به نام خدا

Table of Contents

[**سوال چهار** 3](#_Toc196509592)

[**بخش الف** 3](#_Toc196509593)

[**گزارش‌ کار** 5](#_Toc196509594)

[**خروجی** 6](#_Toc196509595)

[**بخش ب** 7](#_Toc196509596)

[**گزارش کار** 7](#_Toc196509597)

[**خروجی** 8](#_Toc196509598)

# **سوال چهار**

این سوال در 3 قسمت توضیح داده شده است و هر بخش گزارش کار پیاده سازی شده آن آمده است.

## **بخش الف**

2 فیلتر نام برده شده هر دور بر اساس گرادیان لبه ها را تشخیص میدهند این کار با تقریب زدن گرادیان توسط شدت/کمیت عکس صورت میگیرد.

**فیلتر sobel**

این فیلتر تخمین گرادیان عکس را به صورت جهت عمودی و افقی محاسبه میکند. از کانولوشن کمک میگیرد برای هر جهت که معمولا یک کرنل 3 در 3 دارد.

جهت افقی:

[-1 0 +1]

[-2 0 +2]

[-1 0 +1]

جهت عمودی:

[-1 -2 -1]

[ 0 0 0]

[+1 +2 +1]

کرنل افقی شدت تغییرات در جهت افقی را در سرتاسر پیکسل مرکزی اندازه گیری میکند و به سطر مرکزی بیشترین وزن را میدهد. کرنل عمودی دقیقا همین کار را به صورت عمودی میکند. کرنل ها بر روی عکس ورودی پیمایش انجام میدهند و در هر موقعیت یک ضرب نقطه ای محاسبه میکنند که بین کرنل و مجموع پیکسل های عکس است تا یک مقداری که متناظر با پیکسل در خروجی گرادیان تصویر است تولید شود. روش وزن دهی گفته شده باعث میشود که فیلتر گفته شده کمتر به نویز حساس باشد، کرنل هر چه بزرگتر باشد همسایگی بیشتری در نظر میگیرد و سبب میشود اون نواحی بیشتر هموار شوند یعنی نویز کمتر شوند البته که ممکن است این کار باعث blurring و از دست رفته لبه ها بشود.

**فیلتر scharr**

مانند sobel این فیلتر هم تخمین گرادیان را محاسبه میکند. این فیلتر برای بهبود فیلتر سوبل آمده است زیرا فیلتر سوبل حرکت دورانی ندارد و این سبب میشود که واکنش آن نسبت به لبه ها بر اساس جهت لبه ها باشد و فیلتر scharr سعی میکند اینکار را بهبود ببخشید.

کرنل افقی:

[-3 0 +3]

[-10 0 +10]

[-3 0 +3]

کرنل عمودی:

[ -3 -10 -3]

[ 0 0 0]

[ +3 +10 +3]

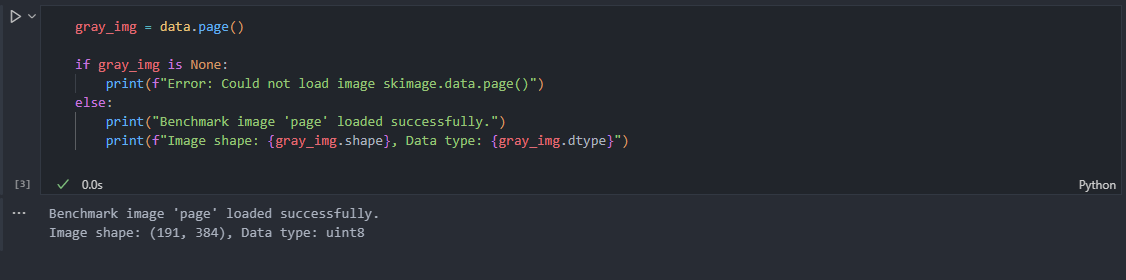
گذاشتن وزن 10 در مرکز سبب میشود که این فیلتر بیشتر به شدت تغییرات حساس شود که باعث میشود بهتر لبه ها را پیدا کند. همچنین وزن های (3و10و3) باعث شده است تا تخمین بهتر انجام شود زیرا به صورت چرخشی حرکت میکند. سایز کرنل آن معمولا 3 در 3 است.

