Gépi Látás

TKNB_INTM038

Rendszámtábla azonosítás

Paluska Máté

HOOXOH

Tata, 2022

Tartalomjegyzék

1 Bevezetés

Napjainkban rohamosan növekedik a forgalomban levő gépjárművek száma, aminek következtében egyre fontosabb szerepet kapnak a különböző gépjármű azonosító informatikai megoldások is. Az ilyen megoldások szükségességét indokolja az is, hogy a modern világban a monoton, gépies munkákat, feladatokat hatékony módon próbálunk automatizálni. Erre kiváló a példa a rendszámok azonosításának esete például egy parkolóházba való belépéskor, autópályákon matricavásárlás ellenőrzésére, lopott vagy körözés alatt álló autók szűrésére, figyelésére, stb. Ilyen esetekben könnyen belátható, hogy nagy szükség van ezekre a rendszerekre, mivel az emberi munkaerő számára ez pusztán csak olvasásból áll és rengeteg energia megspórolható vele. Sokkal könnyebbé és gyorsabbá válik az azonosítás ezen módszerekkel. Mindezek mellett a hibázás lehetőségének csökkentése sem utolsó szempont, mivel emberi munkavégzés esetén sokkal nehezebb a hibák valószínűségét alacsonyan tartani, míg szoftveres megoldással viszonylag magas pontosság is elérhető. A dolgozatomban egy ilyen rendszámtábla azonosító megoldást fogok bemutatni, amely képes egy bemeneti kép alapján a rendszámtábla felismerésére és azonosítására. Fontos tisztázni, hogy a rendszámtábla felismerés nem egyenlő az azonosítással, mivel a felismerés esetében csak a táblát és annak pozícióját határozzuk meg a képen, a tábla szöveges tartalmát már nem olvassuk le róla, míg az azonosítás esetében ez is megtörténik. Ezek mellett nem csak a magyar rendszámtípusok leolvasása a cél, hanem bármilyen olyan rendszám, amely a latin ABC betűit és/vagy számok kombinációit tartalmazza.

2 Feladat elméleti háttere

2.1 Kép szürkeárnyalatossá alakítása

Első lépésként a kép BGR színkódjából egy szürkeárnyalatos képet állítunk elő.

2.2 Bilaterális szűrő használata

A szürkeárnyalatos képen ezután egy bilaterális (kétoldalú) szűrést végzünk. Ez a szűrés egy alapvető művelet a képfeldolgozásban, zajcsökkentés mellett az éleket jobban kivehetően tartja.

A szűrés jelentése ebben a kontextusban, hogy a kép értéke egy megadott helyen a környező képpontok súlyozása alapján lesz kiszámítva. A súlyozás a Gauss féle eloszlást követi, ami azt jelenti, hogy a képponthoz közelebb álló pixelek nagyobb súlyozással számítanak bele a súlyozott átlag kiszámításába. Ez azért van, mivel a képek jellemzően lassan változnak a térben, ezért a középponthoz közelebb álló pixelek nagyobb eséllyel fognak hasonló értékekkel rendelkezni.[1]

A bilaterális szűrés alapötlete az, hogy a kép tartományában ugyanaz hajtódik végre, ami a hagyományos szűrők esetén a saját területükön. Két pixel lehet közel egymáshoz (térben) vagy lehet egymáshoz hasonló (hasonló értékeik vannak).[1]

A térbeli eltérések feltérképezése meghiúsul az éleknél, aminek következtében az élek elmosódnak a képen. Ennek a problémának a megoldására lehet alkalmas az anizotrop diffúzió, ahol parciális differenciálegyenletek megoldásával átlagolnak a problémás területekre.[1]

2.3 Éldetektálás

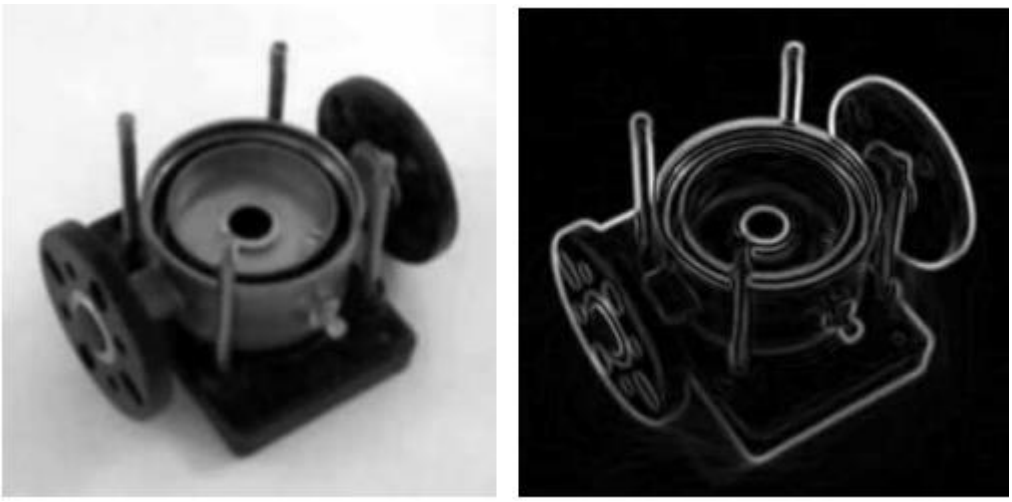
Az éldetektálást célja, hogy jelentősen csökkentsük az adatmennyiséget a képen, miközben a strukturális jellemzőket megőrizzük. Sokféle megoldás létezik éldetektálásra, én a projektemben a Canny algoritmust használtam, melyet John F. Canny dolgozott ki a 80-as években.[2]

