Kennzeichenerkennung

December 18, 2017

1 Bibliotheken importieren

```
In [1]: import glob
        import numpy as np
        from scipy import ndimage
        from skimage.transform import rotate
        import imageio as io
        from scipy import misc
        import matplotlib.pyplot as plt
        import configparser
        import cv2
        import math
```

2 Bilder importieren

Als erstes laden wird alle Bilder aus unserem Datenverzeichnis in den Speicher.

```
In [2]: jpg = []
        for image_path in glob.glob("./data/*.jpg"):
            jpg.append(io.imread(image_path))
        rawData = np.asarray(jpg).astype('uint8')
```

3 Label und Metadaten aus INI-Dateien importieren

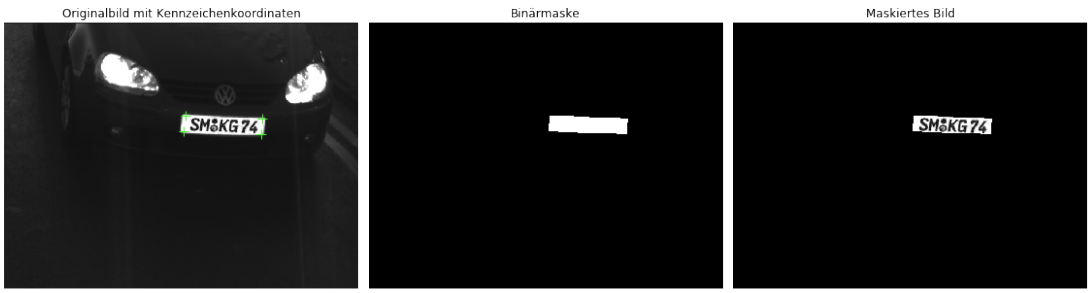
Als Nächstes parsen wir Labels und Koordinaten aus den zugehörigen INI-Dateien.

```
In [3]: labels = []
        coords = []
        config = configparser.ConfigParser()
        for ini_path in glob.glob("./data/*.ini"):
            config.read(ini_path)
            c = config['Plate']
            labels.append(c['name'])
            coords.append([c['x1'], c['x2'], c['x3'], c['x4'], c['y1'], c['y2'], c['y3'], c['y4']])
        labels = np.asarray(labels)
        coords = np.asarray(coords).astype(int)
```

4 Maskierung

In diesem Schritt verwerfen wir alle Informationen im Bild, die für die Kennzeichenerkennung unwichtig sind. Wir erstellen ein SchwarzweiSS-Maske aus den Koordinaten-Vektoren **x** und **y** und stellen diese als Subplot dar. Danach multiplizieren wir die Pixelfarben der Maske und des Originalbilds elementweise miteinander. Das Ergebnis ist ein gröSStenteils geschwärztes Bild mit isoliertem Kennzeichen.

```
In [4]: i = 0
        fig, (p1, p2, p3) = plt.subplots(1, 3, figsize=(20,20))
        #Raw Image
        p1.title.set_text('Originalbild mit Kennzeichenkoordinaten')
        p1.axis('off')
        p1.imshow(rawData[i], cmap='gray')
        x = coords[i][0:4]
        y = coords[i][4:8]
        p1.plot(x, y, '+',color='#42FA25', markersize=10)

        #Mask
        p2.title.set_text('Binärmaske')
        p2.axis('off')
        mask = np.zeros(rawData[i].shape[:2], dtype = 'uint8')
        rect = np.array((x,y)).T
        cv2.drawContours(mask, [rect], 0, 1, -1)
        p2.imshow(mask, cmap='gray')

        #Masked Image
        p3.title.set_text('Maskiertes Bild')
        p3.axis('off')
        masked =  rawData[i] * mask
        p3.imshow(masked, cmap='gray')
        plt.subplots_adjust(wspace = 0.03)
        plt.show()
```