### **گزارش‌ کار**

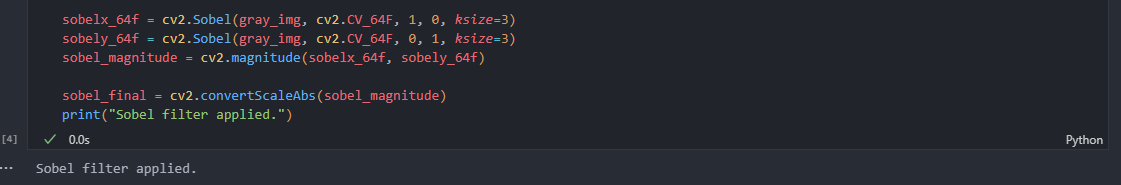


از کتابخانه cv2 برای اعمال عملیات ها مختلف مانند خواندن عکس‌ها استفاده میکنیم.

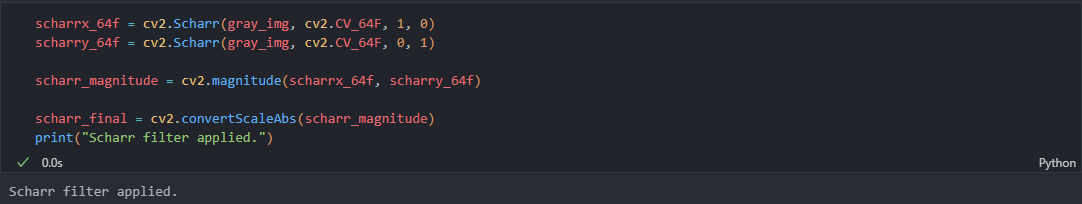
برای نشان دادن خروجی از matplotlib استفاده کرده‌ایم. همچنین تصاویر مورد نیاز را از scikit-image تحت عنوان skimage بارگذاری کرده‌ایم تا بتوانیم یک benchmark معیار را برای سنجش فیلترهای پیاده سازی شده خودمان داشته باشیم و نتایج را بررسی کنیم.



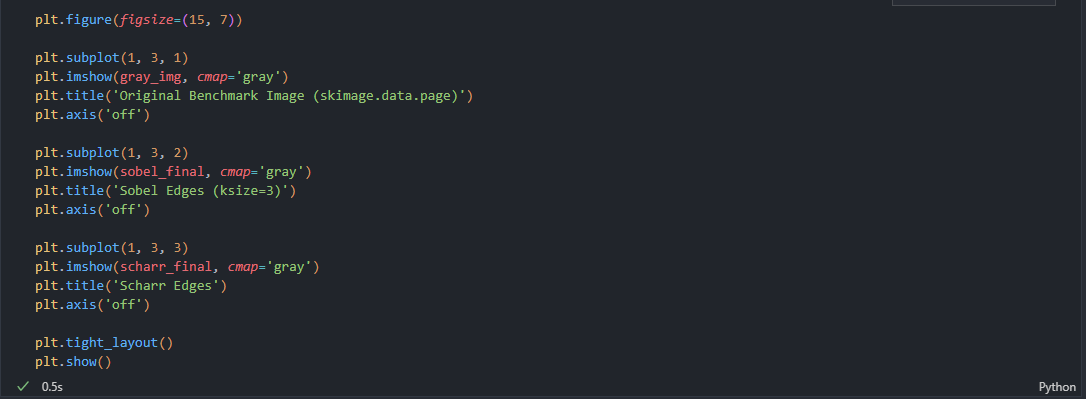
در این قسمت عکس benchmark را بارگذاری کرده ایم و مطمئن میشویم کد به درستی کار میکند.



همانطور که توضیح دادیم با کمک کتابخانه cv2 فیلتر سوبل را بر روی تصویر اعمال میکنیم. گرادیان را در 2 جهت x و y محاسبه میکنیم با دقت 64 بیت. اندازه کرنل را 3 در نظر گرفتیم. در نهایت اندازه گرادیان را بدست آوردیم و بعد اندازه را به خروجی 8 بیت بدون علامت تبدیل کردیم تا بتوانیم آن را نمایش بدهیم.

