek segítségével a margók a következőformában számolhatók.

☛ ✟

✠

def calc\_margins(image):

*"""*

*Calculate the margins of the image .*

*:param image : the NumPy array of page intensity image*

*: return: a Region instance """*

row\_profile = np.mean(image, axis=1)

column\_profile = np.mean(image, axis=0) row = find\_first\_change( row\_profile)

column = find\_first\_change( column\_profile)

n\_rows = find\_last\_change( row\_profile) *−* row

margins = Region(row , column , n\_rows , n\_columns)

return margins

n\_columns = find\_last\_change( column\_profile) *−* column

✡

Az ilyen formában becsült margókat amatplotlibfüggvénykönyvtár segítségével visz- sza is lehet rajzolni a képre.

## Paragrafusokra bontás

A paragrafusokra bontásnál feltételezés az, hogy a dokumentumnak van egy adott hát- térszíne. Ez ugyanaz az érték, amely a kép széleinél megjelenik (vagyis a margók kere- séséhez is felhasználásra került). Nyilvánvalóan ez a dokumentumok jelentős részében fehér, vagyis szürkeárnyalatos skálán a 255 intenzitásértéknek felel meg.

A sorokra bontásnál azt a tulajdonságot használhatjuk fel, hogy a soronként vett átlagintenzitások kisebbek (az áltagos szín sötétebb), mint a sorok között (3.2. ábra).



* 1. ábra. A jellemzőátlagintenzitások a soronként vett proﬁlban

A paramgrafusok elkülönítésénél az a cél, hogy a nem háttérszínhez tartozó szaka- szok ki legyenek gyűjtve. A Python kód ez esetben is könnyen áttekinthető, annak a működését a következőkódrészben láthatjuk.

☛ ✟

def find\_segments(values , background\_color):

*"""*

*:param values : intensity values*

*:param background\_color: the background color which should be skipped*

*: return: l i s t of segments as [ start , end) tuples of indices """*

segments = [] start = None end = None

for i, value in enumerate(values):

*Find the segments with non−background colors in the iterable .*

✠

if value != background\_color:

if start is None: start = i

elif start is not None: end = i

segments.append(( start, end)) start = None

return segments

✡

Jelenleg feltételezzük, hogy olyan dokumentumbról van szó, amelyben a sorok és a paragrafusok közötti távolságok különböznek. Ennek vizsgálatához gyűjtsük ki a pa- ragrafusok között lévőtávolságokat, majd vizsgáljuk meg azok eloszlását! A gyakori- sághisztogram a 3.3. ábrán látható.



* 1. ábra. A sorok és paragrafusok közötti távolságok eloszlása.

Jól látható, hogy tartomány közepén nincsenek értékek, így például egy 30 kép- pontnyi küszöbérték megfelelőaz elkülönítéshez.

A szegmensek (mint nem háttér színűösszetartozó részek) elkülönítésénél balról zárt, jobbról nyitott intervallumok szerepeltek. Ezeket egy külön függvény segítségével érdemes egymáshoz kapcsolni, abban az esetben, hogy ha a távolságuk nem haladja

meg az adott minimális távolságot. Az alábbi kódrészlet ezt a feladatot látja el.

☛ ✟

def join\_segments(segments , min\_spacing):

*"""*

*Join the segments which are closer to each others than the minimal spacing.*

*:param segments: l i s t of segments as [ start , end) intervals*

*:param min\_spacing: the minimal spacing between the joined segments*

*: return: l i s t of segments in the same format as the input """*

✠

joined\_segments = [ segments[0]]

for segment in segments[1:]:

if ( segment[0] *−* joined\_segments[*−*1][1]) < min\_spacing:

else:

joined\_segments[*−*1] = ( joined\_segments[*−*1][0], segment[1])

joined\_segments.append( segment)

return joined\_segments

✡

Az eredmény megjelenítéséhez amatplotlibpéldául az alábbi formában biztosít esz- közöket.

☛ ✟

margins = calc\_margins(image)

row\_profile = np.mean(image, axis=1) background\_color = 255

segments = find\_segments(row\_profile , background\_color) fig, ax = plt. subplots( figsize=(10 , 20))

plt.imshow(image, cmap=’gray’)

for segment in joined\_segments:

rectangle = plt. Rectangle(

( margins.column , segment[0]), margins.n\_columns , n\_rows , facecolor=’red’, alpha=0.1)

ax. add\_patch( rectangle) plt.show()

✡

n\_rows = segment[1] *−* segment[0]

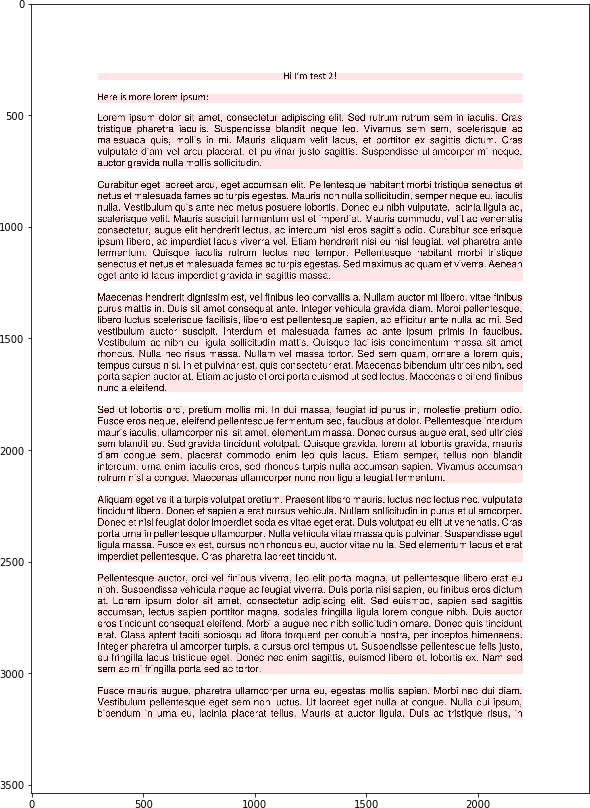
✠ Ez az eredeti képen kijelöli a kigyűjtött paragrafusokat (pontosabban azokat a részeket, amelyek a jelenlegi változat szerint a távolság alapján paragrafusként detektálhatók).

Amennyiben a szövegen kívül más nem szerepel az adott oldalon, a sorokra bontás-

hoz nem szükséges, hogy az adott oldalt felbontsuk paragrafusokra, az eredeti képből könnyedén megkaphatjuk a sorokat. Viszont, ha nem csak szöveges részeket tartalmazó dokumentummal van dolgunk, abban az esetben a sorokra bontás előtt szükséges meg- vizsgálni, hogy az oldal mely részei tartalmaznak szöveget, és mely részei tartalmaznak egyéb elemeket (például képeket vagy táblázatokat). Ezt a problémát majd a dolgozat későbbi részében fejtem ki bővebben.

## Szavakra bontás

A szavakra bontásnál szükségem van az eddig levágott sorokra, így egy*for*cikluson belül egyesével beolvasomőket, majd elsőlépésként kigyűjtöm a szavak koordinátáit. Itt már nem az*y*, hanem az*x*tengely menti intenzitást használom fel, és az előző algoritmusokhoz hasonlóan a világos, 240 feletti intenzitást keresem, és amint találok belőle egymás mellett legalább 5 darabot, akkor mentem le a koordinátát. Mivel a világos részek érzékelésével keresem a szavakat, így a legelsőszó elején is kell hogy legyen valamennyi világos terület hogy ne hagyja ki az algoritmus a vizsgálat során. Emiatt nem vágom le a soroknak a bal részéről a margót, mivel így biztosan megtalálja az összes szót. A másik megoldás az lehetne, hogy automatikusan lementem a 0 indexet, mint kezdőindex, ezzel is biztosítva hogy a kezdeti szó koordinátái is meglegyenek, de úgy véltem hogy a margó elhagyása egy logikusabb lépés.



