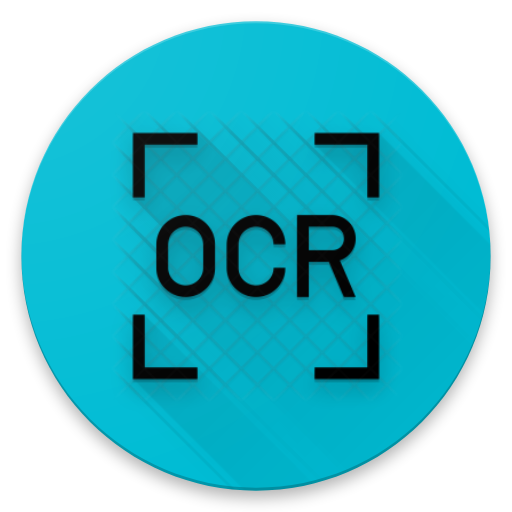
Computer Vision

Character Recognition



Datum: 26-03-2021

Opdrachtgever: Diederik Yamamoto Roijers

Door: Joris Heemskerk

Inhoudsopgave

[Inleiding 3](#_Toc67732317)

[Onderzoek 4](#_Toc67732318)

[Canny 6](#_Toc67732319)

[Contrast stretching 7](#_Toc67732320)

[Sobel 8](#_Toc67732321)

[Resolutie 10](#_Toc67732322)

[SVM 10](#_Toc67732323)

[Resultaten 12](#_Toc67732324)

[Conclusie 13](#_Toc67732325)

[Discussie 14](#_Toc67732326)

[Bronnenlijst 15](#_Toc67732327)

# Inleiding

Voor het vak Beeldverwerking en Computer Vision kregen we de keuze voor onze eindopdracht. We konden een paper met een computer vision of beeldverwerking taak kiezen en deze proberen te reproduceren of we konden een uitdaging pakken/bedenken passend bij een computer vision of beeldverwerking taak.

Ik had in eerst instantie een paper gezocht (bron 1). Echter bleek uit nader overleg met Diederik dat deze paper vrij slecht gedocumenteerd heeft hoe de methodes uitgevoerd worden. Mede hierdoor werd mijn aangeraden de challenge opdracht te kiezen en te kijken of ik de door de paper gebruikte implementaties kon reproduceren, maar als dit niet zou lukken zou ik ook een eigen invulling eraan kunnen geven.

Gedurende de lesweken ben ik erg druk geweest met een herkansing die ik nog open had staan, met de andere vakken en met de weekopdrachten voor Beeldverwerking en Computer Vision. Hierdoor is het mij dus niet gelukt in de lesweken veel tijd te besteden aan mijn eindopdracht. Helaas betekent dit dat ik niet alles heb kunnen implementeren wat ik zou willen en dat ik mijn plan drastisch bij heb moeten stellen.

Hoewel dit niet ideaal is ben ik nog steeds blij met de resultaten.

Ik heb een vrij simplistische versie van character recognition toe gepast. Mijn implementatie berust op verschillende soorten edge detection die ik vervolgens los laat op een SVM.

Voor de realisatie hiervan heb ik gebruik gemaakt van een dataset (bron 2) die afbeeldingen bevat van letters die bijvoorbeeld uit een tijdschrift zijn geknipt of op een bord staan (zie hieronder).



De dataset bevat ook nog afbeeldingen van characters die met de hand geschreven zijn en die computer gegenereerd zijn. Deze datasets laat ik achterwegen. Ook de characters uit het Kannada (een Indiase taal) laat ik achterwegen.

De dataset bestaat uit 12503 unieke characters. Deze characters zijn alle 26 letters als kleine letters en hoofletters plus de getallen 0 tot en met 9. Dit samen resulteert in 62 verschillende klassen/categorieën.

# Onderzoek

In dit hoofdstuk behandel ik het verloop van mijn implementatie en de keuzes en ontdekkingen die ik tussendoor heb gemaakt.

In het begin was ik van plan twee a 3 van de methodes uit de paper te reproduceren. Echter heb ik per methode (Shape Contexts, Scale Invarient Feature Transform en Maximum Response of filters) een tijdje onderzoek gedaan naar hoe het werkt en het was toch iets te complex. Na een gesprek met Diederik bleek dat de methodes per stuk al behoorlijk veel vragen (een deel van deze bronnen staan onderaan het document, echter heb ik ze tussendoor helaas niet genoteerd en staat niet alles er bij). Het werd aangeraden om bijvoorbeeld te beginnen met een vrij standaard en simpele edge detection en om deze in een SVM zetten voor de predictions.

Dit is dan ook precies waar ik mee aan de slag ben gegaan.  
Het eerste obstakel wat ik tegenkwam was de dataset. Deze was vrij uitgebreid en ik wist niet goed hoe ik het uitlezen van de bestanden in de gegeven structuur goed aan kon pakken. Na wat zoeken bleek op de site van de dataset (bron 2) een bestand te staan met allemaal handige informatie. Er staat een lists.tgz waarin een aantal .mat bestanden staan die ik met python uit kan lezen. Ook staat er welke data er in dit bestand zit:

* ALLlabels: class label for each sample
* ALLnames: sub-directory and name of the image for each sample
* classlabels: set of labels (classes) in this dataset, coded numerically, e.g. 10=A, 11=B, ..., 64=z
* classnames: scrings of the directories where samples of each class are stored
* NUMclasses: total number of classes in this dataset
* TRNind: indexes of the training samples. If 20 splits are used, this is a matrix of N\_train\_samples X 20
* TSTind: indexes of the test samples. If 20 splits are used, this is a matrix of N\_test\_samples X 20
* VALind: indexes of the validation samples. If 20 splits are used, this is a matrix of N\_validation\_samples X 20
* TXNind: indexes of the texton samples, i.e., samples used to build the vocabulary with the bag-of-visual-words method. If 20 splits are used, this is a matrix of N\_texton\_samples X 20

Na een analyse van de lists.mat (die ik samen met Rick Horeman heb uitgevoerd) bleek deze volgorde niet helemaal te kloppen en bleek er ook een extra lijst te zijn. De volgende indexes van de door mij gebruikte list.mat (Lists/English/Img/lists.mat) komen overeen met de volgende punten van de hier boven genoemde lijst:

0 = ALLnames

1 = lijkt niet aanwezig te zijn in bovenstaande lijst (het lijkt niet nodig voor mijn onderzoek)

2 = ALLlabels

3 = classlabels

4 = classnames

5 = NUMclasses

6 = TSTind

7 = VALind

8 en 9 = TRNind en TXNind (deze indexlijsten zijn in deze list.mat hetzelfde)