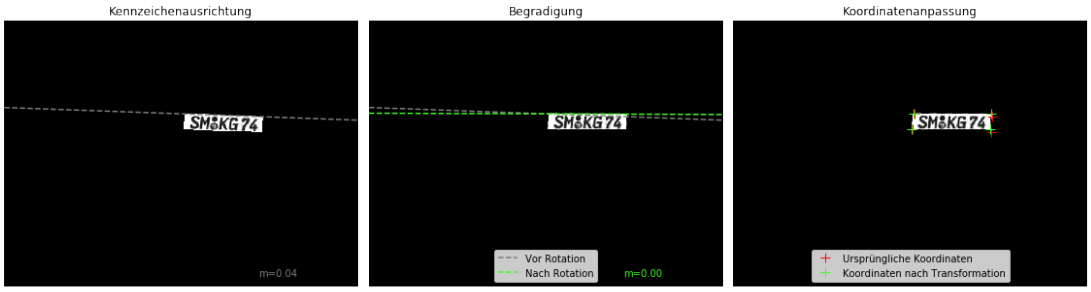
5 Kennzeichen begradigen

Der nächste Schritt ist die Begradigung der oberen und unteren Kante des Kennzeichens. Dazu errechnen wir die Steigungen der beiden Kantengeraden und berechnen ihr Mittel. Mithilfe des Arkustangens ermitteln wir den Winkel dieses Anstiegs und konvertieren diesen anschlieSSend von Radiant in BogenmaSS. Die Umwandlung ist für den Funktionsaufruf *rotate()* notwendig. Damit rotieren wir das zuvor ausgeschnittene Bild so, dass die obere und untere Kante begradigt werden. Bei perspektivischen Verzerrungen ist dieser Lösungsansatz nicht 100% perfekt, aber vollkommen ausreichend.

Durch die Rotation des Bildes entsteht ein Problem, denn die Eckpunkte des Kennzeichens stimmen jetzt nicht mehr mit den angegebenen Polygon-Eckpunkten aus unserer INI-Datei überein. Die Rotation, die wir am Bild vorgenommen haben, müssen wir deshalb auch genauso auf die Polygon-Eckpunkte übertragen. Da dafür kein spezieller Befehl zur Verfügung steht, muss dies händisch durchgeführt werden. Wir erzeugen defür erst einmal eine Rotationsmatrix **R** mit dem Winkel **alphaInRad** um den wir zuvor begradigt haben und definieren unseren Rotationsursprung **origin** in der Mitte des Bildes. Eine einfache Multiplikation mit der Rotationsmatrix würde um den Nullpunkt rotieren. Da wir aber um den definierten Rotationsursprung **origin** rotieren wollen, müssen wir diesen zuvor von der Punktkoordinate subtrahieren. Durch die Multiplikation mit der Rotationsmatrix erfolgt dann die eigentliche Rotation und um den Rotationsprozess abzuschlieSSen, addieren wir den Rotationsursprung wieder auf. Zur Übersicht zeichnen wir die alten Koordinaten rot und die neuen Koordinaten grün in einen Subplot ein.

```
In [5]: fig, (p1, p2, p3) = plt.subplots(1, 3, figsize=(20,20))

        #Ursprüngliches Bild aus letztem Schritt anzeigen
        p1.title.set_text('Kennzeichenausrichtung')
        p1.axis('off')
        p1.imshow(masked, cmap='gray')

        #y-Koordinate der Gerade bei x=0 und x=640 bestimmen
        m1 = (y[0] - y[3]) / (x[0] - x[3]);
        m2 = (y[1] - y[2]) / (x[1] - x[2]);
        b = y[0] - m1 * x[0]
        yend = m1 * 640 + b
        xline = [0, 639]
        yline = [b, yend]
        p1.plot( xline, yline, linestyle="--", color='grey')
        p1.text(460, 460, s='m='+ f'{m1:.2f}', color='grey')

        #Bild rotieren
        p2.title.set_text('Begradigung')
        p2.axis('off')
        alphaInRad = math.atan((m1 + m2) / 2);
        alphaInDeg = alphaInRad * 180 / math.pi;
        rotated = rotate(masked, alphaInDeg, center=None)
        p2.imshow(rotated, cmap='gray')

        #Die vier Eckpunkte rotieren
        p3.title.set_text('Koordinatenanpassung')
        p3.axis('off')
        R = np.array([[math.cos(alphaInRad), math.sin(alphaInRad)],
                      [-math.sin(alphaInRad), math.cos(alphaInRad)]])
```

```
origin = np.array([[320], [240]])

x_corrected = np.zeros(4)
y_corrected = np.zeros(4)

for k in range(0, 4):
    vertex = np.array([[x[k]],[ y[k]]])
    vertex = R.dot(vertex - origin) + origin
    x_corrected[k] = vertex[0]
    y_corrected[k] = vertex[1]
p3.imshow(rotated, cmap='gray')
p3.plot(x, y, '+r', markersize=10, label='Ursprüngliche Koordinaten')
p3.plot(x_corrected, y_corrected, '+',color='#42FA25', markersize=10,
        label='Koordinaten nach Transformation')
p3.legend(loc=8)

#Gerade ermittlen und in zweitem Plot einzeichnen
m3 = (y_corrected[0] - y_corrected[3]) / (x_corrected[0] - x_corrected[3]);
b2 = y_corrected[0] - m3 * x_corrected[0]
yend2 = m3 * 640 + b2
xline2 = [0, 639]
yline2 = [b2, yend2]
p2.plot( xline, yline, linestyle="--", color='grey', label='Vor Rotation')
p2.plot( xline2, yline2, linestyle="--", color='#42FA25', label='Nach Rotation')
p2.legend(loc=8)
p2.text(460, 460, s='m='+ f'{m3:.2f}', color='#42FA25')
plt.subplots_adjust(wspace = 0.03)
plt.show()
```



