UDACITY

PROJECT

Advanced Lane Finding

A part of the Self-Driving Car Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Requires Changes

2 SPECIFICATIONS REQUIRE CHANGES

You have done a big job and have a very good first step! It is a really tough project. And it is awesome that you almost complete it 👍
Please correct a few small issues mentioned below to meet specification and you are done! Good luck!

Writeup / README

The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled.

Well done with writeup! 👍
It is always important to understand advantages and limitations of your algorithm and know ways to improve it.

Camera Calibration

OpenCV functions or other methods were used to calculate the correct camera matrix and distortion coefficients using the calibration chessboard images provided in the repository (note these are 9x6 chessboard images, unlike the 8x6 images used in the lesson). The distortion matrix should be used to un-distort one of the calibration images provided as a demonstration that the calibration is correct. Example of undistorted calibration image is Included in the writeup (or saved to a folder).

Well done with camera calibration process! Here is more info about this process in documentation if you are interested:
http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html?
http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
http://docs.opencv.org/3.1.0/dc/dbb/tutorial_py_calibration.html

And very detailed Microsoft research:
http://research.microsoft.com/en-us/um/people/zhang/calib/

Pipeline (test images)

Distortion correction that was calculated via camera calibration has been correctly applied to each image. An example of a distortion corrected image should be included in the writeup (or saved to a folder) and submitted with the project.

`undistort` function is successfully used for images.

Remember that usually Python is used by self-driving car engineers for prototyping and final algorithms are usually implemented in C/C++ as it is faster than Python and almost all data are handled in self-driving car in real time. Here you can find article about distortion correction in C++:
https://medium.com/@mimoralea/but-self-driving-car-engineers-dont-need-to-know-c-c-right-3230725a7542#.ge488ni33

Note please that you will apply C++ in term 2 (though in other context).

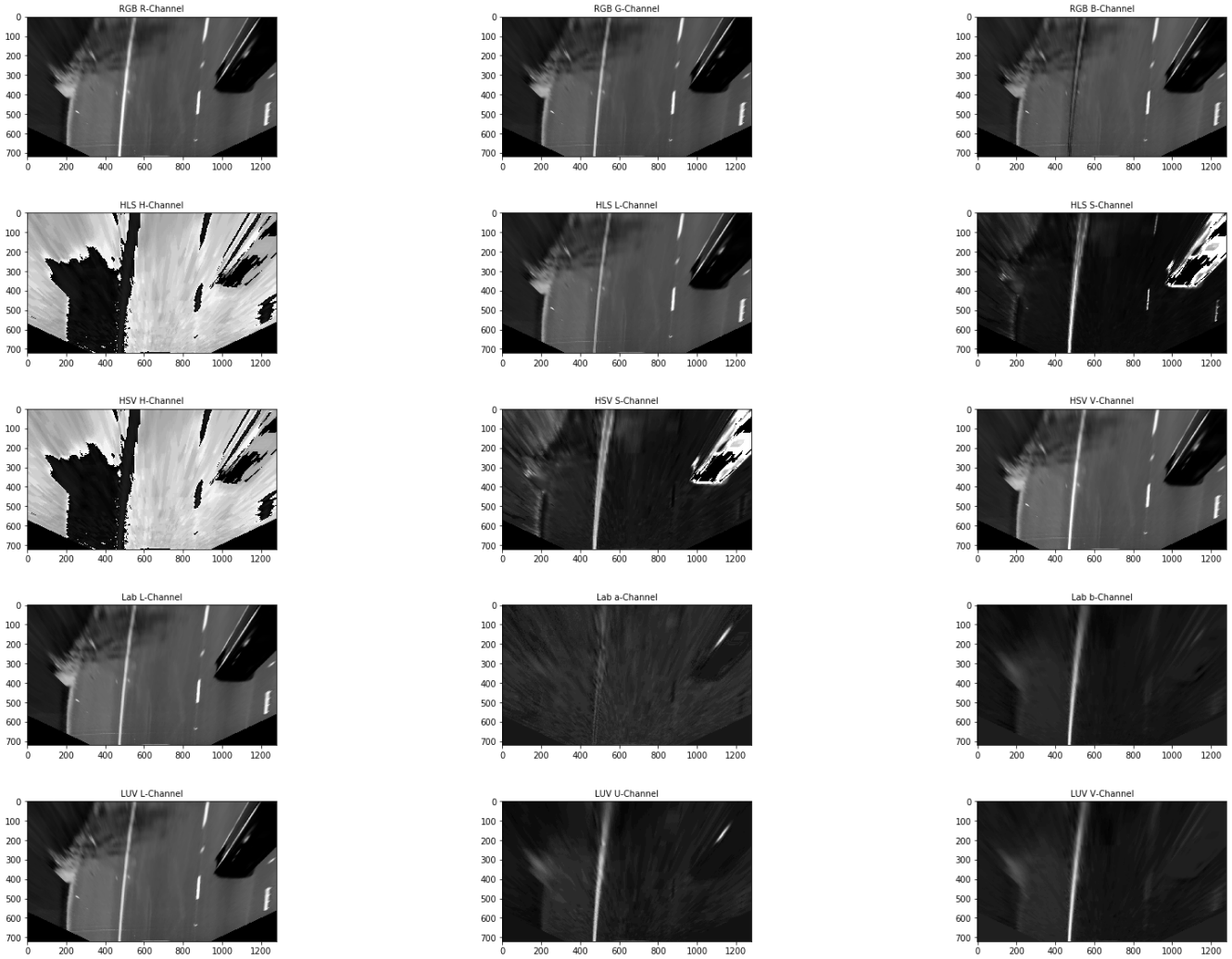
A method or combination of methods (i.e., color transforms, gradients) has been used to create a binary image containing likely lane pixels. There is no "ground truth" here, just visual verification that the pixels identified as part of the lane lines are, in fact, part of the lines. Example binary images should be included in the writeup (or saved to a folder) and submitted with the project.