A Canny algoritmus 5 fő lépésből áll:

- **Simítás**: *A kép elhomályosítása zajcsökkentés céljából.*
- **Színátmenetek keresése**: *Az élek megjelölése, ahol a kép színátmenetei nagyok.*
- **Non-maximum suppression (nem-maximális elnyomás)***: Csak a lokális maximumokat kell élként megjelölni.*
- **Kettős küszöb***: A lehetséges éleket egy küszöbérték határozza meg.*
- **Élkövetés hisztérzissel***: A legvégső élek meghatározás olyan módon, hogy eltüntetjük az összes olyan élt, amely nem kapcsolódik egy erősebb élhez.*

Minden kamerával készített képen található valamennyi zaj. Ezért **simításkor** egy Gauss szűrő alkalmazásával kicsit homályosabbá tesszük a képet.[2]

Az algoritmus ott talál éleket, ahol a szürkeárnyalat intenzitása leginkább változik. Ezeket a területeket a **színátmenetek keresésével** lehet meghatározni. Ezeket az átmeneteket a Sobel-operátor segítségével lehet meghatározni. Első lépésként a színátmenet x és y irányú közelítését kell végrehajtani egy kernel segítségével. Az átmenet nagyságait ezután euklideszi távolságmértékként tudjuk meg határozni a Pitagorasz tétel alkalmazásával. A színátmenetek keresése utáni állapot az **1. ábrán** látható.[2]



1. ábra: simítás után, majd a színátmenetek keresése után.
Forrás:[2] 3. o. 2. ábra

A **non-maximum suppression** lényege, hogy a képen látható vastag éleket vékonyabbá alakítsuk. Ez úgy történik, hogy az algoritmus az összes helyi maximumot megőrzi a képen a többit pedig törli. Ezáltal vékony, élesebb éleket kapunk.[2]

Az élek vékonyítása után még mindig lehetnek nem valós élek a képen, melyeket okozhatnak zajok vagy színeltérések a durva felület miatt. Hogy ezeket megkülönböztessük egymástól, egy küszöböt kell használnunk, így csak egy bizonyos értéknél erősebb élek maradnak meg. A **kettős küszöb** az jelenti a Canny algoritmus esetében, hogy a felső küszöbnél erősebb pixelek értékeit „erősnek” jelöli meg, az alsó küszöb alatti értékeket elhagyja, a két küszöb közötti értékeket pedig „gyengének” jelöli.[2]

Az erősnek jelölt élek esetében biztosak lehetünk, hogy azok valódi élek, a gyengék esetében csak akkor lehetünk biztosak benne, ha valamelyik erős élhez kapcsolódnak, mivel a zaj és más apró eltérések valószínűleg nem eredményeznek erős éleket. Az **élkövetést** BLOB (Binary Large Object) analízissel lehet megvalósítani. Az élek pixeleit szét kell bontani összekapcsolt BLOB-okra, egy BLOB tartalmazza a középső pixelt és a 8 darab őt körülvevő pixelt is. Azok a BLOB-ok amelyek legalább 1 erős jelölést tartalmaznak megmaradnak, a többi nem.[2]

2.4 Zárt téglalapok keresése

A sikeres éldetektálás után, a zárt téglalapok keresésének módszerével ki tudjuk használni a rendszámtábla alakját, mivel szinte minden esetben téglalap alakúak. A bináris képen meg tudjuk vizsgálni az élek alakzatát, hány oldaluk van, hol helyezkednek el. Ezekből az információkból ki tudjuk számolni a négyszögek magasságának és szélességének arányát, amiből tudunk következtetni hogy megfelelő méretű téglalap lehet e.