Nu is het tijd om de data uit de dataset uit te lezen. Hiervoor heb ik het volgende stuk code gebruikt:

mat = scipy.io.loadmat('Lists/English/Img/lists.mat')

all\_names = mat['list'][0][0][0]

imgSize = 50

all\_img = np.empty((len(all\_names), imgSize, imgSize, 3))

# make and save all images into all\_img.npy

for i in range(len(all\_names)):

    filename = os.path.join('Data/Img/' + all\_names[i] + '.png')

    all\_img[i] =  resize(skimage.io.imread(filename), (imgSize, imgSize, 3))

Om te beginnen lees ik hier de lists.mat uit. Hieruit haal ik vervolgens de lijst op index 0, waarin de data van alle bestanden staan. In de for loop lees ik alle bestanden uit en deze sla ik op in all\_img in het formaat N bij N bij 3. Dit resulteert dus in een variabele all\_img waar al mijn afbeeldingen in zitten op de volgorde uit de lists.mat.

Nu wil ik al mijn afbeeldingen scheiden in twee nieuwe lijsten; één voor de test afbeeldingen en één voor de train afbeeldingen. Gelukkig heeft de lists.mat hier een handige lijst van indexen voor. Met name TSTind en TRNind.

test\_indexes = mat['list'][0][0][6]

test\_img = np.empty((len(test\_indexes) \* len(test\_indexes[0]),

imgSize, imgSize, 3))

test\_labels = np.empty(len(test\_indexes) \* len(test\_indexes[0]))

iterator = 0

for i in range(len(test\_indexes)):

    for j in range(len(test\_indexes[i])):

        if test\_indexes[i][j] != 0:

            test\_img[iterator] = all\_img[test\_indexes[i][j]-1]

            test\_labels[iterator] = all\_labels[test\_indexes[i][j]-1]

            iterator += 1

test\_img = test\_img[:iterator]

test\_labels = test\_labels[:iterator]

Met deze code filter ik door al mijn images heen en pak ik degene die overeen komen met de TSTind. Echter is het belangrijk dat ik de index 0 oversla. De TSTind gebruikt namelijk behoorlijk vaak een 0, terwijl de indexing vanaf 1 gebeurt. Om te zorgen dat ik hier rekening mee hou, sla ik deze waardes dus over. Het betekent wel dat mijn inschatting voor de grootte van de numpy array niet klopt. Om dit op te lossen kap ik de array af op de zelf bijgehouden iterator.  
Na het loslaten van dit stukje code, en een vergelijkbaar stuk voor de training images, op mijn all\_img hield ik vier arrays over; een array mettraining images, een array met testing images en twee arrays met de passende labels.

Vervolgens is het essentieel om deze belangrijke arrays op te slaan. Ik heb hiervoor gebruik gemaakt van de .npy files waar scipy.io naar kan schrijven.

with open('npy/test\_img.npy', 'wb') as f:

    np.save(f, test\_img)

with open('npy/test\_labels.npy', 'wb') as f:

    np.save(f, test\_labels)

Ik sla de arrays dus op in bijbehorende bestanden zodat ik niet elke keer mijn data hoef te genereren voor elke test die ik run.

De lijst met train afbeeldingen bestaat uit 9300 bestanden, De lijst met test afbeeldingen bestaat uit 14880 bestanden.

