

Skanner for kvitteringer

Sivert M. Skarning

Mars 2019

1 Introduksjon

Som et prosjekt i faget bildebehandling og mønstergjenkjenning ved Høgskolen i Østfold har jeg valgt å utvikle en digital kvitteringskanner. En kvitteringskanner er et program som ved hjelp av et kamera klarer å konvertere en kvittering fra fysisk til digital form. I figur 1 ser vi et utdrag fra en kvittering.



Figure 1: Eksempel på en kvittering fra en svensk butikk

2 Oppgave

I dette prosjektet vil vi kun se på teknologien bak analysen av bilde. Det vil ikke inneholde beskrivelse av hva som skjer før og etter denne analysen, som hvordan man får inn bilde som input eller hvordan teksten blir fremvist. Oppgaven blir dermed å få inn et bilde av en kvittering finne ut hva som har blitt kjøpt,

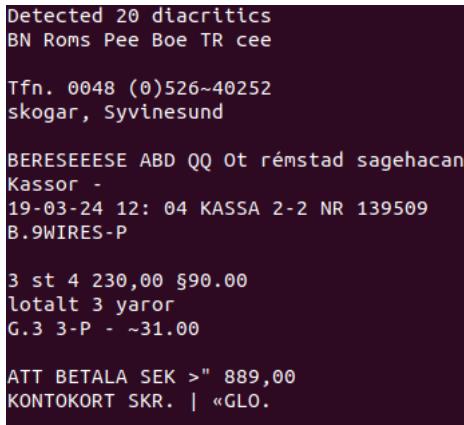
prisen, hvor produktet er blitt kjøpt, avslag og eventuelle skatter som har blitt betalt på produktet. Prosjektet kommer til å bruke kodebibliotek og eksisterende teknologier, men egen programkode skal styre retting og behandling av bilde.

3 Teknologi

Det finnes mange teknologier innenfor bildebehandling og tekstgjenkjenning. Jeg vil i dette prosjektet fokusere på å velge det som er enklest og implementere og open-source programvare der det er tilgjengelig.

3.0.1 Tekstgjenkjenning

Tesseract er et open-source programvare som utfører tekstgjenkjenning. Den kan utfører tekstgjennkjenning på mange typer filformater. Den har også API'er som gjør at det er mulig å bruke det med Python og andre programmeringspråk¹. I figur 2 ser dere output som blir generert etter å ha skannet bilde i figur 3



The image shows the output of Tesseract OCR on a scanned receipt. The text is white on a dark background. It includes:

- Detected 20 diacritics
- BN Roms Pee Boe TR cee
- Tfn. 0048 (0)526~40252
- skogar, Syvinesund
- BERESEESE ABD QQ Ot rémstad sagehacan
- Kassor -
- 19-03-24 12: 04 KASSA 2-2 NR 139509
- B.9WIRES-P
- 3 st 4 230,00 §90,00
- totalt 3 yaror
- G.3 3-P - ~31.00
- ATT BETALA SEK >" 889,00
- KONTOKORT SKR. | «GLO.

Figure 2: Output generert av Tesseract

Deskewing For at bilde skal kunne bli analysert på riktig måte, er vi nødt til skalere, kroppe og rotere bilde. Rotering av bilde kan bli gjort med et konsept som kalles homography. Her kan man bruke referansepunkter på et annet bilde til å rette det. Skalering og kropping er det mulig å programmere selv, eller bruke innebygde funksjoner i scikit-image.²

¹<https://www.learnopencv.com/deep-learning-based-text-recognition-ocr-using-tesseract-and-opencv/>

²<https://www.learnopencv.com/image-alignment-feature-based-using-opencv-c-python/>

Oppskarpning Det kan være utfordrende for tekstgjenkjennings-algoritmer å kjenne igjen tekst. De fleste programmer som kjører tekstgjenkjenning kjører egne filtre for å optimalisere det for tekstgjenkjenning. For å få mer kontroll og øke sjansen for å at gjenkjenningen blir nøyaktig vil jeg bruke min egen optimerer. Den skal kunne gjøre bilde monokromt, fjerne støy, gjøre bilde skarpere og forbedre kontrasten.³ ⁴



Figure 3: Behandlet versjon av figur 1

4 Gjennomføring

4.1 Deskewing

Et problem med å gjennomføre tekstgjenkjenning på et bilde uten å ha behandlet det først er at man ikke har noen garanti på at bilde har riktig vinkel. Hvis teksten ikke går parallelt med bilde vil resultatet kunne bli dårligere. Som observert i figur 5 er bildet feil orientert. På dette bildet klarer ikke tesseract å gjenkjenne teksten. På bildet som er riktig orientert får vi følgende resultat som sett i figur 4. Dette resultatet er bedre, men fortsatt langt fra treffsikkert.

4.2 Hough

Teori Hough transformasjonen er brukt for å detektere former. Algoritmen har potensiale til å detektere forskjellige typer former, men i denne sammenhengen er den kun nyttig for å detektere linjer⁵. Hough transformasjonen baserer

³<https://docparser.com/blog/improve-ocr>

⁴<https://docparser.com/blog/improve-ocr-accuracy/>

⁵https://en.wikipedia.org/wiki/Hough_transform

```

BPYSOIK :PIS80d
YAN SZ1 000 196
ve NN ante Der PG MA NE

LH0X
JaJ8A 7 UNS

pr 139nGeg

1 20 oBueu 114008 BNDRUOG

Sy - SLI :14PSUY

g500679t :1998

gereZerst :SUPII
LL :67:560 610260 90

: JUB I

uap|eH ZGA| USUNSY

P[OL1S0 | Japeud SIES] USNNIS
x BULJØLAYSLG *

overs 19206 varnS set Bl EP VAA DOP OG MAE EY sten er

```

Figure 4: Resultate av tesseract OCR

seg på at en linje kan bli representert matematisk som

$$y = m * x + b$$

eller

$$r = x * \cos(\theta) + y * \sin(\theta)$$

Problemet med den første formen er at hvis vinkelen blir 90 grader i forhold til den horizontale linjen over bildet vil verdien m, eller stigningen m bli evig stor. Man vil derfor bruke en parameterisert form i polar-koordinater. Variablen r er lengden fra origo til et punkt P på linjen, theta er vinkelen mellom linjen OP og 'initial ray' eller X-aksen på bildet i dette tilfellet.