Well done with color transforms and gradient methods to create binary images for further lane detection!

Briefly about lines, colorspaces, and channels in different colorspaces:

- S-channel of HLS colorspace is good to find the yellow line and in combination with gradients, it can give you a very good result.
- R-channel of RGB colorspace is pretty good to find required lines in some conditions.
- L-channel of LUV colorspace for the white line.
- B-channel of LAB colorspace may be good for the yellow line.
  You just need to choose good thresholds for them.

Here are some examples of mentioned channels (in birds-eye view for the original image provided below):



Original image for channels shown above:

Note please also that gradients can give big noise in some conditions (for example with shadowed road or when lightning conditions changes drastically).
If you use Sobel remember that kernel size is limited to 1, 3, 5, or 7 values only:
http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#cv.Sobel

Here is a code to build a widget for parameters fine tuning (for jupyter notebook only):

```
from IPython.html import widgets
from IPython.html.widgets import interact
from IPython.display import display

image = mpimg.imread('path_to_your_image')
image = cv2.undistort(image, mtx, dist, None, mtx)

def interactive_mask(ksize, mag_low, mag_high, dir_low, dir_high, hls_low, hls_high, bright_low, bright_high):
    combined = combined_binary_mask(image, ksize, mag_low, mag_high, dir_low, dir_high,\
                                    hls_low, hls_high, bright_low, bright_high)
    plt.figure(figsize=(10,10))
    plt.imshow(combined,cmap='gray')

interact(interactive_mask, ksize=(1,31,2), mag_low=(0,255), mag_high=(0,255),\
         dir_low=(0, np.pi/2), dir_high=(0, np.pi/2), hls_low=(0,255),\
         hls_high=(0,255), bright_low=(0,255), bright_high=(0,255))
```

Where `combined_binary_mask` function returns binary combination of different color and/or gradient thresholds and

```
ksize,
mag_low,
mag_high,
dir_low,
dir_high,
hls_low,
hls_high,
bright_low,
bright_high
```

are parameters for thresholding (if you use other, you can change this part of code).

As a result, you will receive something like that and will be able to fine-tune parameters on the fly:
parameters-fine-tune.png
and will be able to see an impact of each parameter on the resulting binary image with defined combination of thresholds.