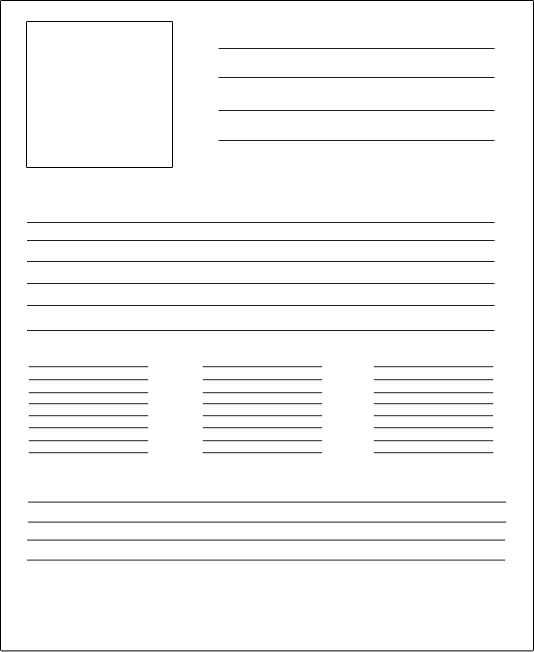
* 1. ábra. A program által megtalált és kijelölt paragrafusok.

## Karakter szintűelemzés

A szavak betűkre bontása már egy érdekesebb témakör. Ennél az algoritmusnál már nem a világos, hanem épp hogy a sötét részeket kerestem, és ha már 1 pixelnek megfelelő sötét részt is találtam már mentettem az adott indexet. Ez a legtöbb esetben szépen működött, és megkaptam egyesével a betűket. Viszont a ligatúrák esetében a betűk egymásba lógnak, ezért az algoritmusom nem vágta szétőket, hanem egybe mentette le. Ilyen esetek voltak például az*r*és az*f*vagy*t*találkozása, vagy például a dupla*t* vagy*f*betűk.

## Bonyolultabb dokumentumok

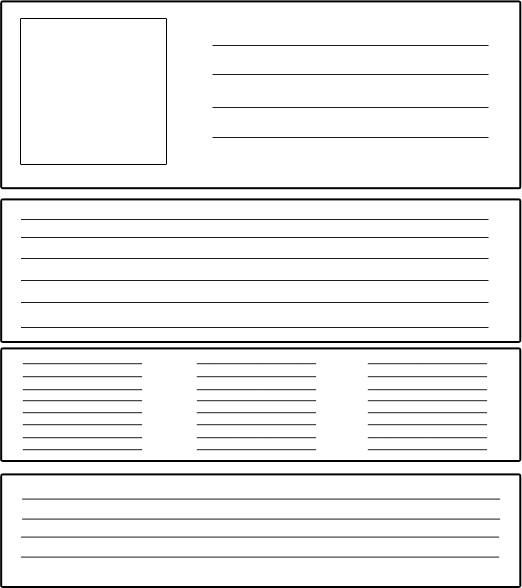
Az eddigiekben azt feltételeztük hogy a dokumentum amelyet vizsgálunk, az szövegen kívül mást nem tartalmaz, és az adott szöveg is folytonos, egyetlen hasáb. Amennyiben egy bonyolultabb szerkezetűPDF dokumentumot kell feldolgoznunk (például amely sematikus vázlata a 3.5. ábrán látható), abban az esetben az eddigi algoritmusok a paragrafusokra bontásig működnek, ám az esetlegesen előforduló képeket és a tábláza- tokat nem kezeljük külön, abban az esetben a sorokra bontásnál a folyamat megszakad. Annak érdekében, hogy ezt elkerüljük, a paragrafusokra bontásnál egyéb vizsgálatokra van szükség.



* 1. ábra. Nem csak egyszerűszöveget tartalmazó dokumentum sematikus vázlata A paragrafusra bontás során a vizsgálat az*y*tengely mentén történik. A képek

amelyeket ez úton megkapunk, vízszintesen bontják részekre a dokumentumot, tehát ha több hasábunk, vagy netalántán egy képünk van mellette valamennyi szöveggel, azok mind egy egységet alkotnak. A feladatunk hogy ezeket az egységeket szétbontsuk, és megismerjük hogy az adott részegységek szöveget tartalmaznak vagy valami mást. (A felbontás elsőnagyobb lépésének eredményét a 3.6. ábrán láthatjuk.)

A további szétbontás hasonlóan működik, mint az eddigi algoritmusok. Betöltjük



* 1. ábra. Az egységekre bontás elsőszakaszának eredménye

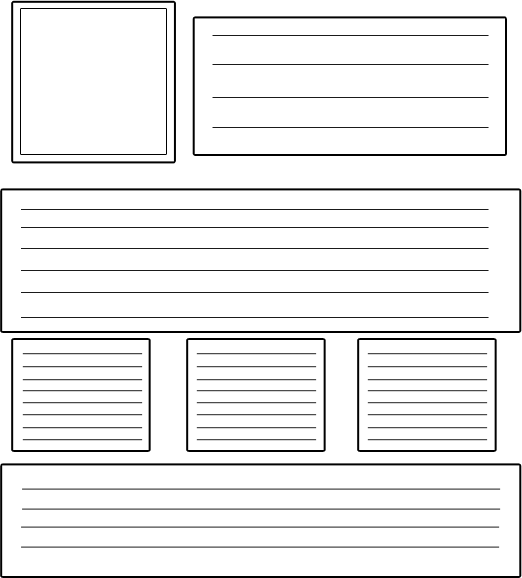
magát a képet, végrehajtunk rajta egy intenzitás vizsgálatot méghozzá az*x*tengely mentén, majd ezt megvizsgálva kimentjük a fehér részek koordinátáit. (Az eredmény szemléltetése a 3.7. ábrán látható.)

Abban az esetben, ha az algoritmus nem talál koordinátát, akkor visszatér egy hamis értékkel, és a teljes képpel dolgozik tovább.

Amennyiben talál, akkor következik a részegységekre bontás. Mivel a paragrafusok- nál a bal és jobb szélsőmargó már nem kerülﬁgyelembevételre a képen, így nincsen kezdeti fehér rész amelyből az algoritmus kiindulhatna. Ezért a koordinátákat tartal- mazó lista kezdő, és végpontja a margókon belüli rész kell legyen. Ezt a két értéket a kivágáshoz használt (crop) metódus előtt hozzáadom a listához.

Amennyiben a kép vagy szöveg tartalmaz behúzást, abban az esetben az algoritmus megtalálja a jobb/bal oldali fehér részt, így a kezdeti és végpont hozzáadása már nem szükséges mivel a lista már tartalmazza azokat. Egy egyszerűelágazással meg tudom állapítani, hogy szükséges-e a hozzáadás vagy sem. Ezt a megoldást a későbbiekben bevezettem a szavakra bontásnál is, hiszen így nem kell üres területeket kihagynom a sorokra bontás során.

Eredményül a program visszaadja a dokumentumot egységekre bontva. Ezek után



* 1. ábra. Az egységekre bontás második szakaszának eredménye

már csak azt kell megvizsgálni, hogy az adott egység kép vagy szöveg, és ennek megfe- lelően menteni.

## Képek és szövegek megkülönböztetése

A vizsgálathoz azt a tényt használtam fel, hogy ha a szöveg akár csak egy sorból is áll, alatta és felette mindenképp található világos intenzitás (nem biztos hogy teljesen fehér, hiszen a szöveg vonala alá érőbetűknek, mint például a*p*, a szára beletartozik az adott intenzitásba így az már nem csak 255-ös intenzitásértékeket tartalmaz). Az algoritmus világos intenzitást keres elsőnek. Amint talál legalább 2 egységnyi világos területet (240 feletti intenzitás), abban az esetben az adott ponttól elindul, és megnézi hogy talál-e egybefüggősötét területet (240 alatti intenzitás). Amennyiben igen, megvizsgálja, hogy a sötét intenzitású terület után talál egy újabb fehér részt. Amennyiben erre is az a válasz hogy igen, az algoritmus arra a következtetésre jut, hogy az adott egység szöveget tartalmaz, így azt a paragrafusokhoz menti, egyébként pedig a képekhez. Természetesen az algoritmus elég kezdetleges, így könnyen át lehet verni egy világos hátterű, középen sötét árnyalatú objektumot tartalmazó képpel (lásd 3.8. ábra).