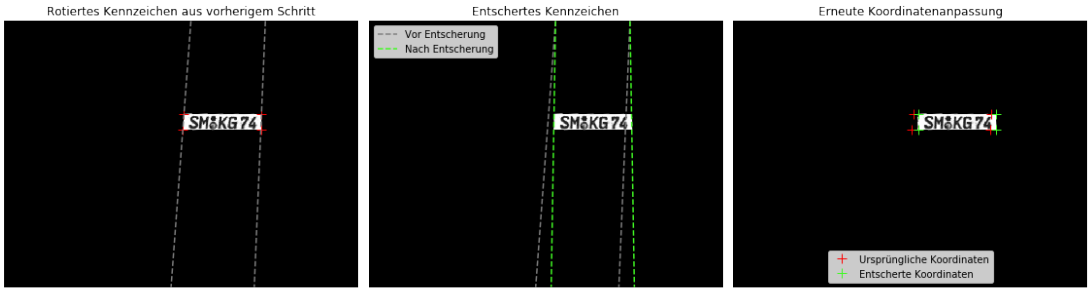
6 Entscherung

Nun folgt noch die Entscherung der Seiten des Kennzeichens. Eine weitere Rotation würde uns nicht weiterhelfen, das sie die Ober- und Unterkante wieder verdrehen würde, somit müssen wir auf eine zweidimensionale affine Bildtransformation zurückgreifen. Dafür erstellen wir eine Scherungsmatrix **M** und geben diese an den Methodenaufruf *warpAffine()* von OpenCV weiter. Da der Aufruf eine dreidimensionale Scherungsmatrix erwartet, haben wir die Scherungsmatrix für unseren zweidimensionalen Fall in der dritten Dimension genullt.