Here is similar decision for plain python (without notebook):

```
def adjust_threshold():
    image1 = cv2.imread("./test_images/test1.jpg")
    image2 = cv2.imread("./test_images/test2.jpg")
    image3 = cv2.imread("./test_images/test3.jpg")
    image4 = cv2.imread("./test_images/test4.jpg")
    image5 = cv2.imread("./test_images/test5.jpg")
    image6 = cv2.imread("./test_images/test6.jpg")
```

```
image = np.concatenate((np.concatenate((image1, image2), axis=0),np.concatenate((image3, image4), axis=0)), axis=1)
image = np.concatenate((np.concatenate((image5, image6), axis=0), image), axis=1)

threshold = [0 , 68]

kernel = 9

direction = [0.7, 1.3]
direction_delta = 0.01

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)


cv2.namedWindow('test', cv2.WINDOW_NORMAL)

cv2.waitKey(0)

while True:
    key = cv2.waitKey(1000)
    print("key = ", key)
    if key == 55: # key "Home"
        if threshold[0] > 0:
            threshold[0] = threshold[0] - 1
        if direction[0] > 0:
            direction[0] = direction[0] - direction_delta

    if key == 57: # key "PgUp"
        if threshold[0] < threshold[1]:
            threshold[0] = threshold[0] + 1

        if direction[0] < direction[1] - direction_delta:
            direction[0] = direction[0] + direction_delta

    if key == 2424832: # left arrow
        if threshold[1] > threshold[0]:
            threshold[1] = threshold[1] - 1

        if direction[1] > direction[0] + direction_delta:
            direction[1] = direction[1] - direction_delta

    if key == 2555904: # right arrow
        if threshold[1] < 255:
            threshold[1] = threshold[1] + 1

        if direction[1] < np.pi/2:
            direction[1] = direction[1] + direction_delta

    if key == 49: # key "End"
        if(kernel > 2):
            kernel = kernel - 2
    if key == 51: # key "PgDn"
        if(kernel < 31):
            kernel = kernel + 2

    if key == 27: # ESC
        break

    binary = np.zeros_like(image)
    binary[(gray >= threshold[0]) & (gray <= threshold[1])] = 1

    cv2.imshow("test", 255*binary)
    print(threshold)
```

```
        print(direction)
        print(kernel)
```

You will be able to tune different parameters using keypress. The script works in an infinite loop. Instead of this plain python, you can also use cv2 user interface API for this purpose:
http://docs.opencv.org/2.4.11/modules/highgui/doc/user_interface.html#

And here is also a helper code to find yellow and white lines separately if you want to investigate this question further:

```
    b = np.zeros((img.shape[0],img.shape[1]))

    def thresh(img, thresh_min, thresh_max):
        ret = np.zeros_like(img)
        ret[(img >= thresh_min) & (img <= thresh_max)] = 1
        return ret

    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    H = hsv[:,:,0]
    S = hsv[:,:,1]
    V = hsv[:,:,2]

    R = img[:,:,0]
    G = img[:,:,1]
    B = img[:,:,2]

    t_yellow_H = thresh(H,10,30)
    t_yellow_S = thresh(S,50,255)
    t_yellow_V = thresh(V,150,255)

    t_white_R = thresh(R,225,255)
    t_white_V = thresh(V,230,255)

    b[(t_yellow_H==1) & (t_yellow_S==1) & (t_yellow_V==1)] = 1
    b[(t_white_R==1)|(t_white_V==1)] = 1
```

One more approach is to use a voting system for the final thresholding instead of the logical combination of different binary images. When the threshold is very clean like B channel from LAB space, it will have a relatively high confidence number. At the end, all the confidence number are summed up and compared with a total threshold. However, this method takes more run time than simply taking some combinations:

```
    conf_1 = 1
    conf_2 = 2
    conf_3 = 3
    threshold_vote = 3
    addup = img_sobelAbs*conf_1 +img_sobelMag*conf_1 +img_SThresh*conf_1 +img_LThresh*conf_2 +img_sobelDir*conf_1 + img_BThr
esh*conf_3

    combined[(addup >= threshold_vote)]=1
```

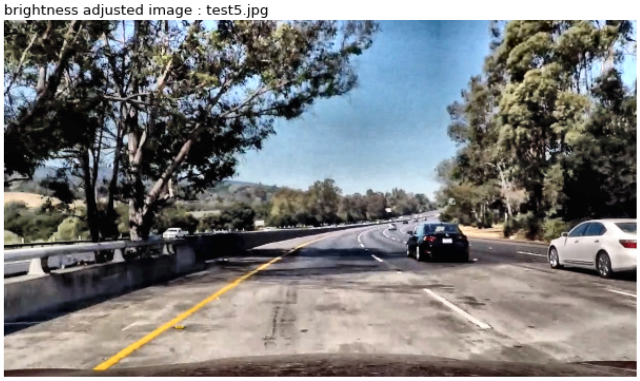
Where `img_sobelAbs`, `img_sobelMag`, `img_SThresh`, `img_LThresh`, `img_sobelDir`, `img_BThresh` are different color or sobel thresholds.