2.5 Optikai karakterfelismerés

Az optikai karakterfelismerés célja, hogy a nyomtatott szöveget és képeket digitalizálja, azokat gépileg kezelhetővé tegye. Ez egy összetett probléma a nyelvek, betűtípusok sokfélesége miatt.[3]

Az optikai karakterfelismerés általános lépési:

- **Képbeszerzés** (Image acquisition)
- **Előfeldolgozás** (Preprocessing)
- **Karakter szegmentáció** (Character segmentation)
- **Jellemzők kinyerése** (Feature extraction)
- **Karakterek besorolása** (Character classification)
- **Utófeldolgozás** (Post processing)

A **képbeszerzés** után következő **előfeldolgozás** a képminőség javítását célozza. Itt különböző típusú szűrők alkalmazhatók, például átlagoló, minimum és maximum szűrők, ezek mellett különböző morfológiai műveletek is végezhetők (pl. erózió, tágítás, nyitás, zárás). Fontos része az előfeldolgozásnak a dokumentum ferdeségének kiderítése. Erre is többféle technológia létezik, például Hough-transzformáció, legközelebbi szomszédsági algoritmusok. Az előfeldolgozás végére már ismerjük a szövegsorok pozícióját, a pixelek vetületei vagy klaszterezése alapján.[3]

A **karakter szegmentációs** lépésnél a képet karakterekre kell bontani, ami megtörténhet explicit vagy implicit módon az osztályozási fázis közben is. Ezen kívül a többi fázis is segíthet a hasznos kontextuális információk megszerzésében a kép szegmentálásához.[3]

A **Feature extraction** szakaszban a karakterek különféle jellemzőit kell kivonni, amik képesek egyedileg azonosítani az adott karaktert, mint például a geometriai jellemzők (vonások, hurkok). Végső soron a kép méretét is csökkenteni kell bizonyos esetekben, melyekre szintén különböző technikák alkalmasak (pl. főkomponens-elemzés).[3]

Ezután a **karakterek besorolása**, osztályozása következik, amelynek a megvalósítása a képkomponensekben található kapcsolatokon alapul. Vannak statisztikai megközelítések, melyek egy osztályozó diszkriminációs függvény használatán alapulnak, ilyen pl. a Bayes-osztályozó, döntési fa-osztályozó, neurális hálózatokkal végzett osztályozás vagy a legközelebbi szomszédság módszerén alapuló osztályozók. Ezek mellet vannak olyan egyéb olyan osztályozó megoldások is, melyek szintaktikai megközelítsen alapulnak, ezek nyelvtani megközelítést feltételeznek.[3]

Az utófeldolgozás főbb célja a karakterfelismerés eredményének javítása. Itt lehet használni különböző megoldásokat, az egy ilyen megközelítés lehet az, hogy egynél több osztályozót használunk és összevetjük az eredményeket. Az eredmények javítása érdekében kontextuális elemzés is végezhető, ezek mellett a Markov-modelleken és szótáron alapuló lexikális feldolgozás is hasznos lehet.[3]

3 Kivitelezés

3.1 megoldás OCR használatával

A kivitelezésnél két részre osztottam a programot, az első részben az EasyOCR segítségével történik a karakterek felismerése, a másodiknál pedig egy általam készített algoritmus hasonlítja össze a rendszámtáblán található karaktereket az ABC betűivel, majd a legpontosabb egyezést kiválasztva próbálja meghatározni az eredeti karaktereket. Ezzel a bontással a végén össze lehet hasonlítani az OCR és a saját megoldásom pontosságát.

3.1.1 Felhasznált könyvtárak

1. kódrészlet: felhasznált könyvtárak

```
import easyocr
import cv2
import imutils
import skimage
import os
```

Az **1. kódrészletben** láthatóak a projekthez felhasznált könyvtárak. A *cv2* library tartalmazza az openCV nevű nyílt forráskódú könyvtárat, amely alapvető funkciókat tartalmaz a gépi látás és a képfeldolgozás témaköréhez kapcsolódóan. A már korábban említett *easyocr* library az optikai karakterfelismeréshez szükséges. Az *imutils* könyvtárra a kontúrok kezelése miatt volt szükség, a *skimage* különböző algoritmusokat tartalmaz a képfeldolgozáshoz kapcsolódóan, az én megoldásom szempontjából csak a képek összehasonlítása miatt volt rá szükség. Az *os* library az operációs rendszertől függő funkciókat tartalmaz.

3.1.2 Előfeldolgozás

2. kódrészlet: előfeldolgozás lépései

```
img = cv2.imread('resources/sjz2_.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

bfilter = cv2.bilateralFilter(gray, 11, 17, 17)  # noise reduction
cv2.imshow("bilateral", bfilter)
imgCanny = cv2.Canny(bfilter, 30, 200)  # edge detection
cv2.imshow('canny', imgCanny)
points = cv2.findContours(imgCanny.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
contours = imutils.grab_contours(points)
contours = sorted(contours, key=cv2.contourArea, reverse=True)[:10]
```