* 1. ábra. Az algoritmus ezt a képet szövegként érzékeli

Az eddigi feldolgozás során a dokumentumot alulról-felfelé bontottuk részekre, vi- szont ahhoz, hogy végeredményként értelmes szöveget kapjunk, elengedhetetlen hogy a képek az oldalon való előfordulás szerinti sorrendben legyenek lementve.

Mivel az intenzitás vizsgálat a lap aljáról indul, így a lista pont fordított sorrendben tartalmazza a koordinátákat. Eddig acropmetódusban a*for*ciklusom a lista elejéről, most viszont a lista végéről indul el, így a lap tetején lévőparagrafus lesz lementve elsőnek, nem pedig a lap alján levő. A paragrafusokon kívül a sorok crop metódusát kellett még módosítanom, a szavakét, és a karakterekét nem, mivel azokat az*x*tengely mentén vágom, így azok eleve megfelelősorrenbe kerültek mentésre.

Ezzel a dokumentumom fel van bontva a legapróbb egységre (karakter) és megfe- lelősorrendben követik egymást az egységek, így következhet a karakterek felismerése *Optical Character Recognition*(röviden OCR) segítségével.

## Karakterek felismerése

A Python egyik leginkább elterjedt OCR eszközét, a*Python-tesseract*-ot választottam [2]. Elsőlépésben a karakter levágásnál átadtam apytesseract image\_to\_string nevűmetódusának a képet, majd a válaszként kapott sztringet hozzáfűztem egy már meglévősztringemhez. Amint végig ért a metódus minden soron, egy külsőállományba lementettem a megkapott szöveget. Ezen elsőpróbálkozásnál sajnos azt tapasztaltam, hogy az OCR egyetlen karaktert sem ismert fel.

Ebből arra következtettem, hogy a felbontása az adott képnek túl alacsony lehet, így megpróbálkoztam egy nagyobb felbontású képpel, de sajnos úgy sem jártam sikerrel.

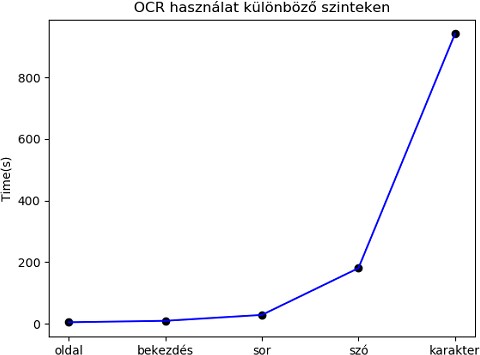
Következőlépésként egy szintet visszább léptem, és most nem a karaktereket, ha- nem a szavakat adtam át a metódusnak. Itt már sokkal kielégítőbb eredményt kaptam. A szavakat nagyjából 80%-os pontossággal beazonosította, viszont itt is akadtak prob- lémák bőven. Az egy és két betűs szavakat/kötőszavakat (például*az*,*és*,*a*,*s*) és a

számokat nem ismerte fel a metódus (vagy csak nagyon ritkán), és üres szövegeket kaptam helyettük. Ezen kívül elég sok ékezet lemaradt, és gyakran keverte a betűket, például az*I*és az*l*esetében.

Ezen tapasztalataim után nekiláttam az OCR paramétereinek a vizsgálatához, és néhány beállítással sikerült elérnem hogy az OCR működjön karakter szinten is. Ehhez a paraméterek változtatása mellett biztosítottam, hogy az általam kivágott betűk körül mindig legyen egy fehér színű, 10 képpont méretűkeret, ezzel segítve a programot a sikeresebb felismerésben. A betűtípustól függően akadnak itt is kisebb-nagyobb hibák. A word alapértelmezett*Calibri*betűtípusában szereplő*l*betűaz tulajdonképpen egy vonal, így azt a program a | (AltGr + W) karakterként ismeri fel. Emellett gyakran összetéveszti a*c*betűt az*e*betűvel, a vesszőt pedig a ponttal.

Ezek természetesen az eddigi hibákhoz képest sokkal biztatóbb eredmények, és így a kapott szöveg már olvasható.

A különbözőszöveg szerkezeti egységeken való OCR használat kapcsán elvégeztem néhány tesztet. Megvizsgáltam, hogy az adott szinteken (oldal, bekezdés, sor, szó, ka- rakter) milyen időkülönbséggel fut le a karakterfelismerő. Minden esetben a 3.1. ábra PDF dokumentumát használtam bemenetként. Ez egy egy oldalas dokumentum amely 672 szót és 4520 karaktert tartalmaz. Az eredmény a 3.9. ábrán lévőgraﬁkonon látható.



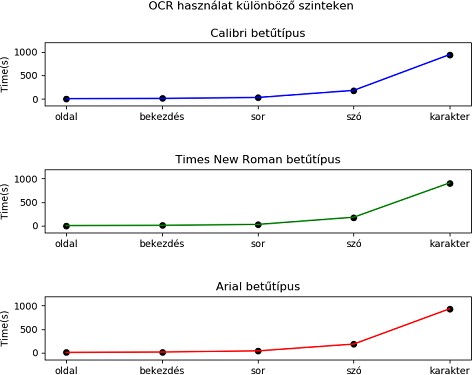
* 1. ábra. Az OCR futásának ideje másodpercben különbözőszöveg szerkezeti egysé- geken

Amikor az OCR egy nagyobb egységet kapott feldolgozásra, akár egybe egy oldal- nyi szöveget, egy bekezdést vagy csak egy sort, csekély, mindössze 5-20 másodperces különbséggel már vissza is adta az eredményt. Viszont amikor már kisebb egységekkel volt dolga, sokkal több időbe telt neki mire megtalálta a legmegfelelőbb karakter. A

szavaknál már olyan átlag 180 másodpercet, míg a karaktereknél ez olyan átlag 930 másodpercet vett igénybe.

Ez a különbség főként abból adódhat, hogy bizonyos karaktereknél szüksége van az OCR-nek viszonyítási alapra, önmagában nem tudja biztosan megmondani hogy milyen karaktert kapott. Tegyük fel hogy beadunk neki egy kis*v*betűt. Ha nincs mellette viszonyítási alap, tegyük fel egy*a*betű, akkor a*v*az lehet kicsi és nagy betűis. Viszont ha már egybe látja az OCR a kettőbetűt,*av*akkor már megtudja állapítani hogy a karakter az kicsi. Tapasztalatom szerint ez a szavak szintjén sem kielégítő, viszont a sorok szintjén már elég nagy pontossággal megtudja különböztetni a kis- és nagybetűket.

A tesztet elvégeztem különbözőbetűtípusok esetén (*Calibri*,*Arial*,*Times New Ro- man*). Arra a kérdésre kerestem a választ, hogy megnehezítik-e a feldolgozást bizonyos betűtípusok vagy sem. Mint az a 3.10. ábrán látható, nem nagy az eltérés a különböző betűtípusok feldolgozása között. A három példa közül az*Arial*volt aminek a kiértéke- lése több időt vett igénybe minden szöveg szerkezeti egységen, de ez is nagyon csekély eltérés, így az ábrán nem is szembetűnő.