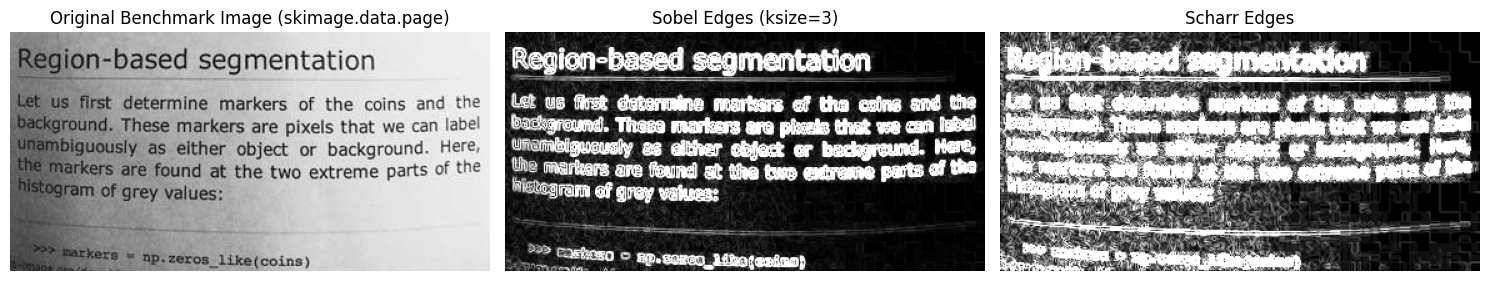


همانطور که توضیح دادیم حالا فیلتر scharr را اعمال میکنیم، به صورت پیشفرض اندازه کرنل 3 هست پس نیازی به تعریف متغیر ksize نخواهد بود. بعد اندازه گرادیان را محاسبه میکنیم و در نهایت مانند سوبل خروجی را به 8 بیت بدون علامت تبدیل میکنیم تا بتوانیم نمایش بدهیم.



در نهایت خروجی را نمایش میدهیم.

### **خروجی**

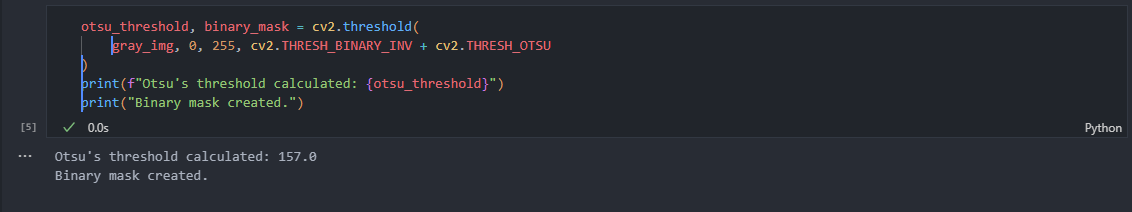


## **بخش ب**

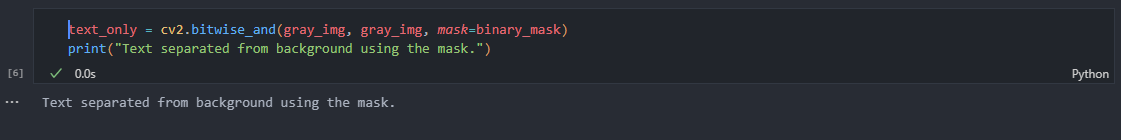
برای پیاده سازی اینکار ما یک آستانه تعریف میکنیم تا بتوانیم به کمک آن یک ماسک باینری بسازیم. با تعریف ماسک باینری پیکسل ها در مجموع 2 مقدار فقط میگیرند یا سیاه هستند 0 و یا سفید هستند (255). در کدی که پیاده سازی میکنیم متن ها در عکس اصلی سفید خواهند بود و پس زمینه سیاه خواهد بود. برای تعریف کردن آستانه نیاز داریم که مقادیر پیکسل هایی که کمیت آنها از آستانه بالاتر بود سفید شوند و یک مقدار بگیرند و همینطور همینکار را برای پیکسل هایی که زیر آستانه هستند انجام میدهیم. به جای اینکه دستی خودمان آستانه تعریف بکنیم از روش Ostu’s Binarization استفاده میکنیم که به صورت خودکار و الگوریتمی میتوانیم آستانه بهینه را تعریف کنیم تا پیکسل ها به 2 کلاس تقسیم شوند. این روش با کاهش واریانس درون هر کلاس صورت میگیرد. بعد از اعمال این روش حالا میتوانیم ماسک خودمان را بسازیم و بعد از آن ماسک را روی تصویر اعمال میکنیم. از عملیات AND به صورت bitwise استفاده میکنیم تا بتوانیم ماسک را روی تصویر اعمال کنیم نحوه عمل کردن این AND به این صورت هست که به هر پیکسل نگاه میکنیم اگر پیکسل متناظر آن در ماسک سفید باشد، مقدار خودش را نگه میدارد. اگر پیکسل متناظر سیاه باشد، پیکسل مقدار خودش را از دست میدهد و مقدار جدید میگیرد.