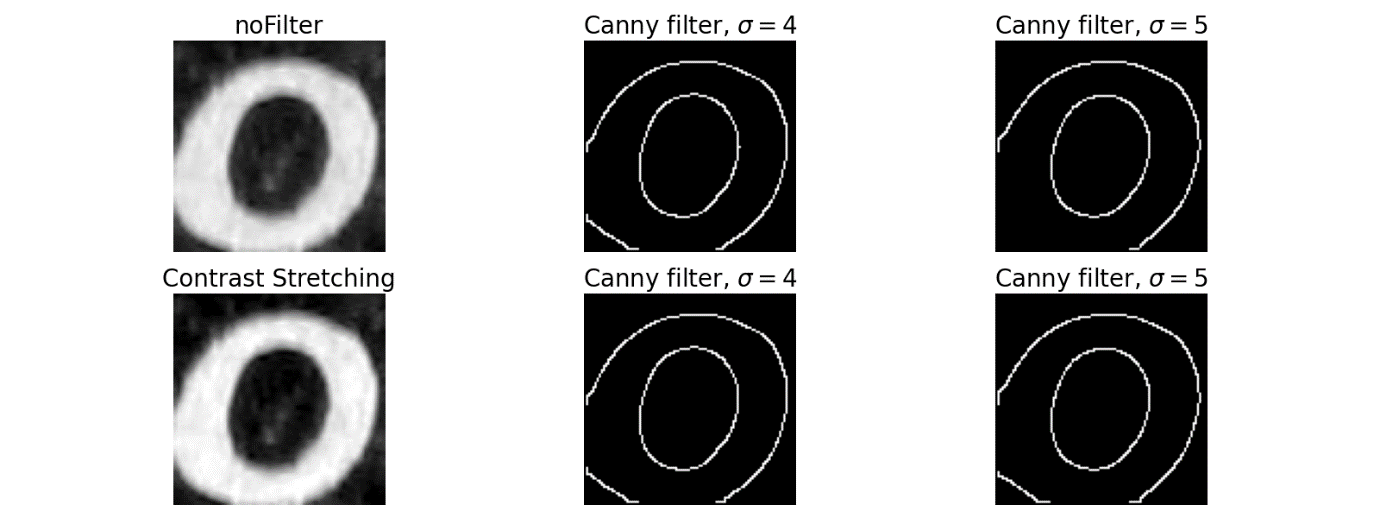
## Canny

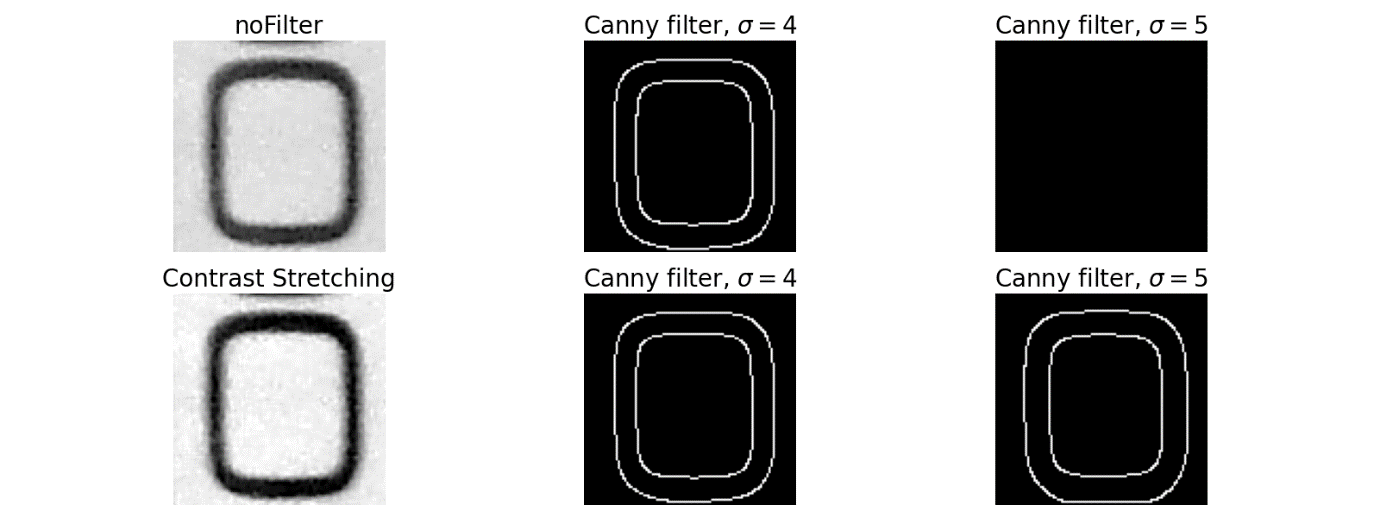
De eerste vorm van edge detection die ik toe ga passen is Canny edge detection. Deze vorm van edge detection werkt met Gaussian filters, gradient magnitudes en hysteresis om de randen van de afbeeldingen te herkennen (bron 3).

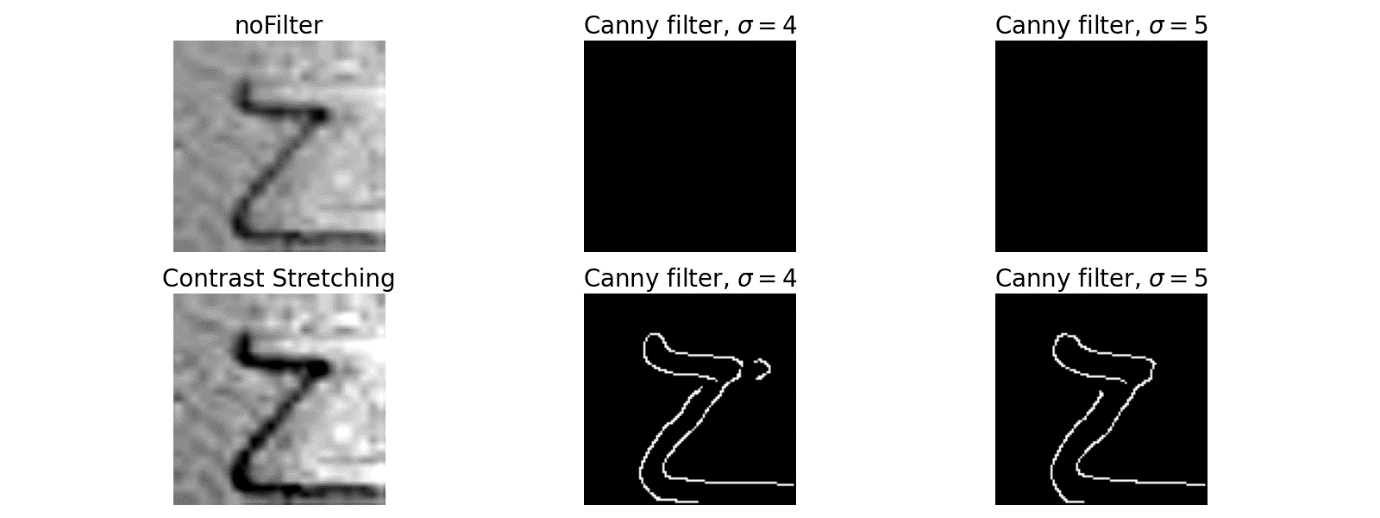
Voor de implementatie hiervan hoef ik echter niet echt te weten hoe Canny edge detection werkt, maar wel hoe ik dit moet implementeren. Gelukkig heeft skimage een “feature” import die dit kan.

filteredImg = feature.canny(img, sigma=n)

Hierin is de sigma vrij belangrijk, deze bepaald hoe intensief de filter is met het detecteren van de edges. Ik heb als aller eerst geprobeerd een afbeelding van 100\*100 pixels. Ik heb een aantal afbeeldingen gepakt en hier de filter overheen gezet met verschillende sigma’s.







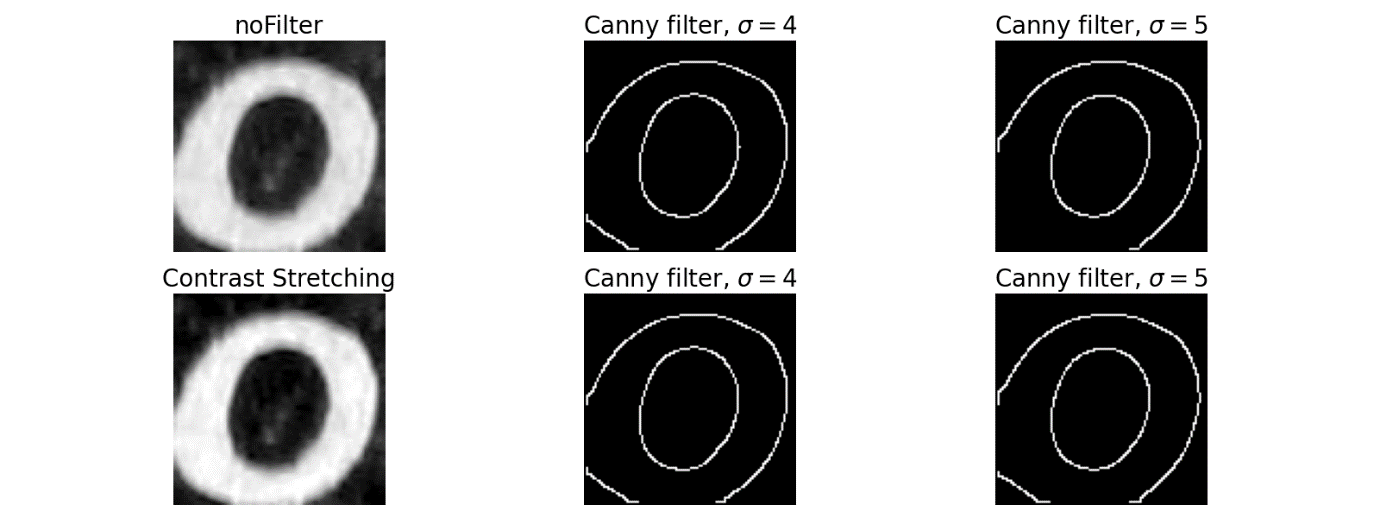
Een selectie van de tests die ik heb uitgevoerd