* 1. ábra. Az OCR futásának ideje különbözőbetűtípusok esetén

Azon kívül, hogy a feldolgozása a nagyobb szöveg egységeknek gyorsabb, a tesztek során az is kiderült, hogy pontosabb is. Ez elsősorban a már fentebb említett karak- terek egymáshoz való viszonyításának köszönhető, hiszen itt bőven akad betűamihez viszonyíthatjuk a többit. Azért a karakterek szintjén történőOCR használatát sem kell kizárni a lehetőségek közül, mivel annál is minden esetben 75% feletti volt az eredmény pontossága. Itt a legtöbb hibát a fentebb említetten kívül az okozza, hogy különböző betűtípusokban találhatók egymásra nagyon hasonlító betűk.

## Küszöbértékek meghatározása

Eddig tapasztalati küszöbértéket használtam a sorok és a paragrafusok elkülönítésére. Mivel minden dokumentumnak más térközei, betűméretei lehetnek így ez nem megfelelő minden PDF-re. Ezért mielőtt megkezdeném a dokumentum feldolgozását, meghatá- rozom az adott értékeket azokat használom fel a számítások során.

Az*y*tengely menti intenzitást vizsgálom, méghozzá azon belül is az összefüggővi-

lágos (háttér) intenzitásokat keresem. Amint más intenzitást találok, abban az esetben az eddigi összefüggőrész nagyságát lementem, majd a számolást újra kezdve haladok tovább. Eredményül egy listát kapok ami tartalmazza a térközök nagyságát.

A 3.1. ábrán szereplődokumentum ilyen módon kiszámított térközeinek eloszlása a 3.11 ábrán látható.



* 1. ábra. A 3.1. ábrán szereplődokumentum térközeinek eloszlása

Az ábrából jól kivehetőhogy 3 főcsomópontja van a térközöknek. A bal szélső a sorköz, a középsőa bekezdés majd az utolsó a margó. Ezen csomópontok kezdeti

értéke között mindig legalább 10 lesz a különbség, ezt használom fel a küszöbértékek meghatározásához. Miután megkaptam az adott nagyságokat, acollectionskönyvtár Countermetódusával rendezemőket, így megkapom, hogy milyen nagyságú intenzi- tásból mennyi van összesen a dokumentumban. Ezeket növekvősorrendbe rendezem. Elindítok egy ciklust, és a 3-nál nagyobb elemeket vizsgálom, mivel a 3 és a kisebb nagyságú térköz nagy valószínűséggel csak zaj. Az elsőelemet lementem küszöbérték- nek, majd tőle elindulva egy új ciklussal addig vizsgálom az térköz nagyságokat amíg nem lesz nagyobb a kettőkülönbsége mint 10. Amint ez igaz, az eddigi küszöbértéket lementem egy listába, és az utoljára vizsgált érték lesz az új küszöböm.

Végeredményül kapok egy listát, ami a 3.11. ábrán látható csomópontok kezdeti értékét tartalmazza. Ennek a listának a 0. eleme lesz a sorköz, az 1. eleme a térköz és a 3-4. eleme pedig az alsó-felsőmargó.

## Képfelismerés neurális háló segítségével

A programom főrésze maga a paragrafusokra bontás, és az ilyenkor történőképfelis- merés. Ekkor dönti el a program hogy érdemes-e az adott résszel foglalkozni, vagy sem. Amennyiben egy szöveget tartalmazó bekezdés a képekhez kerül, a rajta levőszöveg nem értékelődik ki. Amennyiben a kép a szövegeknél marad a program megpróbálja apróbb részekre bontani, és sok felesleges képpel foglalkozik. Ami viszont fontosabb, hogy mind a két eset azt eredményezi, hogy a kapott szöveg minősége romlik. Vagy kimarad belőle több bekezdés, vagy a képek helyére kérdőjel, vagy egyéb oda nem illő karakter kerül.

Nyilvánvalóvá vált hogy ezen a téren fejlődnie kell a programnak, ezért elkezdtem foglalkozni a neurális hálókkal, méghozzá a Python-ban megtalálható*Keras*nevezetű neurális háló API-val [6].

A*Keras*főépítőeleme a model, ami lehetőséget ad a különbözőrétegek kezelésére.

Elsőlépésben konﬁgurálni kell a modellünket, megadni hogy hány és milyen rétegeket szeretnénk használni, majd jöhet a betanítás.

Egy szekvenciális modellt használtam, 2 konvolúciós és 2 pooling réteggel. A be- tanítás során 150 képet adtam be a modellemnek, egyenként32 32-es méretűeket.

*×*

Ahhoz, hogy pontosabb eredményt érjünk el, sokkal több tanuló képre lenne szükség. A betanítás eredménye a 3.12. és a 3.13. ábrákon látható.

Jól látható, hogy a futások számával növekszik a pontosság és csökken a veszteség. A modell a mintákat 85% feletti pontossággal tudta beazonosítani a 10. futás során. Ez egy egészen elfogadható arány, így megkezdhetjük a program tesztelését.

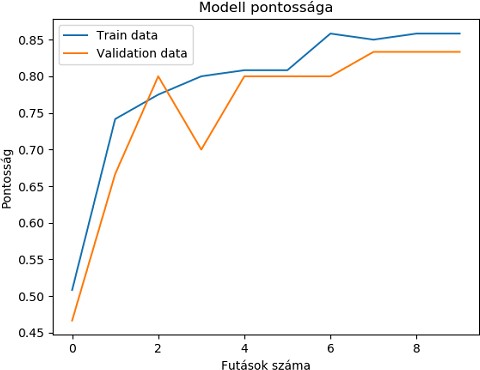
## Program tesztelése

A programot átírtam úgy, hogy egymás után olvassa be a PDF-eket és mindegyiknek az eredményeit egy külön mappába mentse. Összesen 50 dokumentumot gyűjtöttem, amelyeknek különbözőa méretük, struktúrájuk és a betűtípusuk. A tesztelés arra irá- nyult, hogy kiderüljön, hogy a program mennyire stabil, milyen pontossággal nyeri ki az adatokat az adott PDF-ből, és hogy rávilágítson a program hiányosságaira. A gép paraméterei amin teszteltem: 8GB RAM, Intel(R) Core(TM) I5-9400F 2.90 GHz processzor, 64 bites Windows operációs rendszer.

### Futási időbecslése

A program úgy működik, hogy beolvassa az összes oldalt a memóriába, és utána egyen- ként dolgozza felőket. Ez nagyban lassítja a több oldalas dokumentumok feldolgozását és a magas oldalszámú PDF-két pedig ellehetetleníti. Tapasztalatom szerint olyan 150 oldalas dokumentum az, amit még a program az adott körülmények között kiértékel. A futási sebességek a 3.14. ábrán láthatók.

Mint azt az ábra mutatja, a futási időnem csak az oldalszámtól, hanem a PDF bonyolultságától is függ. Ezért lehetséges az, hogy egy 90 oldalas dokumentumot ha-



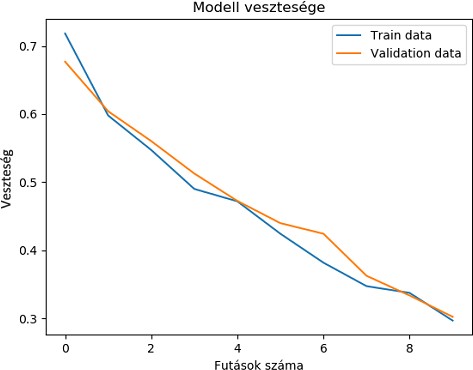
* 1. ábra. A modell pontossága

marabb feldolgozott a program mint egy 81 oldalasat. Az átlagos futási idő8,44 másod- perc/oldal volt a tesztelés során, az oldalszámok szerinti átlagot a piros vonal mutatja.