A **2. kódrészletben** a kiválasztott kép beolvasása történik, majd a beolvasott kép szürkeárnyalatossá alakítása. Ezután a szürkeárnyalatos képen egy bilaterális szűrést végzünk a zajcsökkentés érdekében. Itt meg kell adnunk második paraméterként a szűrő méretét és utána két szigma értéket is. Minél magasabb értékeket adunk meg annál erősebb lesz a szűrő hatása, itt kicsit magasabb értékeket állítottam be, mivel azt vettem észre, hogy növeli a pontosságot. Ez után a kapott képen végrehajtjuk az éldetektálást a Canny algoritmus segítségével, ahol a bemeneti kép mellet még az alsó és felső küszöbértékeket is meg kell adnunk a hiszterézis küszöböléshez. A *findContours* metódus használatával hozzájutunk a kontúrokhoz egy tömbben, ahol az észlelt körvonalak pontjai vektorokban vannak tárolva. Második paraméterként a kapott kontúrok hierarchiáját állítjuk be. A RETR_TREE megadásával egy teljes „család” hierarchiát kapunk, melyben látható, hogy az egyes kontúrok milyen szinten vannak a többihez képest. A harmadik paraméterként egy kontúrközelítő algoritmust kell kiválasztanunk, a CHAIN_APPROX_SIMPLE kiválasztásával csak a sarokpontokat kapjuk meg az egyenesek mentén, ezért kevésbé lesz időigényes futás közben és kevesebb memóriát is használunk. Miután hozzájutottunk a kontúrok sarokpontjaihoz, sorba rendezzük őket a méretük alapján és csak az első tíz darabot tartjuk meg, ezáltal a kisebb alakzatoktól meg tudunk szabadulni, mivel tudjuk, hogy a rendszámtábla a képen a nagyobbak között lesz.

3.1.3 Téglalapok keresése és vizsgálata

3. kódrészlet: kontúrok vizsgálata

```
while True:
    location = None
    foundRectangle = False

    if not bool(contours):
        break

    for contour in contours:
        approx = cv2.approxPolyDP(contour, 10, True)
        if len(approx) == 4:
            foundRectangle = True
            (x, y) = approx[1][0]
            (x3, y3) = approx[3][0]
            (x2, y2) = approx[2][0]
            (x0, y0) = approx[0][0]
            height = y3 - y
            width = x0 - x2
            aspectRatio = width / height
            if aspectRatio >= 2 and aspectRatio <= 8:
                location = approx
                contours.remove(contour)
                break
            contours.remove(contour)
            break

    if not foundRectangle:
        print("License plate not found.")
        break
```

A sikeres előfeldolgozás után következik a téglalapok keresése a kontúrokból. Ahogy a **3. kódrészletben** is látható, egy végtelen ciklust nyitunk, amiből három féle módon lehet kijutni:

1. ha nem találtunk kontúrokat a képen
2. ha a kontúrok között nincs olyan alakzat, amelynek négy sarka van
3. ha találtunk olyan alakzatot aminek négy sarka van és az OCR is sikeresen lefutott rajta

Egy for ciklusban vizsgáljuk a korábban megkapott kontúrokat tartalmazó tömb elemeit, ahol az *approxPolyDp* fügvénnyel kvázi összekötjük az első alakzat sarokpontjait és egy új tömbbe mentjük a kapott eredményt. Az új tömb hossza megegyezik az alakzat oldalainak számával, ezért meg tudjuk vizsgálni, hogy négy oldala van e. Ha négy oldala van akkor kiszámoljuk a négyszög szélességét és magasságát, amit elosztva egymással megkapjuk a méretarányt. ha a méretarány 2 és 8 közé esik, akkor biztosak lehetünk benne, hogy egy téglalapról van szó és tovább léphetünk a karakterfelismeréshez. Mielőtt továbblépnénk, a vizsgált alakzatot eltávolítjuk az eredeti *contours* tömbből, hogy amikor újra vizsgáljuk ezt a tömböt, akkor csak olyan alakzatok legyenek benne, amelyeket még nem vizsgáltunk. Ez abban az esetben következik be, amikor téglalapot találtunk, de a karakterfelismerés fázisában nem találtunk karaktereket, ilyenkor visszakerülünk a while ciklus elejére és újra kezdődik az alakzatok vizsgálata.

3.1.4 easyOCR használata

4. kódrészlet: easyOCR használata

```
    if(location is not None):
        cv2.rectangle(img, (location[0][0]), (location[2][0]),
                      color=(0, 255, 0))
        cv2.imshow('plate found', img)

        plate = cv2.boundingRect(location)
        x, y, w, h = plate

        roi = img[y:y+abs(h), x:x+abs(w)]  # height, width

        reader = easyocr.Reader(['en'])
        result = reader.readtext(roi)

        if bool(result):
            print("License plate found!")
            print(result)
            break


print("OCR Program end.")
```

A **4. kódrészletben** még mindig a korábban említett végtelen while cikluson belül vagyunk, ahol a *location* változóban visszük tovább a megtalált négyszögünket. Ha ez a változó üres, akkor a négyszög nem felelt meg a méretaránynak és visszatérünk a ciklus elejére. Amennyiben nem üres, a *cv2.rectangle* funkcióval az eredeti képen láthatóvá tesszük a téglalapunkat a két szemközti sarokpont magadásával és egy szín kiválasztásával. A *boundingRect* metódus használatával megkapjuk az x és y koordinátákat, a magasságot és a szélességet, amik segítségével ki tudjuk vágni az eredeti képből a feltételezett rendszámtáblát, majd bemenetként tudjuk adni a OCR-nak. A *result* változóban csak akkor kapjuk meg az OCR futtatásának eredményét, ha talált karaktereket az átadott képen. Ebben az esetben kilépünk a ciklusból és a program első részének a futása sikeresen lezajlott. Ellenkező esetben a változó üres marad és újra kezdődik a téglalap keresés.