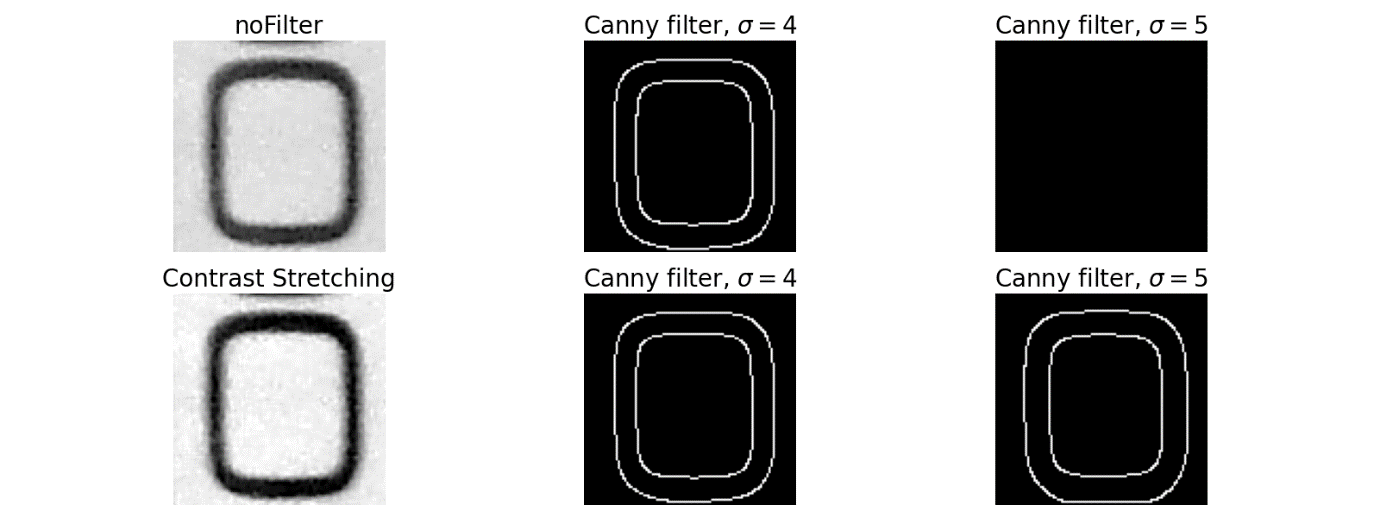
Bij de eerste image bleek een sigma van 3 het best te werken, echter bleek dit toch vrij veel ruis op te pakken van de achtergrond mee te pakken. Ik had een oplossing nodig die een groot deel van de ruis uit de afbeeldingen haalt, zodat ik hier een Canny filter overheen kan halen en betere resultaten krijg.

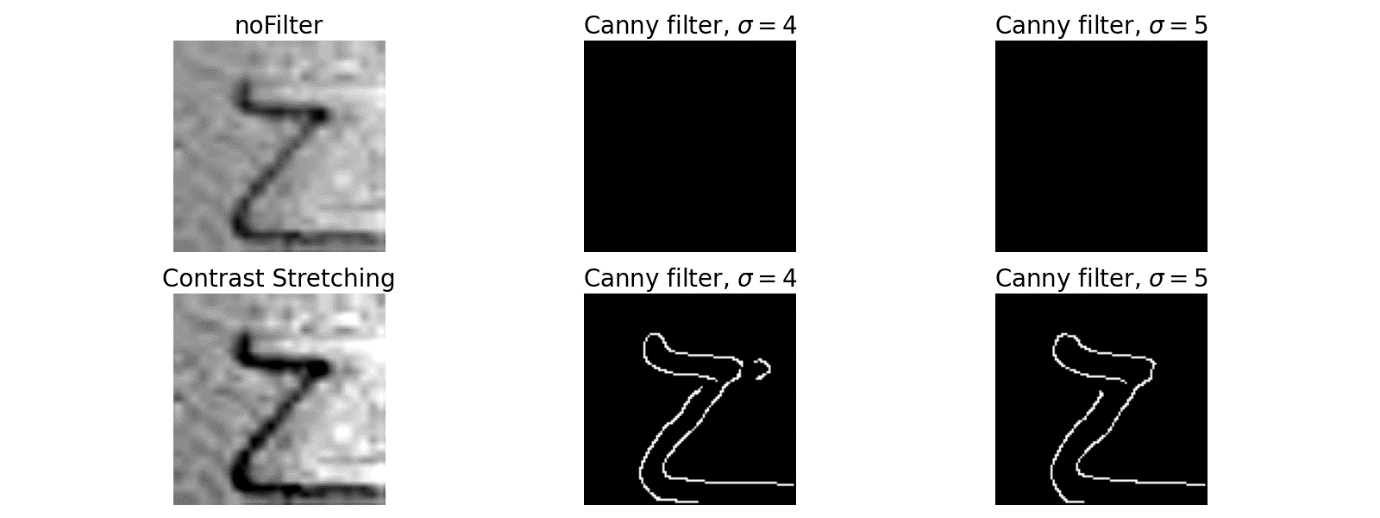
## Contrast stretching

Mijn oplossing voor dit probleem was contrast stretching (bron 4).

Door dit toe te passen zouden de lijnen wat duidelijker geaccentueerd kunnen worden. Dit helpt ook omdat ik de afbeelding zwart-wit moet maken en sommige transities van kleur in zwart wit niet duidelijk zijn.







Zoals in bovenstaande voorbeelden te zien is, zijn de characters ook daadwerkelijk veel beter herkenbaar met een Canny filter over de contrast stretching heen.