Wie auch bei der Begradigung müssen wir dafür sorgen, dass unsere Polygon-Eckpunkte ebenfalls an die Entscherung angepasst werden. Da uns hier die Entscherung der Eckpunkte mit einer zweidimensionalen Scherungsmatrix einfacher fällt, erstellen wir diese hier unter dem Namen **R**. Das Skalarprodukt der Matrix und der Eckpunkte führt dann die Entscherung aus.

```
In [6]: fig, (p1, p2, p3) = plt.subplots(1, 3, figsize=(20,20))

        p1.title.set_text('Rotiertes Kennzeichen aus vorherigem Schritt')
        p1.imshow(rotated, cmap='gray')
        p1.plot(x_corrected, y_corrected, '+r', markersize=10,
        label='Ursprüngliche Koordinaten')
        p1.axis('off')

        #Rechte gerade
        mRight = (y_corrected[3] - y_corrected[2]) / (x_corrected[3] - x_corrected[2]);
        bRight = y_corrected[2] - mRight * x_corrected[2]
        xEndRight = (480 - bRight) / mRight
        xRightLine = [-bRight/mRight, xEndRight]
        yRightLine = [0, 479]
        p1.plot( xRightLine, yRightLine, linestyle="--", color='gray', label='Alte Steigung')

        #Linke gerade
        mLeft = (y_corrected[0] - y_corrected[1]) / (x_corrected[0] - x_corrected[1]);
        bLeft = y_corrected[0] - mLeft * x_corrected[0]
        xEndLeft = (480 - bLeft) / mLeft
        xLeftLine = [-bLeft/mLeft, xEndLeft]
        yLeftLine = [0, 479]
        p1.plot( xLeftLine, yLeftLine, linestyle="--", color='gray')

        #Entscherung des Bildes
        s1 = (x_corrected[0]-x_corrected[1]) / (y_corrected[0]-y_corrected[1])
        s2 = (x_corrected[2]-x_corrected[3]) / (y_corrected[2]-y_corrected[3])
        sy = (s1 + s2) / 2
        M = np.float32([[1, -sy, 0],
                        [0, 1, 0]])
        unskewed = cv2.warpAffine(rotated, M, (640, 480), None, flags=cv2.INTER_LINEAR,
                                  borderMode=cv2.BORDER_REFLECT_101 )

        #Entscherung der Eckpunkte
        R = np.array([[1, -sy], [0, 1]])
        x_unskewed = np.zeros(4)
        y_unskewed = np.zeros(4)
        for k in range(0,4):
            vertex = np.array([[x_corrected[k]],[ y_corrected[k]]])
            vertex = R.dot(vertex)
            x_unskewed[k] = vertex[0]
            y_unskewed[k] = vertex[1]
        p2.imshow(unskewed, cmap='gray')
```

```
#Rechte gerade
deltax = (x_unskewed[3] - x_unskewed[2]);
mRight2 = (y_unskewed[3] - y_unskewed[2]) / deltax if deltax != 0 else 0
bRight2 = y_unskewed[2] - mRight2 * x_unskewed[2]
xEndRight2 = (480 - bRight2) / mRight2 if mRight2 != 0 else 0
xRightLine2 = [(-bRight2 / mRight2 if mRight2 != 0 else 0), xEndRight2]
yRightLine2 = [0, 479]


#Linke gerade
mLeft2 = (y_unskewed[0] - y_unskewed[1])
        / (x_unskewed[0] - x_unskewed[1]) if (x_unskewed[0] - x_unskewed[1])
         !=0 else 0
bLeft2 = y_unskewed[0] - mLeft2 * x_unskewed[0]
xEndLeft2 = (480 - bLeft2) / mLeft2 if mLeft2 != 0 else 0
xLeftLine2 = [(-bLeft2/mLeft2) if mLeft2 != 0 else 0, xEndLeft2]
yLeftLine2 = [0, 479]

p2.plot( xRightLine, yRightLine, linestyle="--", color='gray',
        label='Vor Entscherung')
p2.plot( xLeftLine, yLeftLine, linestyle="--", color='gray')
p2.plot( xLeftLine2, yLeftLine2, linestyle="--", color='#42FA25')
p2.plot( xRightLine2, yRightLine2, linestyle="--", color='#42FA25',
        label='Nach Entscherung')
p2.legend(loc=0)
p2.axis('off')
p2.title.set_text('Entschertes Kennzeichen')

p3.axis('off')
p3.imshow(unskewed, cmap='gray')
p3.title.set_text('Erneute Koordinatenanpassung')
p3.plot(x_corrected, y_corrected, '+r', markersize=10,
         label='Ursprüngliche Koordinaten')
p3.plot(x_unskewed, y_unskewed, '+',color='#42FA25', markersize=10,
        label='Entscherte Koordinaten')
p3.legend(loc=8)
plt.subplots_adjust(wspace = 0.03)
plt.show()
```