3.2 Megoldás OCR használata nélkül

3.2.1 Előfeldolgozás és téglalapok keresése

A második programrészhez nem alkalmazunk újabb előfeldolgozást, mivel az előző programrészből ugyanúgy tudjuk használni az előfeldolgozáson átesett képünket. A téglalapok keresése is teljesen ugyanazon elv mentén történik, mint az első programrésznél, amíg el nem jutunk arra a pontra, ahol már megvan a kivágott téglalapunk és a karakterfelismerés következne. Ahogy az **5. kódrészletben** is látszik, a *getPlateChars* függvény kerül meghívásra az easyOCR helyett, ami egy téglalapot (a rendszámtáblát) vár paraméterül és a felismert karakterekkel tér vissza egy string-ben.

5. kódrészlet: OCR kiváltása

```
roi = img[y:y+abs(h), x:x+abs(w)]  # height, width

result = getPlateChars(roi)

if bool(result):
    print("License plate found without OCR!")
    print(result)
    break
```

3.2.2 Thresholding (küszöbölés)

6. kódrészlet: küszöbölés

```
def getPlateChars(foundPlate):

    foundPlate_gray = cv2.cvtColor(foundPlate, cv2.COLOR_BGR2GRAY)

    thresh_foundPlate = cv2.threshold(foundPlate_gray, 0, 255,
cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

A **6. kódrészletben** a paraméterként átadott rendszámtáblát először szürkeárnyalatossá alakítjuk a már ismert módon, majd a *threshold* metódussal egy küszöbölést hajtunk végre a képen. Ez annyit jelent, hogy minden pixelt megvizsgál az algoritmus és attól függően, hogy a küszöbérték alatt vagy felett van a pixel értéke, 0-ra vagy 255-re állítja az értéküket (2. és 3. paraméter). A *THRESH_BINARY_INV* paraméter megadásával érjük el a bináris kimeneti kép inverzét. A *THRESH_OTSU* paraméter egy optimális küszöbértéket számol ki a kép hisztogramjából és azzal dolgozik. Ezek után a **2. ábrán** látható kimenetet kapjuk.

2. ábra: küszöbölés után

3.2.3 Karakterek elkülönítése

7. kódrészlet: határoló dobozok megjelenítése

```
contours, hierarchy = cv2.findContours(thresh_foundPlate, cv2.RETR_CCOMP,
cv2.CHAIN_APPROX_SIMPLE)

boundRect = []

for i, contour in enumerate(contours):

    if hierarchy[0][i][3] == -1:
       boundRect.append(cv2.boundingRect(cv2.approxPolyDP(contour,3,True)))

for i in range(len(boundRect)):
    color = (0, 255, 0)
    cv2.rectangle(foundPlate, (int(boundRect[i][0]), int(boundRect[i][1])),
(int(boundRect[i][0] + boundRect[i][2]), int(boundRect[i][1] +
boundRect[i][3])), color, 2)
```

A **7. kódrészletben** a bináris képünkből kinyerjük a kontúrokat és azok hierarchia értékeit is. Itt a *findContours* második paraméterének a *RETR_CCOMP* flag-et adjuk meg, ami annyit jelent, hogy kétszintű hierarchiába rendezi a kontúrokat, így meg tudjuk különböztetni a karakterek külső és belső kontúrjait (például az A betű belső háromszög részét). Ezután végig iterálunk a *contours* tömbön és csak azokra a kontúrokra van szükségünk, ahol hierarchia érték -1, tehát a kontúr az objektum (karakter) külső részéhez tartozik. Ha ez teljesül, akkor a ciklus előtt létrehozott *boundRect* tömbbe írjuk az adott karakter x, y, szélesség és magasság értékeit, majd ezek segítségével a következő for ciklusban a *rectangle* metódus segítségével meg tudjuk jeleníteni az eredeti képen a megtalált határoló dobozokat, ahogy a **3. ábrán** is látható.

3. ábra: karakterek határoló dobozai

Ezután sorbarendezzük az új tömbünk elemeit az x koordináták alapján, hogy a határoló dobozok olyan sorrendben legyenek, mint ahogy a képen vannak. Ez látható a **8. kódrészletben**.

8. kódrészlet: sorbarendezés

```
n = len(boundRect)
for i in range(n - 1):
    for j in range(0, n - i - 1):
        xj = boundRect[j]
        xi = boundRect[j + 1]
        if xi < xj:
            boundRect[j], boundRect[j + 1] = boundRect[j + 1], boundRect[j]
```

3.2.4 karakterek összehasonlítása

9. kódrészlet: karakterek kivágása