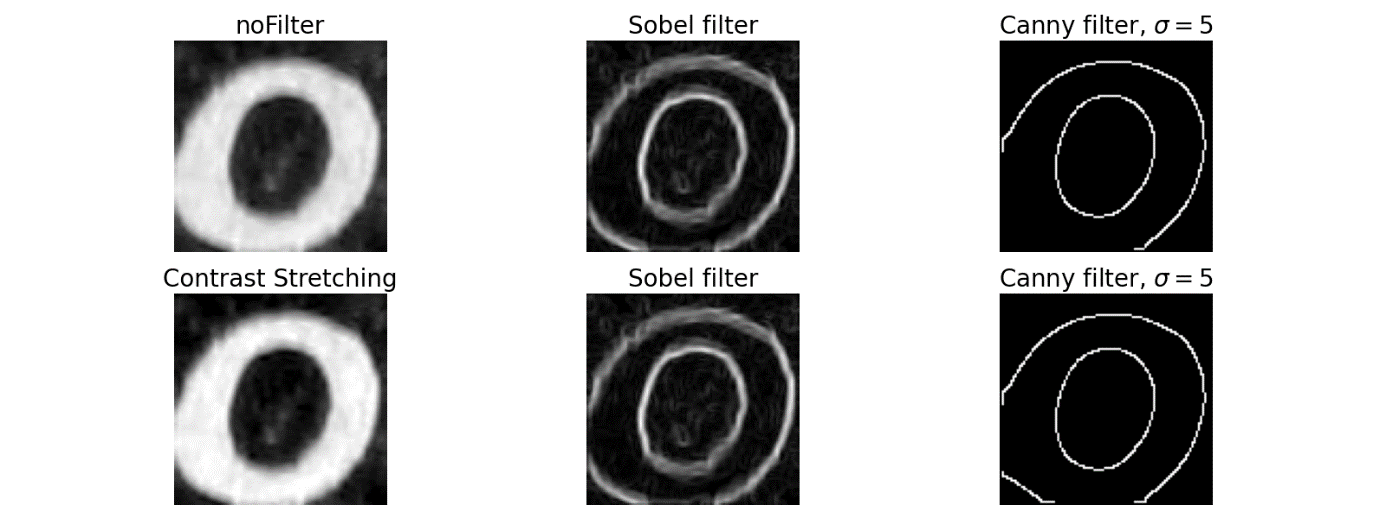
Op basis van deze resultaten heb ik besloten een sigma van 5 te gebruiken voor mijn Canny filter op een 100\*100 resolutie afbeelding.

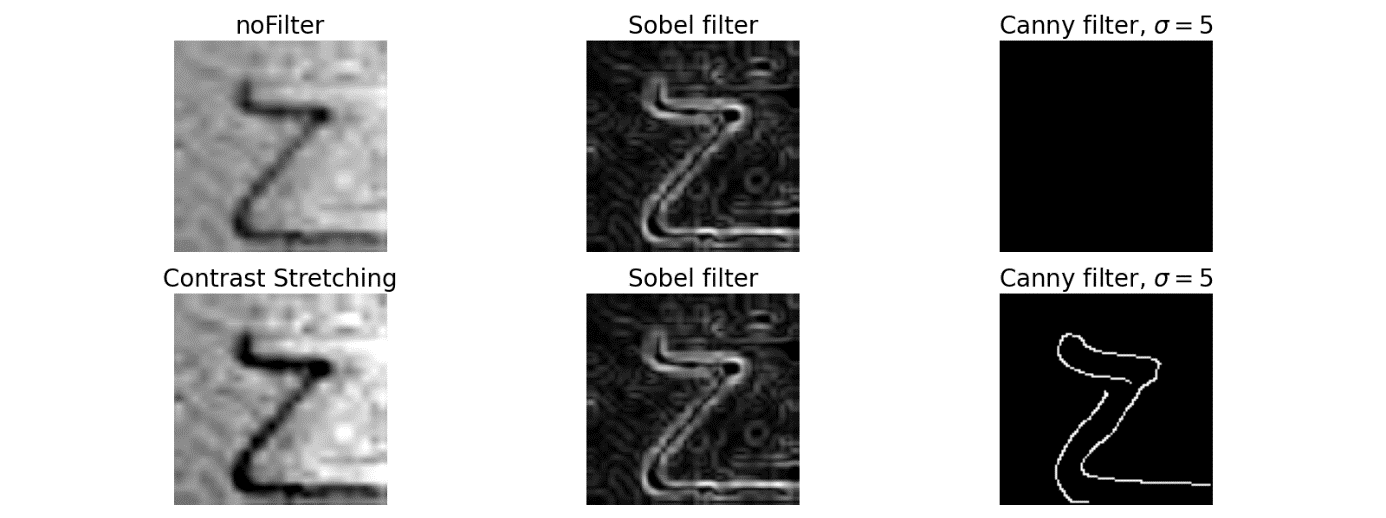
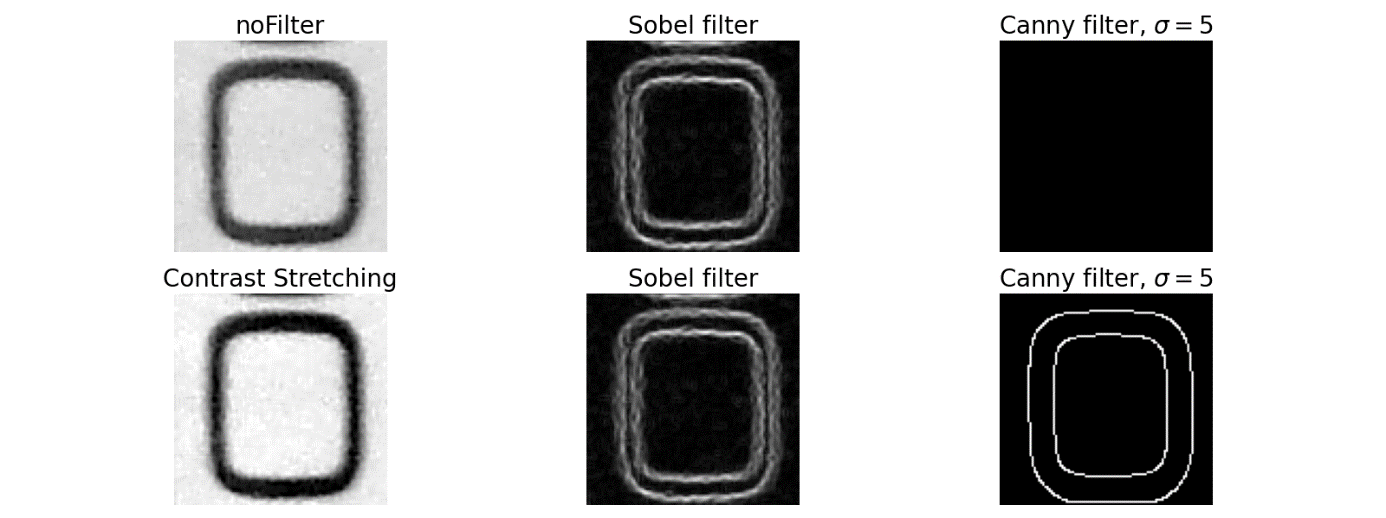
## Sobel