7 Bild ausschneiden und reduzieren

Da unser Kennzeichen jetzt begradigt und entschert ist, können wir das mithilfe unserer Eckpunkte aus dem Gesamtbild ausschneiden. Danach konvertieren wir das Graufstufen bild in ein Binärbild. Für den dafür benötigten Schwellwert verwenden wir den Median des Bildes und korregieren mit **correctionFactor** etwas nach.

```
In [7]: fig, (p1, p2) = plt.subplots(1, 2, figsize=(20,20))
        x1 = int(round( (x_unskewed[0] + x_unskewed[1]) / 2)) + 4
        x2 = int(round( (x_unskewed[2] + x_unskewed[3]) / 2)) - 1
        y1 = int(round( (y_unskewed[0] + y_unskewed[3]) / 2)) + 3
        y2 = int(round( (y_unskewed[1] + y_unskewed[2]) / 2)) - 1
        cropped = unskewed[y1:y2, x1:x2]
        p1.imshow(cropped, cmap='gray')
        p1.axis('off')
        TH = np.median(cropped)
        correctionFactor = 0.1
        result = 1 - (cropped < TH - correctionFactor)
        p2.imshow(result, cmap='gray')
        p2.axis('off')
        plt.show()
```

8 Segmentierung

Im Schritt der Segmentierung wollen wir die Bounding Boxes der einzelnen Buchstaben und Ziffern in dem Bild finden und solche verwerfen die unseren Ausschlusskritieren entsprechen. Da diese Kriterien stark von der BildgröSSe abhängig sind, skalieren wir als erstes das Bild auf eine

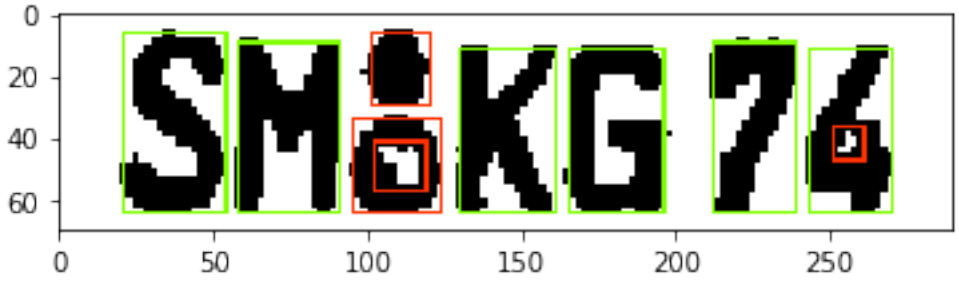
einheitliche GröSSe (280x60px). Dann fügen wir ein weiSSes 5px Padding um das Bild herum ein, damit die Segmentfindung besser arbeiten kann. AuSSerdem erzeugen wir eine Kopie des Bildes und wandeln sie in ein RGB-Bild um. Dieser Schritt ist nur für die Darstellung wichtig, da wir die Bounding Boxes grün und rot visualisieren wollen.

In einigen Kennzeichen fällt auf, dass Buchstaben leicht miteinander verschmelzen. Das ist für die Segmentierung sehr ungünstig, da wir Segmente finden, die mehr als ein Zeichen enthalten. OpenCV stellt uns hierfür die Methode *dilate()* bereit, welche alle schwarzen Bereiche schrumpfen lässt. Wir müssen hier aufpassen, den Effekt nicht zu übertreiben und die Buchstaben unkenntlich zu machen.