```
counter = 1
finalPlateNumber = ""
for i in range(len(boundRect)):

    x, y, w, h = boundRect[i]

    croppedImg = thresh_foundPlate[y:y + h, x:x + w]

    if w > 10 and h > 10 and len(croppedImg) and w < 300:
        croppedImg = cv2.resize(croppedImg, (40, 40))

        folder_dir = ""
        if counter <= 3:
            folder_dir = "SampleLetters/letters"
        else:
            folder_dir = "SampleLetters/numbers"
```

A karakterek összehasonlításához végig kell iterálnunk a *boundRect* tömbön és minden elemet össze kell hasonlítanunk az abc összes betűjével vagy számokkal, attól függően, hogy hányadik karakterét vizsgáljuk a rendszámnak. Ezt úgy oldottam meg, hogy a *SampleLetters/letters* mappában minden betűről eltároltam egy kivágott képet, amelyeket össze tudok hasonlítani a vizsgált

határolódoboz tartalmával. Ugyanígy a számok esetében is. Fontos megjegyezni, hogy minden karakternél pixel pontossággal kell méretre vágni a minta képeket, mivel a határoló dobozok is így kerülnek rá a képre. Ha fehér pixeleket hagyunk a minta képek szélén, jóval pontatlanabb eredményeket kapunk az összehasonlításánál. A **9. kódrészletben** látható, hogy a már ismert módon az x, y, szélesség és magasság értékekből ki tudjuk vágni az adott határoló doboz tartalmát és szűrni tudjuk, hogy csak a 10 pixelnél szélesebb és magasabb, illetve a 300 pixelnél keskenyebb dobozok tartalmát vizsgáljuk csak. Ez azért fontos mert azok a méretek amik ezen kívül esnek, valószínűleg nem tartalmaznak semmilyen karaktert a méretükből adódóan. A ciklus előtt még létrehozunk egy counter változót, amivel számontartjuk, hogy éppen hányadik elemet vizsgáljuk, mivel az első három karakter a rendszámban biztosan betű, az utolsó három pedig biztosan szám. Fontos, hogy az összehasonlítás előtt a kivágott képrészt egyforma méretűre kell konvertálni a *resize* metódussal.

10. kódrészlet: karakterek összehasonlítása

```
        maxScore = -10
        letter = ""
        for image in os.listdir(folder_dir):

            path = folder_dir + "/" + image

            charToCompare = cv2.imread(path)
            charToCompare = cv2.resize(charToCompare, (40, 40))
            charToCompare = cv2.cvtColor(charToCompare, cv2.COLOR_BGR2GRAY)

            charToCompare = cv2.threshold(charToCompare, 0, 255,
cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]

            score = skimage.metrics.structural_similarity(croppedImg, char-
ToCompare)

            if score > maxScore:
                maxScore = score
                letter = image

        # felségjelzés kiküszöbölése
        if letter != "rossz.png":
            counter += 1
            finalPlateNumber += letter[0]