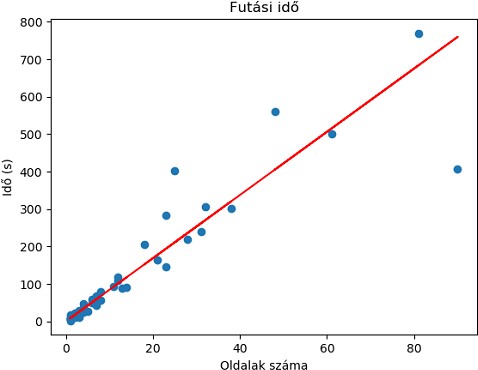
### A paragrafusok feldolgozásának a helyessége és a szöveg helyességének a viszonya

Mint már említettem, a paragrafusok feldolgozására épül minden más mozzanat. Mégis amit a legjobban befolyásol, az a kimeneti szövegnek a helyessége. A tesztelt dokumen- tumoknál a paragrafusok kivágásának a helyessége 83.99%, míg a szövegnek 76.63% volt. A 3.15. ábra a paragrafusok és a szöveg helyességét hasonlítja össze minden egyes dokumentumra.

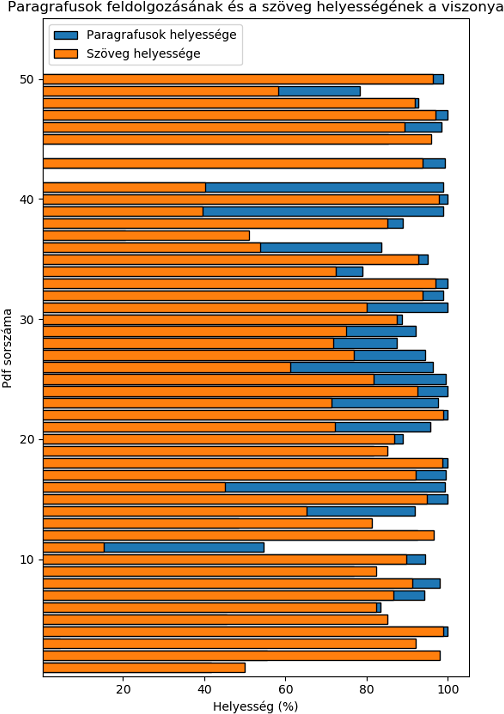
Az ábrán jól kivehető, hogy szinte minden dokumentumra a paragrafusok feldol- gozása 60%, míg a szöveg helyessége 50% felett volt. Két dokumentum esetében mind a kettőérték nulla, mivel az adott PDF-ek csak képeket tartalmaztak, így azok jelen esetben nem relevánsak. Az esetek 79.17%-ban (38 esetben, a két képeket tartalmazó dokumentumokat nem számolva) ahogy romlott a paragrafusok helyességének aránya, úgy romlott vele a szöveg helyességének az aránya is. A legtöbb hiba a képek és pa- ragrafusok megkülönböztetése miatt adódott. Szinte minden esetben előfordult, hogy legalább 1 paragrafust a képekhez, vagy egy képet a paragrafusokhoz sorolt az algorit- mus.



* 1. ábra. A modell vesztesége



* 1. ábra. Futási időa dokumentum oldalainak száma szerint



* 1. ábra. A paragrafusok és a szöveg helyességének az összehasonlítása

1. **fejezet**

**Partícionálás és klaszterezés**

A dolgozat alapproblémájának vizsgálata során két alternatív módszer vetődött fel a kép strukturájának vizsgálatához. A fejezetben ezek áttekintésére, a hozzájuk felhasz- nált algoritmusok bemutatására és a kapott eredmények szemléltetésére kerül sor.

## Oldalak partícionálása

Apartitioning.ipynbfájlban található munkafüzet egy lehetséges felosztási mód- szert mutat be. Ez arra az észrevételre épül, hogy a dokumentumok képének elrendezése alapvetően ortogonális és közökkel tagolt.

A felosztási algoritmus egyik alapeleme az a függvény, amelyik képes kiszámítani, hogy szegmensek sorozatában hol található a legnagyobb köz. Az implementációja az

alábbi formában néz ki.

☛ ✟

def find\_max\_split\_position( segments):

*"""*

*Calculate the spacing between the segments.*

*:param segments: l i s t of segments as [ start , end) intervals*

*: return: tuple of split position with maximal spacing and the spacing i t* self

*"""*

spacing = [] max\_space = 0

max\_split\_position = None

for i in range(len( segments) *−* 1):

if space > max\_space:

max\_space = space

space = segments[i + 1][0] *−* segments[i][1]

max\_split\_position =

( segments[i][1] + segments[i + 1][0]) // 2

return max\_split\_position , max\_space

✡

✠ Tulajdonképpen egy egyszerűmaximum keresésről van szó, amelyik a pozíciót a leg- nagyobb köz közepére számítja. Ezt a függvényt így egyaránt lehet használni a sor- és oszlop proﬁlokra is.

Az előbbi függvény alapján már kiszámítható, hogy egy képet (képrészletet) mi- lyen tengely szerint és milyen pozícionál kell vágni a heurisztika alapján. Ez az alábbi

☛formában került implementálásra. ✟

def find\_split\_position(image, region):

*"""*

*Find the optimal position of the splitting .*

*:param image : a NumPy image*

*:param region: the considered part of the image*

*: return: tuple of axis (0 or 1) and position """*

roi = image[

region.row:region.row + region.n\_rows ,

region.column:region.column + region. n\_columns] row\_profile = np.mean(roi, axis=1)

segments = find\_segments(row\_profile , 255)

row\_split , row\_space = find\_max\_split\_position( segments)

if row\_split is not None:

row\_split += region.row

column\_profile = np.mean(roi, axis=0)

segments = find\_segments( column\_profile , 255)

column\_split , column\_space = find\_max\_split\_position( segments)

if column\_split is not None:

column\_split += region.column

if column\_split is None:

return 0, row\_split

if row\_split is None:

return 1, column\_split

if row\_space >= column\_space:

return 0, row\_split

else:

return 1, column\_split

✡

✠ Ahogy a dokumentációs kommentben látható, ez a sor szerinti felosztást 0-val, az oszlop szerinti felosztást 1-el jelöli.

A felosztáshoz elengedhetetlen, hogy a régiókat két részre lehessen osztani. Ennek az implementációja is következetesen adódik az előbbi függvény által számított adatok

☛segítségével. ✟

def split\_region(region , axis, position):

*"""*

*Split the region into two half by the given axis .*

*:param region: region of interest*

*:param axis : 0 or 1 for row and column*

*:param position : position of the splitting*

*: return: tuple of two region objects """*

if axis == 0:

upper\_region = Region(

region.row , region.column ,

)

position *−* region.row , region. n\_columns

✠

lower\_region = Region(

position , region.column ,

)

region.row + region.n\_rows *−* position , region. n\_columns

return upper\_region , lower\_region

else:

left\_region = Region(

region.row , region.column ,

)

region.n\_rows , position *−* region.column

right\_region = Region(

region.row , position ,

)

region.n\_rows , region.column + region. n\_columns *−* position

return left\_region , right\_region

✡

Mindezek szükségesek ahhoz, hogy az említett rekurzív felosztást meg lehessen csinálni. Mivel az alapműveletek már rendelkezésre állnak, így a rekurzív algoritmus aránylag rövid lesz.

☛ ✟

def build\_tree(image, region):

*"""*

*Building a binary tree by splitting the image horizontally or vertically .*

*:param image : the image which should be recursively partitioned*

*:param region: the considered part of the image*

*: return: a binary tree of regions """*

axis, position = find\_split\_position(image, region)

if position is not None:

region\_1 , region\_2 = split\_region(region , axis, position) left\_child = build\_tree(image, region\_1 )

right\_child = build\_tree(image, region\_2 )

return [left\_child , right\_child]

else:

return region

✡

✠ A dokumentum strukturális elemeit tartalmazó bináris fa tetszőleges mélységben egy- mást tartalmazó Pythonlistobjektumok formájában áll ennek az eredményeképpen rendelkezésre.

Most vizsgáljuk meg, hogy hogyan ellenőrízhetőa program megfelelőműködése.