Um die Segmentierung durchzuführen, setzen wir die Methode *findCountours()* von OpenCV ein, welche uns eine Liste der gefundenen Bounding Boxes zurückliefert. Danach iterieren wir über diese Liste und entscheiden uns nach fein abgestimmten Kriterien bestimmte Segmente aufzunehmen oder abzulehnen. Die Kriterien hier sind die Fläche und die Höhe einer Kontur. Die Höhe ist ein gutes Kriterium um schwarze Flecken auszuschlieSSen, da Kennzeichen nur GroSSbuchstaben und Ziffern enthalten können. Die Höhe allein reicht aber nicht aus, da die Konturfindung dazu tendiert auch mal den Kennzeichenrahmen zu finden.

```
In [8]: img = result.astype(dtype='uint8') * 255
        img = cv2.resize(img, (280, 60), interpolation = cv2.INTER_NEAREST)

        padding = 5
        img = cv2.copyMakeBorder(img, padding, padding, padding, padding,
                cv2.BORDER_CONSTANT, value=(255, 255, 255))
        dst = np.zeros((img.shape[0], img.shape[1],3)).astype(dtype='uint8')
        dst[:,:,0] = img
        dst[:,:,1] = img
        dst[:,:,2] = img

        kernel = np.ones((3,3),np.uint8)
        img = cv2.dilate(img, kernel, iterations=1)

        ret, thresh = cv2.threshold(img, 1,150,1)

        im, contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
                cv2.CHAIN_APPROX_SIMPLE)
        rects = []
        a = 0
        for cnt in contours:
            x,y,w,h = cv2.boundingRect(cnt)
            area = cv2.contourArea(cnt)
            if area > 150 and area < 1500 and h > 45:
                cv2.rectangle(dst, (x,y), (x+w, y+h), (128, 255, 0), 1)
                rects.append([x, y, w, h])
                print('+Accept: Width: {}, Height: {}, Area: {}'.format(w, h, area))
            else:
                print('  -Reject: Width: {}, Height: {}, Area: {}'.format(w, h, area))
                cv2.rectangle(dst, (x,y), (x+w, y+h), (255, 50, 0), 1)
        plt.imshow(img)
```

```
        plt.show()
        plt.imshow(dst)
        plt.show()

  -Reject: Width: 29, Height: 30, Area: 599.0
  -Reject: Width: 17, Height: 16, Area: 183.5
+Accept: Width: 27, Height: 53, Area: 673.5
  -Reject: Width: 10, Height: 11, Area: 68.5
+Accept: Width: 31, Height: 53, Area: 766.5
+Accept: Width: 31, Height: 53, Area: 635.0
+Accept: Width: 27, Height: 55, Area: 569.5
+Accept: Width: 33, Height: 55, Area: 865.0
  -Reject: Width: 19, Height: 23, Area: 285.0
+Accept: Width: 33, Height: 58, Area: 722.5
```



9 Extraktion der Segmente

Als letzten Schritt müssen wir nur noch die akzeptablen Konturen aus dem Bild extrahieren und auf eine einheitliche GröSSe bringen. Da die Reihenfolge der Konturen unvorhersehbar ist, müssen wir die gefundenen Konturen erst nach ihrer x-Koordinate sortieren. Dann fügen wir ein Padding ein, so dass alle Zeichen zentriert im Segment liegen. AbschlieSSend korrigieren wir noch eventuelle GröSSenabweichhungen, indem wir auf eine einheitliche GröSSe skalieren (32x50px).

In [9]:
```
#Sort by x-value
rects = np.array(rects)
count = rects.shape[0]
fig, arr = plt.subplots(1, count, figsize=(15,15))
rects = sorted(rects,key=lambda x: x[0])
cutouts = []
for i in range(0, count):
    (x,y,w,h) = rects[i]
    c = img[y-1:h+y+1, x-1:x+w+1]
    hpadding = int(35 - h / 2)
    wpadding = int(25 - w / 2)
    c = cv2.copyMakeBorder(c, hpadding, hpadding, wpadding, wpadding,
        cv2.BORDER_CONSTANT, value=(255, 255, 255))
    c = cv2.resize(c, (32, 50), interpolation = cv2.INTER_NEAREST)
    cutouts.append(c)
    arr[i].axis('off')
    arr[i].title.set_text('('+str(c.shape[0])+', '+str(c.shape[1])+')')
    arr[i].imshow(c)
plt.subplots_adjust(wspace = 0.5)
plt.show()
```