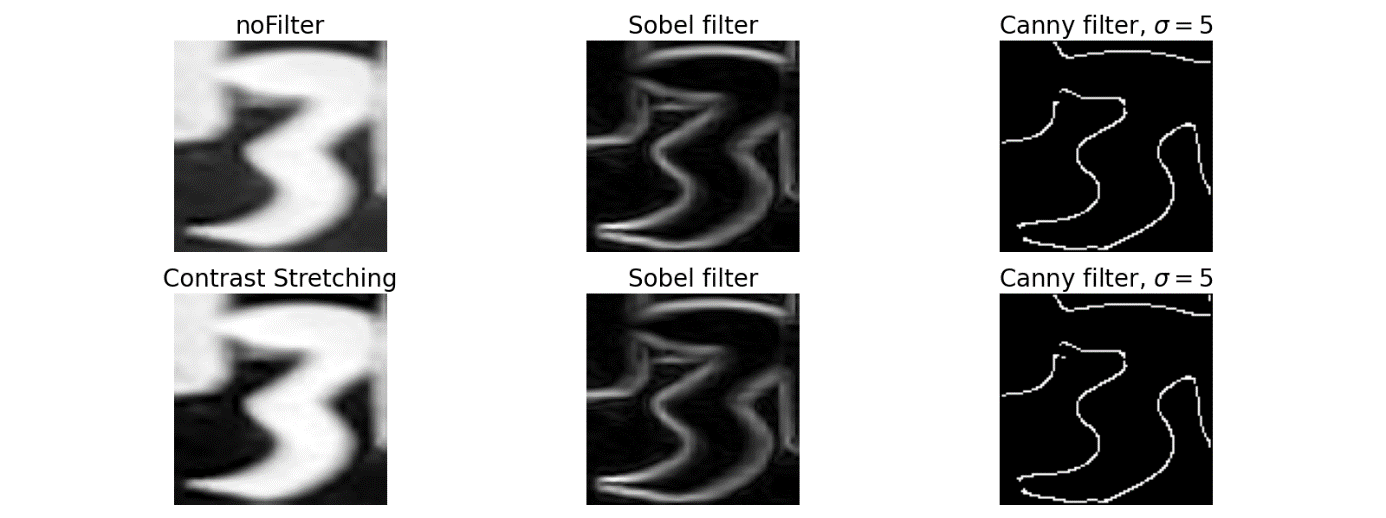
Om de accuraatheid van Canny edge detection te vergelijken met een andere vorm van edge detection ga ik gebruik maken van Sobel edge detection. Deze vorm van detectie gebruikt de combinatie van twee filters (bron 5). Om dit toe te passen heb ik de “filters” import van skimage gepakt om het volgend uit te kunnen voeren.

filteredImg  = filters.sobel(img)

De rest van de code blijft hetzelfde, de afbeeldingen blijven ook 100\*100.







Zoals te zien leveren de sobel filters iets minder scherpen lijnen op. Wat voor gevolg dit heeft op de resultaten weet ik nog niet. Mijn ogen kunnen echter wel zeggen dat de Canny filters beter te herkennen zijn.

## Resolutie

Alleen een vergelijk tussen een Canny en een Sobel filter vind ik niet genoeg. Ik wil graag ook kijken welke ander factoren ik nog kan veranderen. Een van deze factoren is de resolutie. Ik wil beginnen met afbeeldingen van 100 bij 100. Mijn gedachtegang hierachter is dat er meer informatie is en dat de computer hier meer mee kan.

Echter zou het ook kunnen dat een afbeelding van die grootte te veel informatie bevat, waardoor het differentiëren te makkelijk wordt. Om dit te kunnen testen wil ik ook lagere resoluties proberen.  
Ik heb gekozen voor halveringen, beginnend bij 100. Dit betekent dat ik resoluties van 100 bij 100, 50 bij 50, 25 bij 25, 13 bij 13, 6 bij 6 en 3 bij 3 ga gebruiken.

## SVM

Het toepassen van deze filters is allemaal leuk en aardig, maar dit is nog geen computer vision. Ik ben nu op het punt gekomen waar ik de images ook daadwerkelijk moet gaan gebruiken om voorspellingen te doen. Ik gebruik hiervoor de SVM van sklearn.

clf = svm.SVC(gamma=0.001, C=100)

clf.fit(train\_img\_edges, train\_labels)

In deze clf.fit stop ik de training images die ik eerder heb aangemaakt, waar ik eerst uiteraard de eerder gesproken filters overheen zet, en de labels (de getallen die de computer probeert te gokken) die erbij horen.

Vervolgens sla ik de CLF op, zodat ik hem niet elke keer als ik een test wil draaien aan hoef te maken. Hiervoor gebruik ik de “dump” import van joblib.

dump(clf, 'npy/clfSobel.joblib')

Vervolgens kan ik dit in een ander bestand weer uitlezen en hier mijn test afbeeldingen voorspellen met de clf.predict functie.

clf.predict(test\_img\_edges[i:i+1])

Door al de test afbeeldingen door de predict te halen en deze telkens te vergelijken met het label kan ik bepalen hoe accuraat mijn systeem is.

print("\n{} out of {} images are correctly predicted".format(correct,

len(test\_img\_edges)))

print("This is {}%\n".format(correct/len(test\_img\_edges)\*100))

Met bovenstaande code komt de nauwkeurigheid in dit format in mijn terminal terecht:

264 out of 14880 images are correctly predicted

This is 21.935483870967744%

Resultaten

Zoals in mijn onderzoek omschreven heb ik voor verschillende resoluties beide filters toegepast. Echter heb ik wel hier en daar de sigma aangepast, voor betere resultaten.