Ehhez egy olyan függvény megírására volt szükség, amelyik a fa bejárása során kapott régiókat rekurzívan be tudja járni. Mivel a sok kis képobjektum kezelése számításigé- nyes lett volna, ezért az eredményként tekintett kép globális névtérbe került. Amennyi- ben az eredeti kép egyimagenevűváltozóban elérhetű, úgy a felosztás megjelenítése az alábbi formában oldható meg.

☛ ✟

result = cv2.merge(( image, image, image))

def draw\_regions(tree):

✠ Az egyik mintadokumentumra elvégezve a felosztás eredményének egy részletét a 4.1. ábrán láthatjuk.

global result

if isinstance(tree, list): draw\_regions(tree[0]) draw\_regions(tree[1])

else:

region = tree

x\_1 = region.column y\_1 = region.row

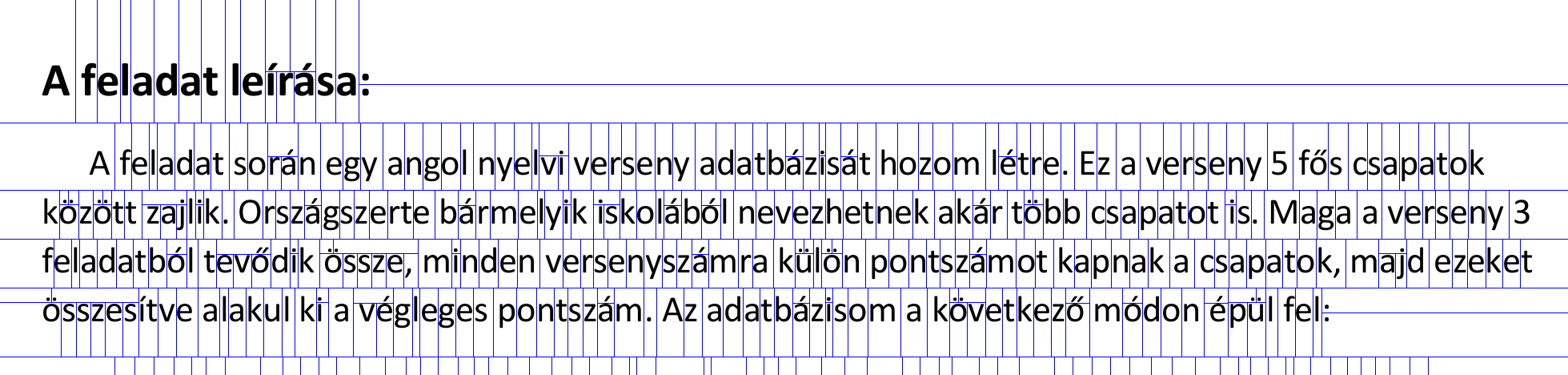
x\_2 = x\_1 + region. n\_columns y\_2 = y\_1 + region.n\_rows

result =

cv2 . rectangle(result , (x\_1 , y\_1), (x\_2 , y\_2 ), (255 , 0, 0))

draw\_regions(tree)

✡



* 1. ábra. A felosztással kapott eredmények egy részlete

## Összefüggőképterületek detektálása

A karakterek elkülönítése szempontjából problémát jelent a kerning. Ez arra szolgál, hogy az egymáshoz szebben, arányosabban illeszthetőkaraktereknél korrigálja az egyen- közűegymás után illesztést. Egy szélsőséges eseteként az*ﬂ*és*ﬀ*betűkből képzett ligatúrák szerepelnek.

A feldolgozás és karakterfelismerés szempontjából ezek különﬁgyelmet igényelnek. A probléma egy lehetséges megoldását az adja, hogy ha az egyenesekkel történőfelosz- tás helyett olyan algoritmust használunk, amely inkább az összefüggőképterületeket keresi meg, vagyis gyakorlatilag klaszterezi a képet. (A klaszterek ebben az értelmezés- ben többségében a karaktereket, vagy az azokhoz hasonlóan összefüggőképterületeket jelenti.)

Az összefüggőképpontok csoportjának (*blob*-oknak) a keresése feltételez valamilyen hasonlóságot a képpontokra nézve. Ez esetben ez két összetevőből adódik. Egyrészt küszöbölés segítségével meg kell állapítani, hogy a képpont világos vagy sötét-e. Mivel PDF-ből konvertált képről van szó, ezért a kép kontrasztos, és többségében csak az élsimításból adódó zajokat tartalmazza, így ez egyszerűen, akár 128-as küszöbértékkel végezhetőegy tipikus dokumentum esetén.

A másik összetevőa szomszédsági viszony megadása. Feltételezzük, hogy az algo- ritmus csak a sor- és oszlop szerinti szomszédságot vesziﬁgyelembe. Így egy egyszerű

szélességi bejárással meghatározhatók az összefüggőcsoportok. Maga az algoritmus tu- lajdonképpen egy színkitöltőalgoritmus (*ﬂoodﬁll*) egyszerűsített változata lesz így [14]. Ablob\_shapes.ipynbnevűmunkafüzetbenfind\_blobnéven került implementálásra.

Ez paraméterként az alábbi objektumokat várja.

* image: A feldolgozandó képNumPytömb formájában.

start\_point: A szélességi bejárás kezdőpontja (sor, oszlop) indexpár formájá- ban.

*•*

* threshold: A küszöböléshez használt érték.

is\_visited: Egy*NumPy*tömb, amely logikai értékeket tartalmaz, és azt tárolja, hogy mely képpontok kerültek már vizsgálatra.

*•*

Mivel Python-ban az objektumok referencia szerint adódnak át, így azis\_visited paraméter egyúttal kimeneti értékként is kezelendő. Maga a függvény elsődleges vissza- térési értéke a*blob*-hoz tartozó képpontoknak a koordinátái lesznek.

Az algoritmus, amelyik a képen lévőösszes összefüggőfoltot megkeresi az alábbi

formában néz ki.

☛ ✟

def find\_blobs(image):

*"""*

*Find the blobs on the image .*

*:param image : a two dimensional NumPy array*

*: return: l i s t of the collected blobs """*

blobs = []

threshold = 128

is\_visited = np.zeros(image.shape, dtype=int) n\_rows , n\_columns = image.shape

for i in range(n\_rows):

for j in range( n\_columns):

if is\_visited[i, j] == False and image[i, j] < threshold: blob = find\_blob(image, (i, j), threshold , is\_visited) blobs.append(blob)

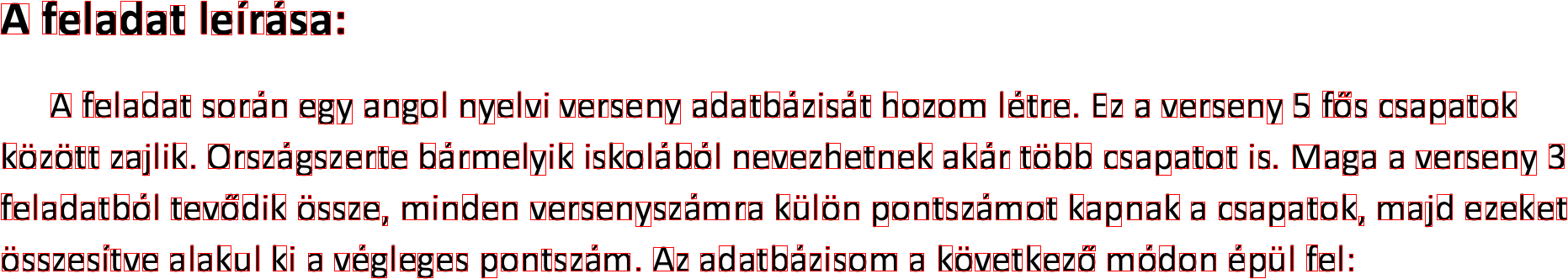
is\_visited[i, j] = True

return blobs

✡

✠

A kapott eredmények ellenőrzéséhez a munkafüzetben kirajzolásra kerültek a foltok befoglaló téglatestjei. Ezt egy mintakép részletére a 4.2. ábrán láthatjuk.



