Code Explanation

0. Install and Import Dependencies

```
!pip install easyocr
!pip install imutils.
import cv2
from matplotlib import pyplot as plt
import numpy as np
import imutils
import easyocr
import easyocr
```

1. Read in an Image, Grayscale and Blue

```
img = cv2.imread('image3.jpg')
```

this command reads the image we want.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

In this step we recolored it

to recolor we used **cvtColour**, so this basically

allows us to convert from single color code

to a different one.

Now here we have passed our initial image

which is stored in a variable called **img**

and then we passed the colour conversion we

want to do here we've passed **BGR2GRAY**

because **cv2** reads the image in Blur Green Red

and then we converted it into **GRAY**.

```
plt.imshow(cv2.cvtColor(gray, cv2.COLOR_BGR2RGB))
```

Out[7]: <matplotlib.image.AxesImage at 0x256e27f7790>

we showed the image using matplotlib

matplotlib expects rgb now because our

image by default is in BGR we've just gone

and converted it to RGB

We can eventually drop this conversion we

are not concerned about how it looks like. To avoid

the vagueness we'll apply that conversion property.

2. Apply filter and find edges for localization.

[Filtering is basically going to allow us to remove noise from our image and our Edge detection is actually going to be able to detect edges within image]

```
bfilter = cv2.bilateralFilter(gray, 11, 17, 17)
```

Noise reduction using **bilateralFilter** and passed image

that we want to perform our noise reduction so

#Our we've passed our gray image and number of

properties, these allow us to specify how intensely

we want our noise reduction and smoothing so in

this case **11, 17, 17** seems to work pretty.

```
edged = cv2.Canny(bfilter, 30, 200)
```

Edge Detection

Canny Algorithms, this allows us to detect edges and

#then we have number of parameters that we can

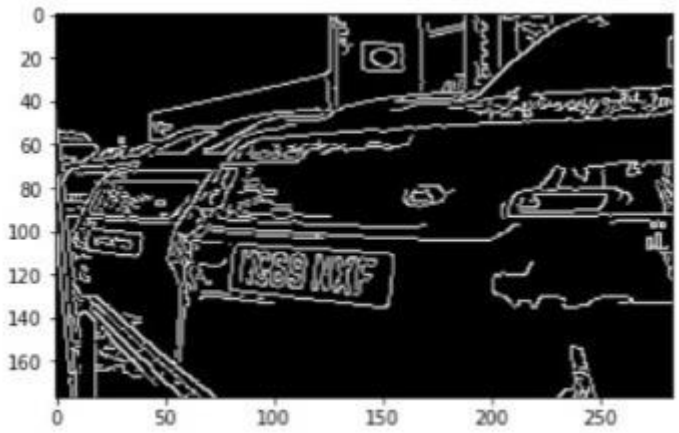
pass through and these can be tuned depending

on what you find actually works but ideally you

want to be able to see these edges exactly or

similar to what you're seeing in the image.

```
plt.imshow(cv2.cvtColor(edged, cv2.COLOR_BGR2RGB))
```



similar line to visualise the edges.

3. Find the contours and Apply Mask

[contour detection is all to do with detecting where these lines are and detecting polygons within those lines so ideally shapes within our images, now what we're actually looking for is a contour which has four points so ideally we want to be able to a rectangle because that's most likely to be the shape of our number plate]

```
keypoints = cv2.findContours(edged.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

[We have used **cv2.findContours** to find contours, so this basically goes through image and tries to find shapes effectively. a.k.a. contours so to do that we've passed our edged image so this is effectively our image that we've applied edge detection to then what we've done is specified two arguments

- **cv2. RETR_TREE:** is how we want our results to returned in this case we've

returned a tree, so this basically allows us to traverse our tree to find different levels of contour.

- **cv2. CHAIN_ APPROX_SIMPLE:** is how we want our results returned or what type of results we want so basically by passing through chain approx. simple we're going to get is a simplified version of the contour ideally if we find a line we don't want every single point on line we just want approximate the two points or key points that represents that line. So this approx simple algorithm basically allows us to approximate what the actual contour looks like so ideally What you should get is a contour with four points for our number plate section so basically after all the process we have stored it in a variable called **keypoints**.

```
contours = imutils.grab_contours(keypoints)
```

Here we've used **imutils** to go and grab our contours so this basically simplifies how our contours actually returned and then

```
contours = sorted(contours, key=cv2.contourArea, reverse=True)[:10]
```

Here we sorted them and returned the top ten Contours. In order to do that we've used python function **sorted** and we also specific the key on whose basis we want the sorting to be done. **reverse=True** means we are going or sorting in descending order. **[:10]** means top 10

[The next thing we're gonna do is loop through each contour one of these contours and see whether or not they actually represent a square or a number plate. so we're basically going
to be checking whether or not there's four keypoints within those so]