return finalPlateNumber
```

A **10. kódrészletben** látható, ahogy a *listdir* metódus segítségével végig tudunk iterálni a mintaképeket tartalmazó mappán, ahol minden egyes képet átméretezünk és végrehajtunk egy küszöbölést a korábban már ismertetett módszerrel. Ezután a *skimage.metrics.structural_similarity* metódusával összehasonlítjuk a kivágott határoló doboz tartalmát az összes minta karakterrel, ez visszatér egy 0 és 1 közötti értékkel (minél pontosabb az egyezés, annál magasabb a szám.). Ez a szám lesz a két kép átlagos szerkezeti hasonlósági indexe. Majd egy maximum kereséssel kiválasztjuk a legmagasabb értéket, majd a végén hozzáfűzzük a végső string-hez. A felségjelzést gyakran felismerte a program, mint önálló karaktert, ezért a mintaképek közé beraktam egy mintát arról is, hogy ha azzal mutatja a legnagyobb hasonlóságot, akkor ne kerüljön bele a rendszám betűi közé. Amikor a teljes programrész lefutott, akkor a függvény visszatér a *finalPlateNumber* értékkel, ami gyakorlatilag a leolvasott rendszámot tartalmazza. Ennek végeztével a program végéhez értünk és a konzolon megjelenik a leolvasott rendszám.

4 Tesztelés

1. Táblázat: tesztek eredménye

Kép neve	*OCR használatával*	*OCR nélkül*	*Hiba oka*
eqv	x	x	nem talált téglalapot
ford2	ok	x	több hibás karakter
ford3	ok	x	több hibás karakter
lzp	x	x	nem talált téglalapot
maa	x	x	OCR 1-el kevesebb karakter
nle	ok	ok	
nnb	ok	ok	
pav	x	x	több hibás karakter
pav2	x	x	rossz téglalap
pav4	x	x	nem talált téglalapot
pba2	ok	ok	
pba3	x	x	nem talált téglalapot
pba4	ok	ok	
pda	x	x	nem talált téglalapot
ppz2_	ok	x	több hibás karakter
pya	ok	ok	
rew	x	x	nem talált téglalapot
rkz_	ok	x	több hibás karakter
rkz2	ok	ok	
rmz	ok	x	több hibás szám
rny	x	x	nem talált téglalapot
rvz	x	x	nem talált téglalapot
rvz1_	ok	x	több hibás karakter
rvz2	ok	ok	
saa1	x	x	rossz téglalap
saa3_	ok	x	több hibás karakter
saa4_	ok	ok	
saa7	ok	x	több hibás karakter
sab2	ok	ok	
sab3	ok	ok	
sab5_	ok	x	több hibás karakter
sab6	ok	ok	
sbj_	ok	x	egy karakter hiányzik
sbj2	ok	x	több hibás karakter
sbk	x	x	nem talált téglalapot
sez2	x	x	nem talált téglalapot
sez3	ok	x	több hibás karakter
sez4_	ok	x	több hibás karakter
sez5	x	ok	eggyel több karakter
sfz_	ok	x	több hibás karakter
sfz1_	ok	x	eggyel kevesebb karakter
sfz2	ok	ok	
sfz3	ok	x	több hibás karakter
sjz	ok	x	több hibás karakter
sjz2_	ok	ok	
sjz10_	ok	x	több hibás karakter
snz2	x	x	rossz téglalap
snz3	ok	x	több hibás karakter
sra3	ok	ok	
srh2	x	x	nem talált téglalapot
suz	ok	ok	
suz2	ok	ok	
suzuki1	x	x	rossz téglalap
suzuki2	x	x	nem talált téglalapot
svz3	x	x	rossz téglalap
taa	x	x	nem talált téglalapot
tbv	x	x	nem talált téglalapot
tcu2_	ok	ok	
tesla1	ok	ok	
tesla3	ok	x	egy karakter hiányzik
össz.:	***38/60***	***19/60***	
	63,30%	***31,60%***	