* 1. ábra. Az összefüggőfoltokhoz tartozó befoglaló téglatestek

A foltok pontjainak ismeretében már további elemzési lehetőségek is adódtak. A munkafüzetben elemzésre került foltokhoz tartozó képpontok számának, a befoglaló téglatestjeik szélességének és magasságának az eloszlásai.

Feltételezhető, hogy a foltok ismeretében nagyobb megbízhatósággal elvégezhető a karakterek felismerése. Ehhez arender\_blobfüggvény a foltot egy adott méretű

képterületre kirajzolja.

☛ ✟

def render\_blob(blob , image\_size):

*"""*

*Render the blob to a fixed size image .*

*:param blob : l i s t of pixel coordinates*

*:param image\_size: size of the rendered image*

*: return: a NumPy array with 0 and 1 values """*

image = np.zeros(image\_size , dtype=int) min\_row = min([i for i, \_ in blob])

min\_column = min([j for \_, j in blob])

for row , column in blob:

return image

image[row *−* min\_row , column *−* min\_column] = 1

✡

✠ A rögzített méret miatt a karakterek, mint képek, a környezőzajok nélkül összeha- sonlíthatóak. Az összehasonlításokhoz a vizsgálatokban egy egyszerűmátrixtávolság-

számítás került megvalósításra az alábbi formában.

☛ ✟

def compare\_blob(blob\_1 , blob\_2 ):

*"""*

*Calculate the distance of the blobs .*

*:param blob\_1 : l i s t of blob pixels*

*:param blob\_2 : l i s t of blob pixels*

*: return: distance of the blobs """*

image\_1 = render\_blob(blob\_1 , (50 , 50))

image\_2 = render\_blob(blob\_2 , (50 , 50))

return distance

distance = np.sum(np.sum(np.abs( image\_1 *−* image\_2 )))

✡

✠

Az50 50-es képméretet az eloszlások alapján egyszerűen lehetett becsülni.

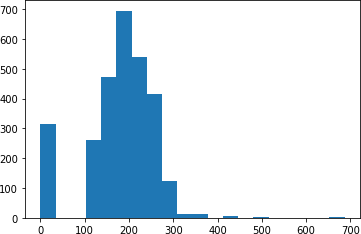
*×*

Ezzel a metrikával már osztályozni lehet a foltokat egy referencia folthoz viszonyított távolság alapján. Legyen ez a folt, most egy*a*karakterhez tartozó folt. A 4.3. ábrán az összes foltnak a referencia folttól való távolságainak a hisztogramja látható (20 intervallumra osztva a tartományt).

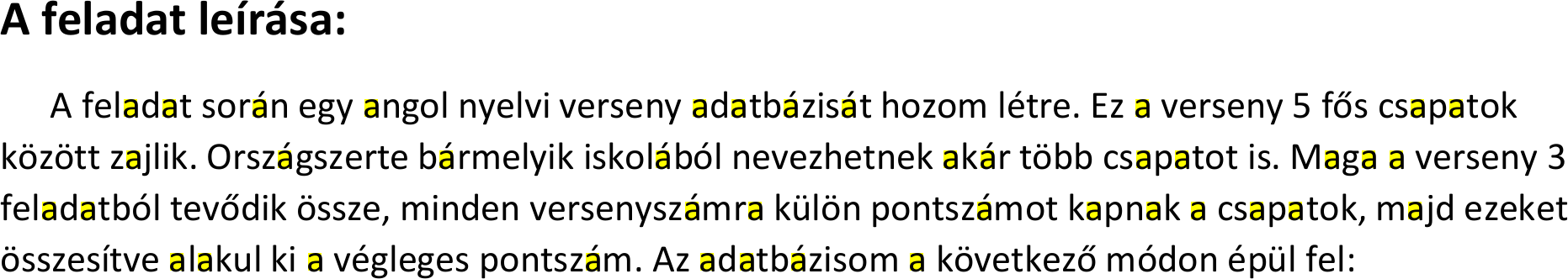
Az értékekből az olvasható le, hogy a[0*,*100]intervallumot a távolságokra nézve

gyakorlatilag bárhol ketté lehet vágni a közepe felé. Ezt válasszuk meg például 60-nak. Ekkor az ezekhez tartozó foltokat sárga színnel kijelölve a 4.4. ábrán láthatjuk.

Érdemes azt észrevenni, hogy a**feladat**szóban szereplő**a**karakter nem lett kijelöl- ve. Ez a miatt van, mert a küszöbérték aránylag alacsony értékre lett beállítva, illetve mert a távolságszámítás önmagában csak a közeli egyezés, a kisebb eltérésekﬁgyelmen kívül hagyására alkalmas.



* 1. ábra. Az*a*karakterhez tartozó referencia folttól való távolságok hisztogramja



* 1. ábra. A mintában sárgával kijelölt*a*karakterek

A módszer kimondottan előnyös több szempontból is. Egyrészt, nem feltételezi, hogy a karakterek egyáltalán sorokba lennének rendezve. Másrészt, a karakterek között lehet olyan értelemben átfedés, amely semmilyen felosztásra épülőbefoglaló téglatestes módszerrel nem lenne kezelhető.

1. **fejezet**

**A vizsgálatokhoz készített programok**

Ahogy a korábbi fejezetekben már említésre került, a dolgozat Python programozási nyelven készült. A vizsgálatokhoz függőségként elsősorban a*NumPy*,*OpenCV*illetve a*Matplotlib*függvénykönyvtárak jelentek meg.

A kutatás, algoritmus fejlesztés és paraméterkeresés esetében*Jupyter*munkafüze- tekbe kerültek a kifejlesztett módszerek. A karakterek felismerése egy külön, a struk- turális elemzéstől hellyel-közzel függetlenül kezelhetőproblémának bizonyult. Ahhoz a *Tesseract*és a*Keras*függvénykönyvtárak kerültek felhasználásra.

A következőszakaszokban röviden bemutatásra kerül, hogy a program milyen ré- szekből épül fel, azoknak melyek a főjellemzői és hogyan lehet használniőket.

## Képek vágása és további feldolgozása

Az elsőmegközelítés esetében az eredeti képeket külön PNG fájlokba konvertálta a program. Ez azért tünt előnyösnek, mert így az*OpenCV*könyvtárral való feldolgo- zásnál már nem kellett azzal foglalkozni, hogy a képek eredetileg PDF formátumú dokumentumból származtak.

Ezek, mint elsőpróbálkozások afirst\_trynevűPython modulba kerültek. Ez fogja össze az alábbi modulokat:

* + - crop\_methods: A képek kivágásáért, fájlba mentéséért felelős.

get\_methods: Azon függvényeknek a gyűjteménye, amelyek a képekből kinyerhe- tőadatok számításait tartalmazzák. A*get*itt arra utal, hogy az adott kép alapján kaphatjuk meg a segítségükkel a szükséges eredményeket.

*•*

manage\_directories: Mivel egy PDF dokumentum esetében több oldal, azon belül pedig elméletileg tetszőleges sok strukturális elemről lehet szó, ezért a modul az ahhoz szükséges jegyzékeket kezeli. A problémát úgy oldja meg, hogy ellenőrzi, hogy milyen jegyzékekre van szükség a különféle képek kezeléséhez, és hogy azok léteznek-e. Főműveletei így a jegyzékek létrehozása és törlése.

*•*

Ezen forráskódok elkészítése több szempontból is nagyon hasznosnak bizonyult. Egyrészt ezek segítségével történt az alapvető, proﬁlokra, abszolút intenzitásokra és azok küszöbölésére épülőmódszerek kidolgozása.