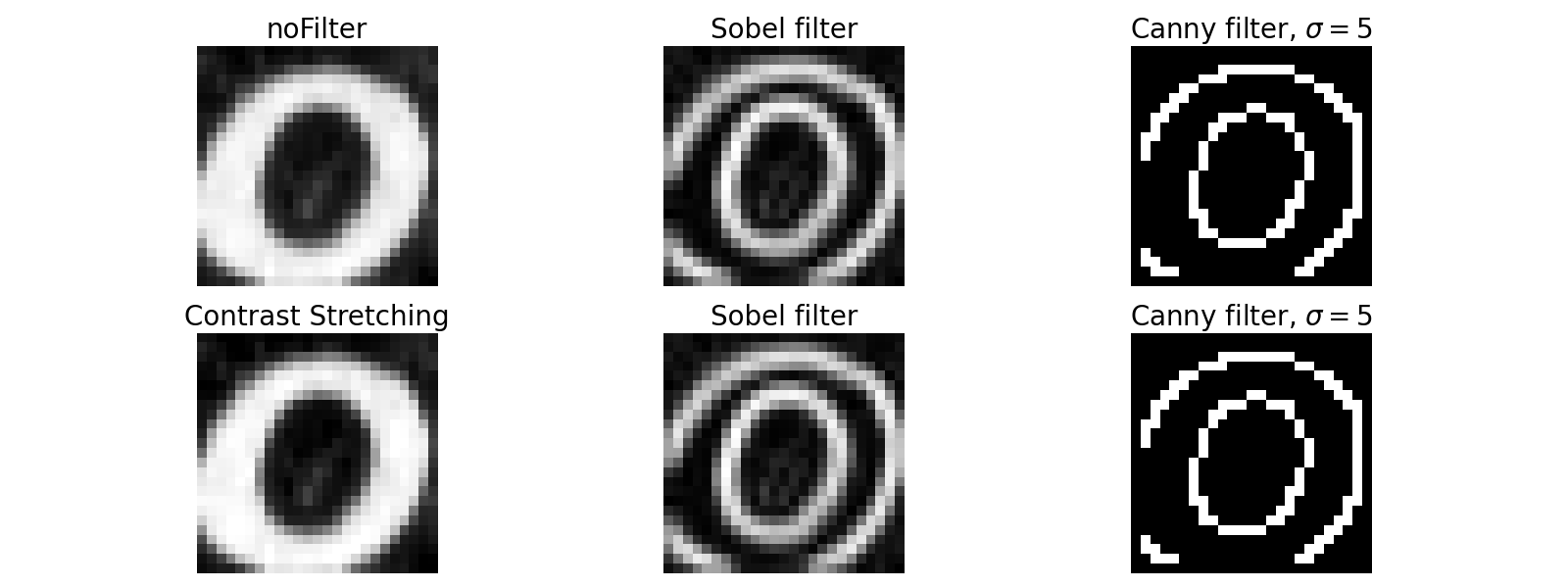
Het is ook belangrijk aan te geven dat, met willekeurig gokken, je een nauwkeurigheid van 1 op de 62 verwacht. Dat is 1.61%.

|  |  |  |
| --- | --- | --- |
| Resolutie | Canny | Sobel |
| 100\*100 | 14.73 % (sigma 5) | 21.72 % |
| 50\*50 | 14.52 % (sigma 5) | 24.19 % |
| 25\*25 | 20.97 % (sigma 1.5) | 28.82 % |
| 13\*13 | 20.75 % (sigma 1.5) | 30.43 % |
| 6\*6 | 7.63 % (sigma 1.5) | 31.51 % |
| 3\*3 | 1.40 % (sigma 1.5) | 11.29 % |
|  | 11.50 % (sigma 0) |  |

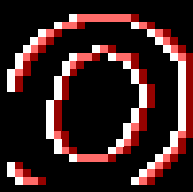
Op basis van deze resultaten leek het mij ook interessant om een combinatie te proberen. Ik heb een test gedaan met een resolutie van 25 bij 25. Ik heb de op de train afbeeldingen een sobel filter toegepast en op de test afbeeldingen heb ik een canny filter toegepast. Het resultaat hiervoor was 21.72%. Dit is iets beter dan alleen de canny filter, maar slechter dan alleen de sobel filter.

Conclusie

Op basis van de resultaten concludeer ik dat het Sobel filter op alle vlakken iets beter is dan de Canny filter. De reden die ik hierachter zie is dat het Sobel filter meer ruimte geeft voor fouten. Bij een kleine nuance tussen de verschillende soorten van dezelfde filters.



Zo kun je duidelijk zien dat je bij de Canny filter al een heel andere afbeelding hebt wanneer alles één pixel opzij geschoven wordt terwijl dit bij de Sobel filter erg meevalt, omdat er nog steeds veel overlap is tussen de witte stukken.

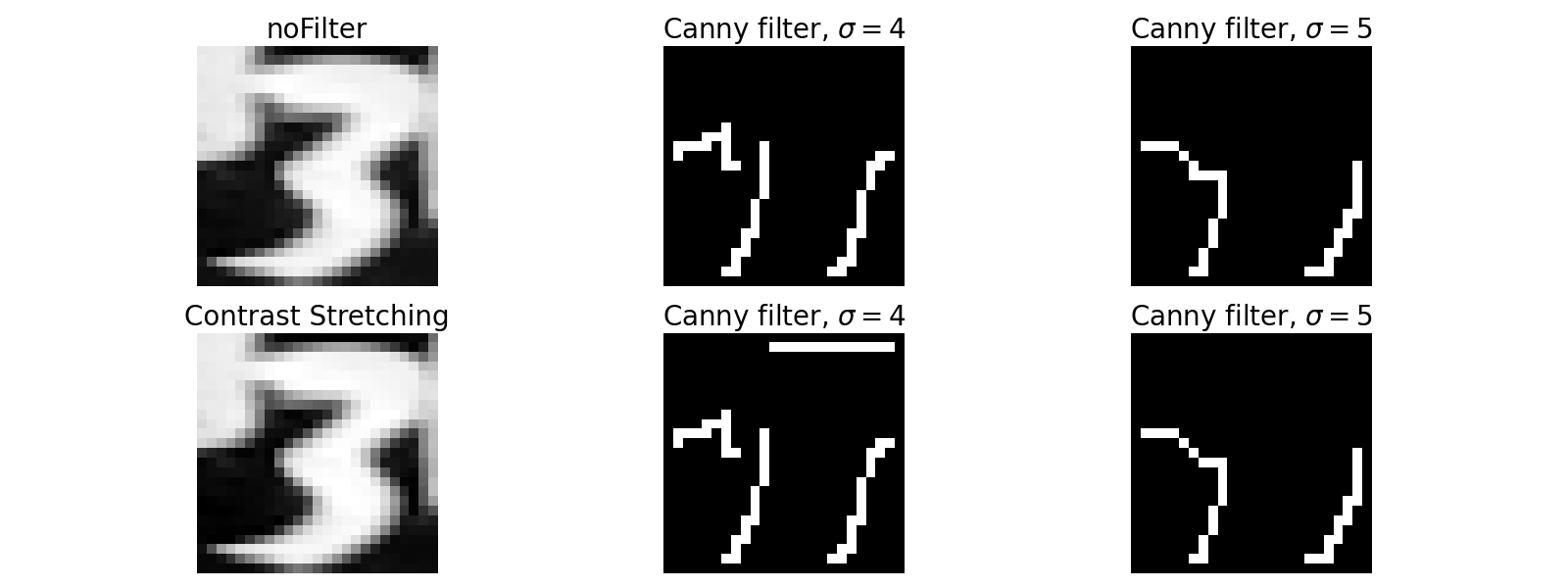


In conclusie is, voor de technieken die ik heb toegepast, de Sobel filter dus het best. Voor de optimale resultaten is het gebruiken van een zwart wit afbeelding van 13 bij 13 resolutie waar eerst contrast stretching op wordt toegepast ideaal.