```
location = None
for contour in contours:
    approx = cv2.approxPolyDP(contour, 10, True)
    if len(approx) == 4:
        location = approx
        break
location
```

so we're now gone and filtered through our different contours in order to find the location of where our number plate might actually be. Here what we're done is that

1. we've set up a temporary variable called **location** and we've looped over each one the contour that we had in variable **contours** up there. This was our contour

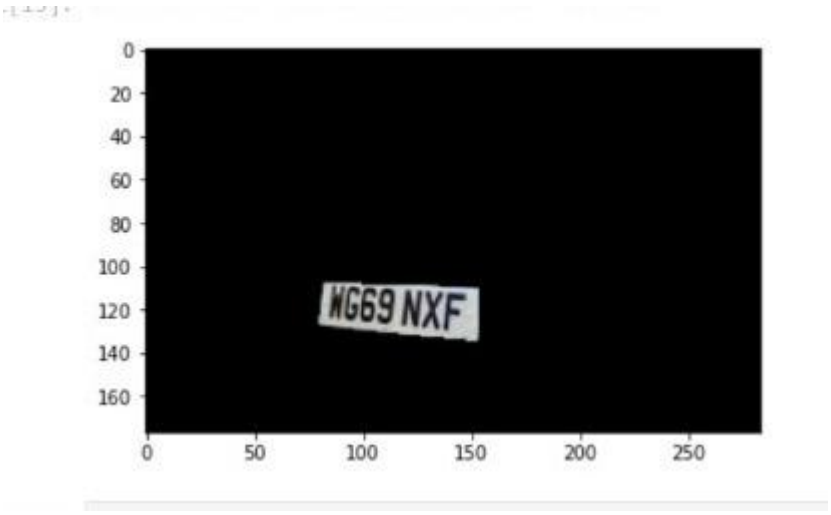
list.

2. Then we have used opencv method **approxPolyDP** So this allows us to approximate the polygon from our contour. Parameters passed:

- **contour**
- **10:** basically allows us to specify how accurate or fine grain our approximation is so say for eg. we've got a contour that that has a whole lot of little dents in it how high we set a polyDP going to round off that contour. So say for eg. you had something which is roughly a rectangle but had a couple of little dents. we set this number the more rough the estimation is going to be. So basically it might skip one over those little dents and specify that that's really a straight line. 10 is fine to detect a sequel but too detect 'it's' got more than 4 keypoints

3. If our approximation have got 4 keypoints that's most likely to be got our number location. Hence we break there.

```
mask = np.zeros(gray.shape, np.uint8)
new_image = cv2.drawContours(mask, [location], 0,255, -1)
new_image = cv2.bitwise_and(img, img, mask=mask)
plt.imshow(cv2.cvtColor(new_image, cv2.COLOR_BGR2RGB))
```



We're here done couple of key things.

1) we've created a blank mask and in order to do that we've used np.zeros so this basically creates a mask that's going to be the same shape as our original gray image to that we've passed the shape of our original image and then we've specified how we want to fill it in, np.uint8 so basically we're filling it with blank zeros.

2) then we've drawn contours within that image so to do that we've passed our

temporary image which is 'mask', we've specified what contour we want to draw in this case it's the location and then we've specified how actually want to draw it and what we want to display in that

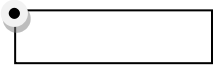
3) Then we've Overlayed that mask over original image and to do that We've used cv2.bitwise_and and this basically allows us to return our segment or the segment of our image which actually represent our number plate

```
(x,y) = np.where(mask==255)
(x1, y1) = (np.min(x), np.min(y))
(x2, y2) = (np.max(x), np.max(y))
cropped_image = gray[x1:x2+1, y1:y2+1]
plt.imshow(cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB))
```

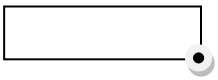
cropping the image

1) we started with finding out every single section where our image isn't black so basically what we're doing is we're scoring those in variable (x, y). so we 're going to get is a of coordinates which represents all of the number plate section

2) Then we've grabbed the minimum and y value

x1,y1 

3) Then we're grabbed the maximum x and y value.

x2,y2 

4) Then we've used a bit of array filtering so we've gone and grabbed point x1 to x2 + 1 to give us a little bit of buffer and this has given us this range over here and then we've grabbed y1: y2+1 and when combined it gives us the cropped- image of our number plate.

4. Use Easy OCR to Read Text.

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

we've used easyocr to go and grab the text from the image.

In order to do that we've used the reader method so this instantiates our reader so first we've specified the language and then we've used our reader to extract text from the **cropped_image**

* whenever you're working on ANPR your result will vary depending upon the type of image you are working with.

5. Render result.

```
text = result[0][-2]
font = cv2.FONT_HERSHEY_SIMPLEX
res = cv2.putText(img, text=text, org=(approx[0][0][0], approx[1][0][1]+60), fontFace=font,
                  fontScale=1, color=(0,255,0), thickness=2, lineType=cv2.LINE_AA)
res = cv2.rectangle(img, tuple(approx[0][0]), tuple(approx[2][0]), (0,255,0),3)
                  plt.imshow(cv2.cvtColor(res, cv2.COLOR_BGR2RGB))
```

Rendered our result to our image

1.) First specified our text, we've gone to the result retuned by easyocr and we're grabbed our first instance here then we've used -2 to go from the end of our array to pick up our result. that. so -2 gives the number plate text.

2) Then to go and overlay our results what we've gone and done is specified our font

3) We've then gone and applied our text which is this component here so to do that we've passed our image, text, then the location that we want it so in this case we've gone & specified the bottom it's a corner plus 60 so it's a little bit further down,

specified the font, font scale, font color, thickness, linetype

4) drawn a rectangle around our number plate passed image, we've passed our first tuple which give us our first coordinate and then second tuple which gives us our bottom coordinate that draws our image, colour, line thickness.