### **گزارش کار**

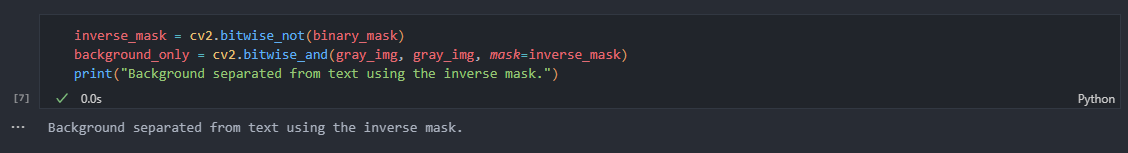
عمده موارد مانند قسمت الف است، در این قسمت فقط موارد جدید اضافه شده را بررسی میکنیم.



همانطور که در بالا توضیح دادیم ابتدا یک ماسک میسازیم و بعد از آن روش Ostu’s را پیاده سازی میکنیم. این کار با کمک کتابخانه open cv انجام میشود، پیکسل هایی که زیر آستانه قرار بگیرند سفید یعنی 255 و آنهایی که بالا باشند 0 میگیرند. از آنجایی که از THRESH\_BINARY\_IMV استفاده میکنیم باعث میشود متن که سیاه است تبدیل به سفید شود.



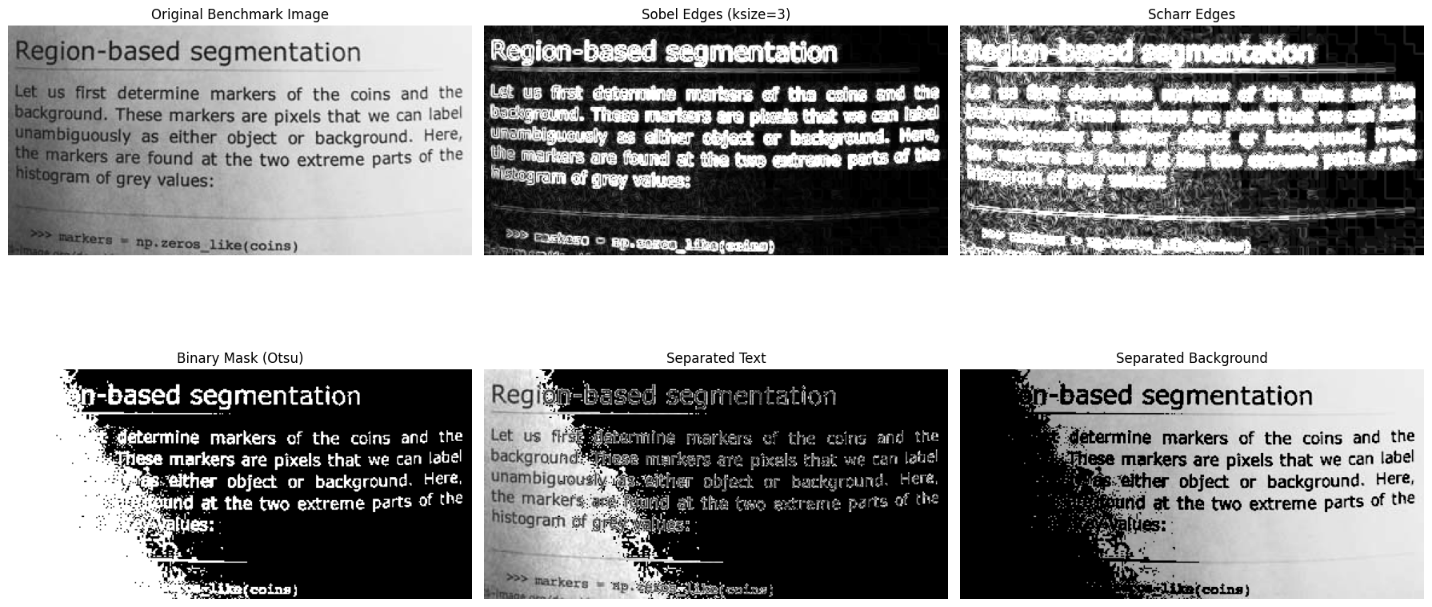
با کمک ماسک میایم و متن را جدا میکنیم با استفاده از اعمال ماسک روی تصویر اولیه که به صورت grayscale است. جایی که ماسک 255 باشد مقدار پیکسل همان اولیه باقی میماند. اگر ماسک صفر باشد مقدار پیکسل در آنجا 0 میشود.



یک ماسک معکوس ساخته ایم تا بتوانیم پس زمینه را جدا کنیم.

در ادامه با کمک کتابخانه matplotlib خروجی را نمایش دادیم.

### **خروجی**



س