Az **1. táblázatban** láthatóak a teszteredmények, összesen 60 darab képen teszteltem a programot, az OCR használatával 63,3%-os pontosságot sikerült elérni, míg az OCR nélküli résszel csak 31,6%-ot. A tesztkészlet a cars mappában található a projekten belül. A leggyakoribb hibaok közé tartozik, hogy a program nem talált téglalapot a képen, amit azzal lehetne javítani, hogy az előfeldolgozásnál más paraméterek beállításával is próbálkozunk. Másik gyakori hibaok volt, hogy túl sok határoló doboz keletkezett a képen, ezért a karakterek száma is több volt, mint amennyi a valóságban, illetve előfordult néhány karaktertévesztés is. Amennyiben a határoló dobozok a megfelelő helyre kerülnek, abban az esetben jelentősen javul a program pontossága.

5 Felhasználói leírás

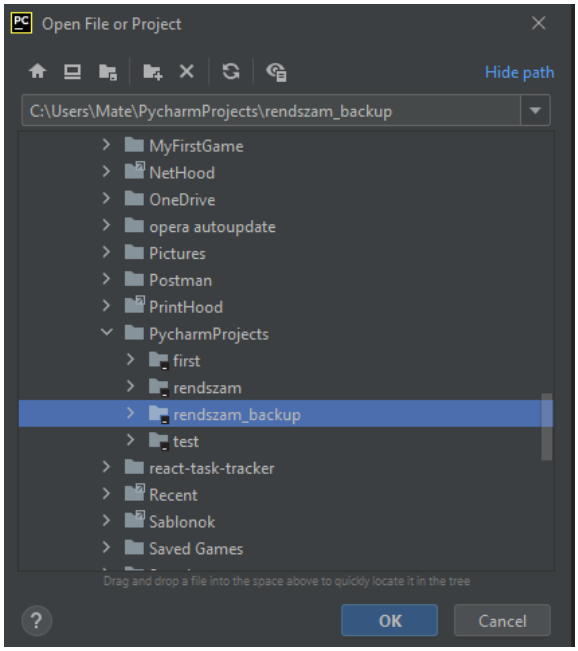
A program futtatásához szükségünk van egy fejlesztő környezetre, én a PyCharm-ot használtam, de bármilyen környezet megfelelő lehet, ahol van lehetőség python csomagok telepítésére. A PyCharm az alábbi linken letölthető:

4. https://www.jetbrains.com/pycharm/

Ezen kívül szükségünk van a Python 3.7.1-es verziójára, mivel az easyOCR még nem kompatibilis az újabb verziókkal.

5. https://www.python.org/downloads/release/python-371/

Ha ezekkel megvagyunk a PyCharm megnyitása után a file menüpontban az *open* fülre kattintunk, és a megjelenő ablakban kiválasztjuk a mappát, amelyik a projektünket tartalmazza, ahogy a **4. ábra** is mutatja.

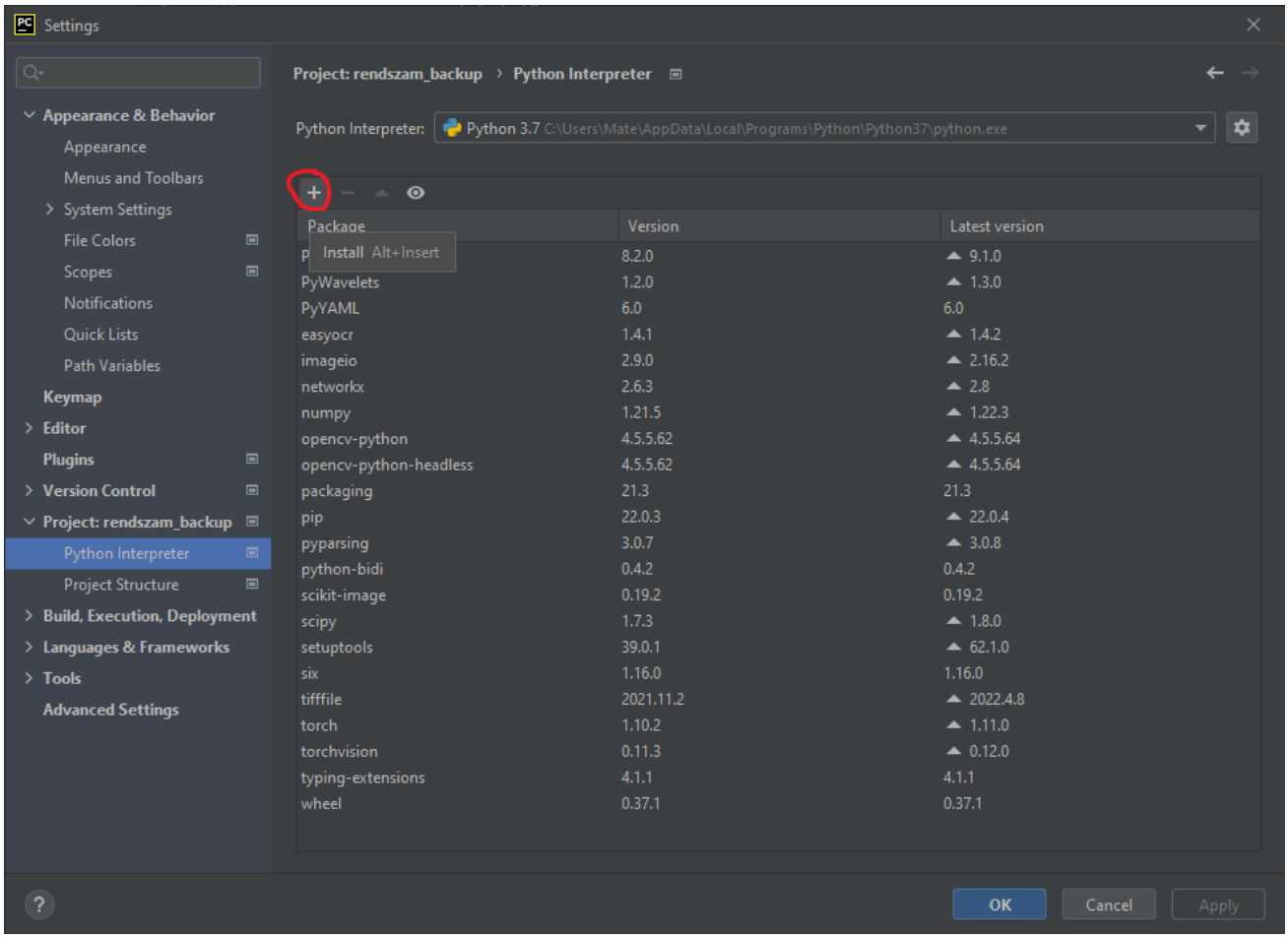


4. ábra: projekt kiválasztása

Miután betöltöttük a projektet, fel kell telepítenünk az alábbi szükséges csomagokat:

- opencv 4.5.4.60
- easyocr 1.4.1
- imutils
- skimage

Ezt úgy tudjuk megtenni, hogy a *file* menüpontra kattintva kiválasztjuk a *settings* menüpontot, majd a felugró ablakban a project fül alatt a *python interpreter*-re kattintva jelenik meg az eddig feltelepített csomagok listája. Itt a jobb bal felső sarokban a hozzáadás gombra kattintunk, ahogy az **5. ábra** is mutatja. Ezután felugrik a kereső, ahova beírva a csomagok nevét, telepíteni tudjuk őket. Fontos, hogy az *opencv* és az *easyocr* csomagok a megfelelő verziójúak legyenek.



5. ábra: csomagok hozzáadása

Ha ezekkel megvagyunk, utána a program elején a *cv2.imread* metódus segítségével tudjuk beolvasni a vizsgálni kívánt képet, ezután pedig csak futtatnunk kell a programot.

6 Felhasznált irodalom

[1] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images", *Proceedings of the 1998 IEEE International Conference on Computer Vision*, Bombay, India.

[2] Moeslund, T. (2009, március 23.), Canny Edge Detection.

[3] Noman Islam, Zeeshan Islam, Nazia Noor, "A Survey on Optical Character Recognition System", *Journal of Information & Communication Technology-JICT Vol. 10 Issue. 2, December 2016*