Also you can notice that in some places of the video there are very intensive shadows. You can pre-processed the images through a brightness adjustment filter using a Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm. This will help you better detect the yellow lane line road markers that have over-cast shadows in the original image for subsequent steps (e.g. getting a binary threshold color mask for yellow and white lane lines).

Brightness adjust using CLAHE algorithm

undistorted image : test5.jpg

brightness adjusted image : test5.jpg



Also, you can use morphological dilation and erosion to make the edge lines continuous:

```
kernel = np.ones((5, 5), np.uint8)
closing = cv2.morphologyEx(binary_mask.astype(np.uint8), cv2.MORPH_CLOSE, kernel)
```

http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=morphologyex#morphologyex
http://docs.opencv.org/2.4/doc/tutorials/imgproc/opening_closing_hats/opening_closing_hats.html

Here you can find an interesting article about other methods (LDA and LLC) to detect lines in different lighting conditions:
https://otik.uk.zcu.cz/bitstream/handle/11025/11945/Jang.pdf?sequence=1

OpenCV function or other method has been used to correctly rectify each image to a "birds-eye view". Transformed images should be included in the writeup (or saved to a folder) and submitted with the project.

Good job with "birds-eye view"!

Remember that the next step to improve this part of algorithm is to take into account road slope and use dynamic points to transform image to "birds-eye view".

Here are several interesting papers about obtaining and usage of birds-eye view:
http://www.ijser.org/researchpaper%5CA-Simple-Birds-Eye-View-Transformation-Technique.pdf
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3355419/
https://pdfs.semanticscholar.org/4964/9006f2d643c0fb613db4167f9e49462546dc.pdf
https://pdfs.semanticscholar.org/4074/183ce3b303ac4bb879af8d400a71e27e4f0b.pdf

Methods have been used to identify lane line pixels in the rectified binary image. The left and right line have been identified and fit with a curved functional form (e.g., spine or polynomial). Example images with line pixels identified and a fit overplotted should be included in the writeup (or saved to a folder) and submitted with the project.

Well done using histogram and sliding window method to find right and left lane lines!

Here are some good links for other methods described:
https://www.researchgate.net/publication/257291768_A_Much_Advanced_and_Efficient_Lane_Detection_Algorithm_for_Intelligent_Highway_Safety
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5017478/

Algorithm based on current project😄:
https://chatbotslife.com/robust-lane-finding-using-advanced-computer-vision-techniques-46875bb3c8aa#.l2uxq26sn

The following validation criteria can be used to remove incorrect lines:

- lane width is in defined borders
- lane lines have the same concavity
- the second degree parameter of both fits are not too different (ratio of parameters is less than some value)
- ratio of lines curvature is not too big (be careful here with straight lane)
- distance between left and right lines at the base of the image is roughly the same as at the top of the image (in birds-eye view)
- lane curvature, distance from the center, polynomial coefficients and so on.. don't differ a lot from the same values from the previous frame

Here the idea is to take the measurements of where the lane lines are and estimate how much the road is curving and where the vehicle is located with respect to the center of the lane. The radius of curvature may be given in meters assuming the curve of the road follows a circle. For the position of the vehicle, you may assume the camera is mounted at the center of the car and the deviation of the midpoint of the lane from the center of the image is the offset you're looking for. As with the polynomial fitting, convert from pixels to meters.