További előnye volt még, hogy így a kép egyes részeit az eredeti képtől függetlenül lehetett kezelni, elemezni.

*5.2. Jupyter munkafüzetek használata*

Acrop\_methodsmodulbancrop\_preﬁxszel szerepelnek azok a függvények, ame- lyek a vizsgált strukturális elemek szerinti elkülönítést végzik.

A modul szkript jelleggel készült, azért hogy az algoritmusok kipróbálása minél egyszerűbb legyen. A benne lévőelemek osztályba szervezhetők, viszont az aktuális felhasználási módja ezt nem indokolta. (Python esetében a modul, mint névvel hi- vatkozható függvénykönyvtár használható,classkulcsszóval történőosztálydeﬁníció nélkül.)

Az említett programrészek futásának eredményeit összesítve a 3. fejezetben láthat- juk.

A programrészek használatához azimport-okban szereplőpipcsomagok telepítése szükséges.

## Jupyter munkafüzetek használata

A módszerek kutatásának, a kapott eredmények elemzésének egy másik módját a Jupy- ter munkafüzetek használata jelenti [10]. Ezekből a dolgozathoz az alábbiak készültek el.

page\_structure.ipynb: A dokumentumok szerkezeti elemzésének lépéseit mu- tatja be.

*•*

partitioning.ipynb: A dokumentumon egy rekurzív felosztásos eljárást alkal- maz, amely a képet addig szeleteli, amíg mindegyik felosztással kapott téglalap alakú területen már csak egyetlen egy összefüggő(nem elvágható) képelem van. Az eredményeket a munkafüzet meg is jeleníti.

*•*

blob\_shapes: Egy*ﬂoodﬁll*algoritmust mutat be, amellyel az egymással szom- szédos sötét színűképpontokból kapott összefüggőalakzatokat tudja kigyűjteni a program. A munkafüzetben egy példa is szerepel, amely az*a*betűmegkeresésének egy módját is szemlélteti.

*•*

Ezek mellett még megtalálható egyRegionosztály, amely a kiemelt fontosságú képterületeket reprezentálja. Ez az, amelyet gyakran RoI-ként (*Region of Interest*) is emlegetnek.

Az említett eredmények megtekintéséhez a Jupyter munkafüzet celláit kell futtatni a szokásos módon. A megfelelőműködéséhez a*NumPy*,*OpenCV*,*Matplotlib*és*pdf2image* függvénykönyvtárakra van szükség.

1. **fejezet Összefoglalás**

A dolgozat a dokumentumok szerkezeti elemzésének néhány lehetőségét mutatta be a Python eszközkészletének felhasználásával. A feldolgozási lépések jellemzően a nagyobb elemektől a kisebbek felé történtek. Ez az irány természetesen adódott, és láthatóvá vált közben az is, hogy a kisebb elemek feldolgozása hasonlóképpen oldható meg, azonos részproblémákat vet fel, mint a nagyobbaké. A részek elkülönítéséhez az intenzitáspro- ﬁlként kapott vektorokkal lehetett dolgozni, amely így egyaránt megfelelőnek bizonyult a margók, sorok és karakterek elválasztása esetében is.

A felvetett problémára csak akkor lehetne tökéletes megoldást adni, hogy ha ponto- san deﬁniálva lenne, hogy milyen elemek és hogyan fordulhatnak előegy dokumentum- ban. Ez hellyel-közzel teljesül, mivel a PDF szabványosított, viszont olyan eseteket nem lehet kizárni, amikor például a dokumentum háttere egy kép, ami miatt a dokumentum értelmezése emberi szemmel nézve is nehézzé válik. Összességében tehát egy adatelem- zési problémaként kellett kezelni a feldolgozást, ahol előfordulhatnak pontatlan, zajos esetek, és az eredményt is becslésnek kell tekinteni.

A részproblémák statisztikai jellege miatt a hisztogramok elemzése gyakran segí- tett. A megfelelőtranszformációk elvégzését követően leolvashatóvá váltak az értékek eloszlásának jellemzői, becsülni lehetett az algoritmusok paramétereit, mint például a küszöbértékeket.

Készültek mérések arra vonatkozóan, hogy a különbözőstrukturális szinteken ho- gyan változik a becslés pontossága a futások számának függvényében. Nagy mennyiségű dokumentum feldolgozása esetén ez azért jó, mert így a teljes futási időis becsülhető lesz, illetve az összes eredmény ellenőrzése nélkül is lesznek információink a megbízha- tóságra vonatkozóan.

A szerkezeti elemzés bizonyos dokumentumok esetében nagyon bonyolult. A táblá- zatok, matematikai képletek, képek vagy egyéb, nem szöveges jellegűelemek felismerése és értelmezése jelenleg is kutatott terület. A dolgozat ezek kezelésére javaslatot tesz, de részletesebb vizsgálatuk külön, későbbi kutatás tárgyát képezi.

# Irodalomjegyzék

1. Gary Bradski and Adrian Kaehler.*Learning OpenCV: Computer vision with the OpenCV library*. " O’Reilly Media, Inc.", 2008.
2. Vaibhaw Singh Chandel. Deep learning based text recognition (ocr) using tesseract and opencv.[www.learnopencv.com,](http://www.learnopencv.com/) 2018.
3. Alex Clark. Pillow (pil fork) documentation, 2015.
4. Hiromichi Fujisawa, Yasuaki Nakano, and Kiyomichi Kurino. Segmentation met- hods for character recognition: from segmentation to document structure analysis. *Proceedings of the IEEE*, 80(7):1079–1092, 1992.
5. Aurélien Géron.*Hands-On Machine Learning with Scikit-Learn, Keras, and Ten- sorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, 2019.
6. Antonio Gulli and Sujit Pal.*Deep learning with Keras*. Packt Publishing Ltd, 2017.
7. Adobe Systems Incorporated.*Document Management - Portable Document For- mat*. 2008.
8. Python Package Index. pdf2image.https://pypi.org/project/pdf2image/, 2020.
9. Deliang Jiang and Xiaohu Yang. Converting pdf to html approach based on text detection. In*Proceedings of the 2nd international conference on interaction sci- ences: Information technology, culture and human*, pages 982–985, 2009.
10. Project Jupyter. Jupyter.https://jupyter.org, 2020.
11. Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada.*Optical character recog- nition*. John Wiley & Sons, Inc., 1999.
12. Travis E Oliphant.*A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.
13. Chirag Patel, Atul Patel, and Dharmendra Patel. Optical character recognition by open source ocr tool tesseract: A case study.*International Journal of Computer Applications*, 55(10):50–56, 2012.
14. Wikipedia. Floodﬁll.https://en.wikipedia.org/wiki/Flood\_fill, 2020.

## Útmutató a mellékelt fájlokhoz

A dolgozathoz tartozó melléklet az alábbiakat tartalmazza.

* + dolgozat.pdf: A szakdolgozat PDF formátumban.
  + szakdolgozat: A szakdolgozat L ATEX forráskódját tartalmazó jegyzék.

program: Az elkészített programok Python forráskódja. Ebben megtalálhatók a Python szkriptek, a Jupyter munkafüzetek, továbbá az elemzéshez összegyűjtött minta PDF dokumentumok is.

*•*

A programok használatához, a munkafüzetek megtekintéséhez Python értelmezőre és a dolgozatban említett függőségek telepítésére van szükség.

A dolgozathoz tartozó fájlok elérhetők*GitHub*-on is a következőcímen:

https://github.com/molnar75/szakdolgozat

*A cikkben/előadásban/tanulmányban ismertetett kutató munka az EFOP-3.6.1-16- 2016-00011 jelű„Fiatalodó és Megújuló Egyetem – Innovatív Tudásváros – a Miskolci Egyetem intelligens szakosodást szolgáló intézményi fejlesztése” projekt részeként – a Széchenyi 2020 keretében – az Európai Unió támogatásával, az Európai Szociális Alap társﬁnanszírozásával valósul meg"*