Med denne parameteriseringen kan vi finne alle linjer som oppfyller likningen per punkt på forgrunden av bildet. Da kan vi plotte alle verdiene for r og theta som en graf. Gjør man dette for alle forgrundspikslene i bildet, får vi en graf som viser når linjer krysser hverandre.

Man kan også beregne akkumulator-cellene. Her genererer man en matrise hvor kolonnen bestemmer theta og raden bestemmer r. For hver forgrund-



Figure 5: Bilde med feil rotering

spiksel vi har gått igjennom å beregnet hvilke verdier for theta og r som oppfyller likingen legger vi til verdien en i legger vi til 1 i akkumulatoren i punkt (r, θ) . Hver gang vi legger til 1 betyr dette at vi har funnet en ny piksel som skal bli lagt til på linjen. Dette betyr at vi sitter igjen med hvor mange piksler som ligger på en linje definert av r og θ i denne grafen. Dette kan være nyttig for å finne tydige linjer i bildet.

Implementasjon I et bilde av tekst eller i mitt eksempel et bilde av en kvittering, kan man detektere linjene som blir dannet av teksten.

Jeg valgte å bruke innebygde funksjoner for hough transformasjon i dette prosjektet. Scikit har funksjoner som *hough_transform* og *hough_transform_peak* som jeg brukte for selve hough transformasjonen.

Prinsippet er at de mest tydige linjene fra Hough transformasjonen vil bli valgt ut for seleksjon. I seleksjonen finner jeg hvilke linjer med omrent samme vinkel som forekommer flest ganger. Håpet er at disse linjene er linjene generert av teksten på kvitteringene. Vinkelen til linjene som forekommer flest ganger bestemmer derfor rotasjonen på bildet. Et eksempel på dette kan bli sett i figur 6. Her har algoritmen funnet flere forekomster av sterke linjer, hvilken vinkel med x-aksen som forekommer flest ganger og deretter rotert bilde med denne vinkelen. I figur 7 ser man listen counter som holder på hvilke vinkler i radianer som forekommer flest ganger. Et problem med denne algoritmen er at den er avhengig av klare linjer. Jeg har kjørt algoritmen på en del bilder og det hender at bildet blir rotert feil vei. Algoritmen tar heller ikke høyde for at bilde kan være opp ned eller rotert over 90 grader.

En løsning på dette ville være å prøve å rotere begge veier for så å sam-

menlikne tesseract resultatet med en ordbok automatisk for å sjekke hvilken rotasjon som var riktig.



Figure 6: Rotering gjort med Hough transform

```
Counter({-0.48: 4, -0.5: 3, -0.32: 3,
, 0.1: 1, 0.01: 1, -0.03: 1, 0.13: 1,
Angle: -27.501974166279513
-0.48
```

Figure 7: Utdrag fra deskewing.py

4.3 Otsu thresholding

Teori Otsus metode er en algoritme for thresholding av et bilde. Den baserer seg på intra-class variance. Varians i statistikkens verden er gjennomsnittelig avstand fra gjennomsnittet. For å finne denne kvadrerer man alle verdien som avviker fra gjennomsnittet og legger de sammen. Man deler så på antall avvik. Intra-class varians er variansen innenfor en klasse.

I Otsus metode finner man histogrammet av input bilde. Optimalt sett vil man se to klasser eller topper innenfor dette histogrammet. Man prøver så å finne en threshold grense k som gjør at intra-class variansen blir så liten som mulig.

Problemet med denne algoritmen er at den er vedlig avhengig av at det histogrammet generert av bilde har to topper. Hvis bilde har mye støy kan dette jevne ut histogrammet og resultatet kan bli dårlig.

Implementasjon For å kode denne algoritmen fulgte jeg stegene i kapittel 10.3 i boken Digital Image Processing.

- Finn histogrammet for input bildet.
- Regn ut de kumulative distribusjons funksjonene for begge klassene
- Finn den kumulative gjennomsnitts funksjonen for den ene klassen
- Finn den kumulative gjennomsnitts funksjonen for hele histogrammet
- Finn avstanden mellom klassene (Between class variance)
- Finn segmenteringen k som gir den lengste avstanden mellom klassene ved å gå gjennom alle k

Dette vil gi den beste segmenteringen avhengig av at histogrammet inneholder to klasser.

Resultat Algorimten fungerte bra på bilder som ikke inneholdt skygger, eller som var rotert for mye. Skyggene førte til en jevnere overgang av de to klassene i histogrammet. Da jeg fjernet skyggene i bilde og de svarte kantene som er et resultat av rotasjonen av bilde, fungerte algoritmen veldig bra. Jeg fjernet også støy med et gaussisk filter for å få et enda bedre resultat. I figur 8 ser man input bilde til venstre, det samme bilde segmentert med otsus metode til høyre. Nederst har jeg fjernet kanten på roteringen og skyggen på nederst på bilde. Segmenteringen blir dermed bedre. På bilde til høyre valgte otsus metode en threshold på 117, på det nederste bilde ble det valgt en threshold på 152.



Figure 8: Otsu segmentering