Radius of curvature looks too big, we expect values from 300 to 1500 meters for the curved road, but in your case values are much greater. As was mentioned earlier the problem is in conversion from meters to pixels (it seems you didn't convert values at all). Here is you code:

```
    ploty = np.linspace(0, 719, num=720)
    y_eval = np.max(ploty)
    left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
    right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit[1])**2)**1.5) / np.absolute(2*right_fit[0])
```

Nad here is appropriate code from the lesson:

```
# Define conversions in x and y from pixels space to meters
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension

# Fit new polynomials to x,y in world space
left_fit_cr = np.polyfit(ploty*ym_per_pix, leftx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, rightx*xm_per_pix, 2)
# Calculate the new radii of curvature
left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
# Now our radius of curvature is in meters
print(left_curverad, 'm', right_curverad, 'm')
# Example values: 632.1 m    626.2 m
```

Position towards the center is also calculated incorrectly. Here is your code:

```
away_from_center = ((right_fitx[0] - left_fitx[0])/2 - 640) * 3.5/600
```

Here `(right_fitx[0] - left_fitx[0])/2` you should calculate center of the lane based on the found lines, but actually you calculate width of the lane instead. To find center of the lane you should sum up base points of the lines.
Another problem here is that you choose points far from the car `right_fitx[0]` and `left_fitx[0]`, but offset should be calculated near the car, so points also should be chosen near the car.

Note that currently offset values are arond 1-2 meters, but having lane width 3.7 meters and offset 1-2 meters car should drive in the neighboring lane.

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. This should demonstrate that the lane boundaries were correctly identified. An example image with lanes, curvature, and position from center should be included in the writeup (or saved to a folder) and submitted with the project.

Pipeline (video)

The image processing pipeline that was established to find the lane lines in images successfully processes the video. The output here should be a new video where the lanes are identified in every frame, and outputs are generated regarding the radius of curvature of the lane and vehicle position within the lane. The pipeline should correctly map out curved lines and not fail when shadows or pavement color changes are present. The output video should be linked to in the writeup and/or saved and submitted with the project.

Problem mentioned earlier is still actual. Currently, the lane is not properly found on the part of the video. On screens below you can see that found lane does not perfectly fit actual lane:


Left curverad: 1781.0031216 Right curverad:4229.80398006
Away from cernter: -0.274361519947Meter


Left curverad: 1329.0608183 Right curverad:1093.44862573
Away from cernter: -2.02473514794Meter


Left curverad: 1532.92941338 Right curverad:1257.3642662
Away from cernter: -1.90648779882Meter

To resolve these problems I would recommend you implement these items in your algorithm:

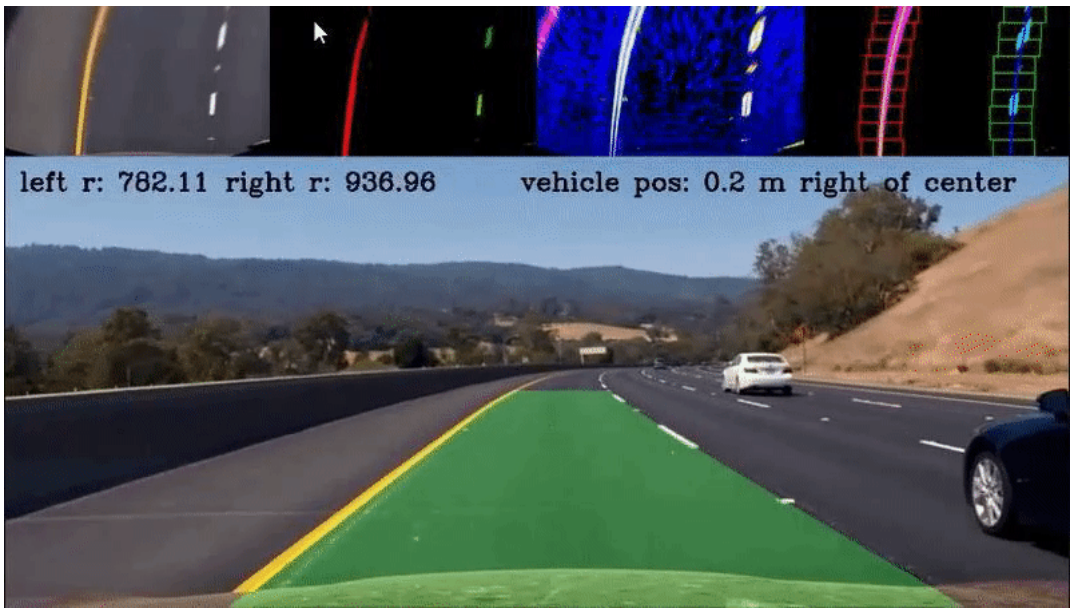
1. if lines were not detected (first frame or validation criteria fails for several times) - use sliding window or centroid method
2. if lines were detected previously - search new lines only around (in some margins) previously detected lines
3. validate correctness of your lines based on validation criteria provided above (in section where I commented your histogram method)
4. average lines based on previous several frames - lines will change more smoothly
5. (here I will just repeat items 1 and 2 in other words) take previously detected lines if validation criteria from item 3 fails or use sliding window once again if lines can't be detected for several frames