# Discussie

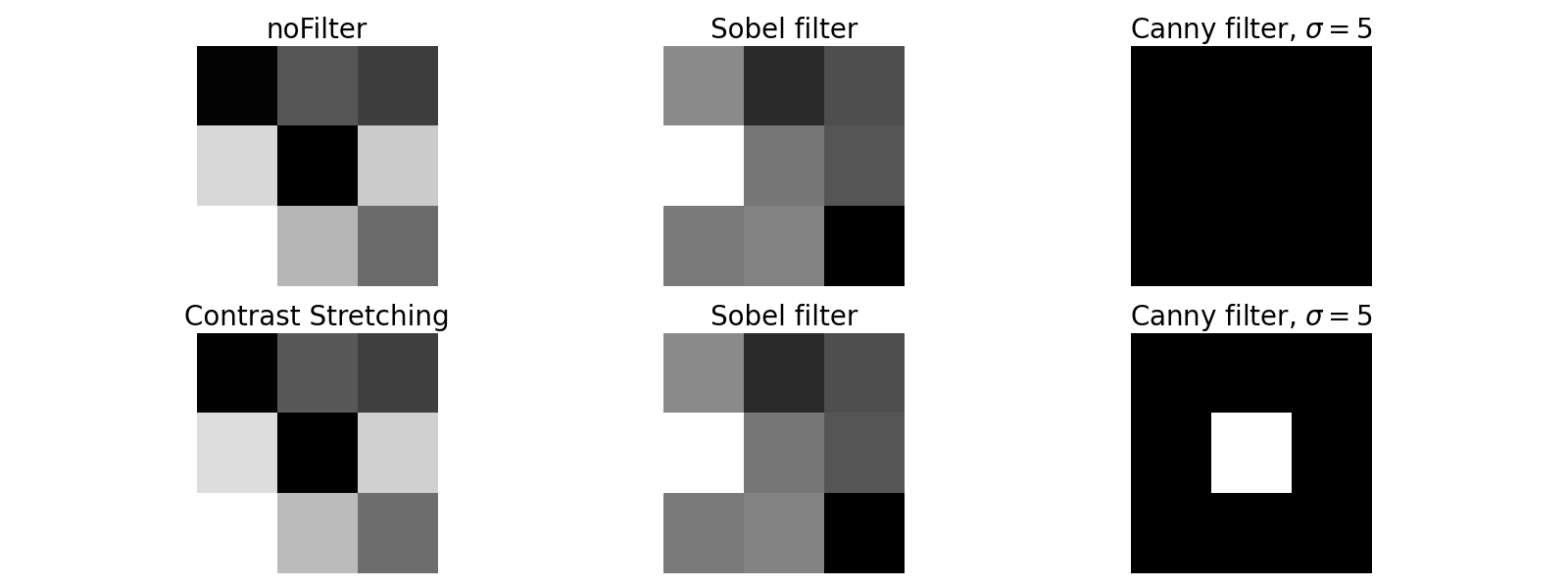
Zoals met elk onderzoek zijn er punten die beter of anders kunnen. Dit onderzoek is geen uitzondering. Hieronder een aantal punten die anders hadden gekund of die de resultaten beïnvloeden.

* De gebruikte dataset bevat twee folders, een “GoodImg” en een “BadImag”. De door mij toegepaste methodes werken uiteraard beter op de goede afbeeldingen dan op de slechte. Echter was het differentiëren tussen de verschillende instanties voor mij te lastig om te doen binnen de beperkte tijd die ik had. Dit is wel een manier om de nauwkeurigheid waarschijnlijk drastisch te verhogen.



Een voorbeeld van een bad image

* Het gebruiken van een dataset met afbeeldingen die vooraf al zwart wit zijn zou fouten bij het vertalen van de afbeelding kunnen voorkomen. Denk hierbij aan een set van computer gegenereerde letters of een handschrift van zwarte letters op wit papier. Kleur vertalen naar zwart wit kan namelijk een letter behoorlijk wegvagen wanneer de kleuren van de letter en de achtergrond te dicht bij elkaar liggen.
* De kans is aanwezig dat er nog een betere edge detection methode is die niet door mij is onderzocht die misschien betere resultaten met zich mee brengt.
* Het gebruik van de Canny filter is misschien niet eerlijk als het aankomt op lage resolutie afbeeldingen, omdat hier bijna geen edges zijn om te detecteren met deze methode. Zie hieronder.



(de laatste kolom hoort sigma = 0 te zin)

# Bronnenlijst

1. de Campos, T. E., Babu, B. R., & Varma, M. (2009). CHARACTER RECOGNITION IN NATURAL IMAGES. VISAPP. <https://www.scitepress.org/Papers/2009/17701/17701.pdf>
2. de Campos, T. E., & Varma, M. (2012, October 15). The Chars74K image dataset - Character Recognition in Natural Images. CVSSP. <http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/>
3. Wikipedia contributors. (2021, February 16). Canny edge detector. Wikipedia. <https://en.wikipedia.org/wiki/Canny_edge_detector>
4. Scikit-image development team. (n.d.). Histogram Equalization — skimage v0.19.0.dev0 docs. Scikit-Image. Retrieved March 26, 2021, from <https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_equalize.html>
5. Wikipedia contributors. (2021b, February 21). Sobel operator. Wikipedia. <https://en.wikipedia.org/wiki/Sobel_operator\>
6. Belongie, S., Malik, J., and Puzicha, J. (2002). Shape matching and object recognition using shape contexts. IEEE Transactions on Pattern Analysis and Machine Intelligence.
7. D. Gravrila and V. Philomin, "Real-Time Object Detection for Smart Vehicles," Proc. Seventh Int’l. Conf. Computer Vision, pp.87-93,1999.
8. S. Belongie, J. Malik, and J. Puzicha, “Matching Shapes,” Proc. Eight int’l. Conf. Computer Vision, pp. 454-461, July 2001