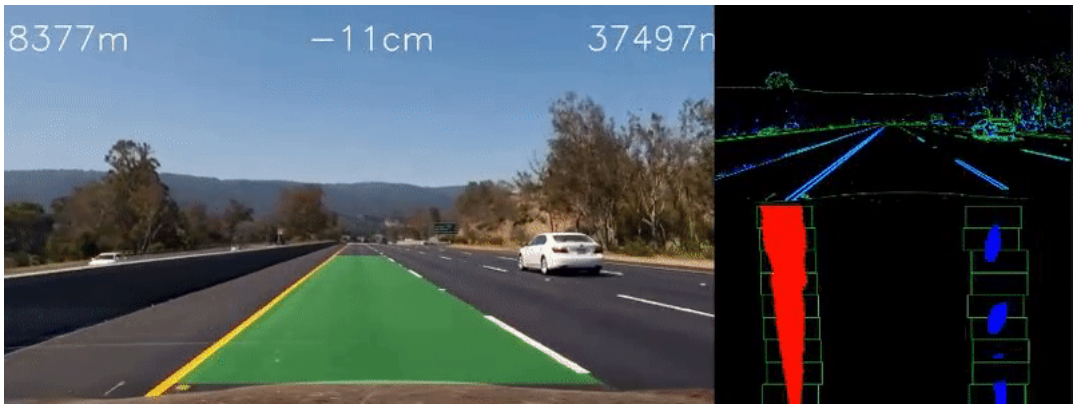
You can follow these tips:
https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/2b62a1c3-e151-4a0e-b6b6-e424fa46ceab/lessons/40ec78ee-fb7c-4b53-94a8-028c5c60b858/concepts/7ee45090-7366-424b-885b-e5d38210958f

Or you can play more with thresholds to be able to detect lines correctly even on shadowed road. As a suggestion, you can save problem frames of the video and try to fine tune your parameters particular for these frames. But such approach is not ideal because using it you overfit your algorithm towards current video, but your purpose is to make it more general.

Note please that `subclip` method can be very useful to test your implementation on short parts (most challenged) of video. You can read more about it for example here:
https://pypi.python.org/pypi/moviepy#example

You can also try to implement debug video like this one:





Here is a helper code that can help you to start with such implementation:

```
# Process pipeline with aditional information
def process_image_ex(img):
    undist = # function that returns undistorted image
    img_binary, img_stack = # funcation that returns binary image (img_binary) and image with combination of all thresholds
 images (img_stack) - it will be displayed near process frame later
    warped, Minv = # function that returns birds-eye view
    lanes, ploty, left_fitx, right_fitx, left_curverad, right_curverad, center_dist = # function that detects lines and lane
    output = # function that warp image back to perspective view

    output1 = cv2.resize(img_stack,(640, 360), interpolation = cv2.INTER_AREA)
    output2 = cv2.resize(lanes,(640, 360), interpolation = cv2.INTER_AREA)

    # Create an array big enough to hold both images next to each other.
    vis = np.zeros((720, 1280+640, 3))

    # Copy both images into the composed image.
    vis[:720, :1280,:] = output
    vis[:360, 1280:1920,:] = output1
    vis[360:720, 1280:1920,:] = output2

    return vis
```

Note please that values when you combine `output1` and `output2` images into final visualization can vary in your implementation.

Discussion

Discussion includes some consideration of problems/issues faced, what could be improved about their algorithm/pipeline, and what hypothetical cases would cause their pipeline to fail.

Well done with final discussion!

RESUBMIT

DOWNLOAD PROJECT

Learn the best practices for revising and resubmitting your project.

RETURN TO PATH