Advanced Lane finding

The goal of this project is to identify road lanes in a video stream of a forward facing camera mounted centrally in a moving vehicle.

We'll be using image manipulation techniques to extract enough information from each frame or image of the video and identify the lane lines, the radius of curvature and the distance from the camera to the center line of the road.

Project Structure:

- Advanced_lane_finding.ipynb : Pipeline implemented in Jupyter notebook
- camera_cal/ : Directory with calibration images
- test_images/ : Directory with test images
- output_images : directory with output images
- videos/ : Directory with input and output videos

Project Overview:

In order to detect the lane lines in a video stream we must accomplish the following:

- **Camera Calibration** - Calibrate the camera to correct for image distortions. For this we use a set of chessboard images, knowing the distance and angles between common features like corners, we can calculate the transformation functions and apply them to the video frames.

- **Color Transform** - We use a set of image manipulation techniques to accentuate certain features like lane lines. We use color space transformations, like from RGB to HLS, channel separation, like separating the S channel from the HLS image and image gradient to allow us to identify the desired lines.

- **Perspective Transform** - We apply a "bird's-eye view transform" that let's us view a lane from above and thus identify the lane lines, measure its curvature and respective radius.

- **Lane Pixel Detection** - We then analyze the transformed image and try to detect the lane pixels. We use a series of windows and identify the lane lines by finding the peeks in a histogram of each window's.

- **Image augmentation** - We add a series of overlays to the image to identify the lane lines, show the "bird's eye view" perspective, show the location of the rectangle windows where the lane pixels are and finally metrics on the radius of curvature and distance to the center of the road.

- **Pipeline** - We finally put it all together in a pipeline so we can apply it to the video stream.

Camera Calibration

Before we can use the images from the front facing camera we must calibrate it so we can correctly measure distances between features in the image.

To do this we first must find the calibration matrix and distortion coefficients for the camera given a set of chessboard images.

We start by preparing a pattern variable to hold the coordinates of the chessboard's squares

```
pattern = np.zeros((pattern_size[1] * pattern_size[0], 3), np.float32)
pattern[:, :2] = np.mgrid[0:pattern_size[0], 0:pattern_size[1]].T.reshape(-1, 2)
```

we then loop through each image, convert it to grayscale, and try to find the chessboard corners with **OpenCV findChessboardCorners** function.

```
image = mpimg.imread(path)
# convert to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
# find the chessboard corners
f, corners = cv2.findChessboardCorners(gray, pattern_size, None)
```

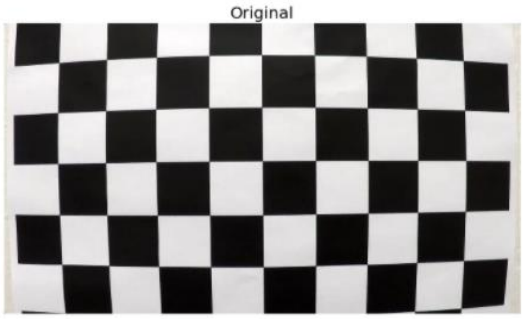
In the case where we can find the corners, we add them to our pattern_points and image_points so we can later use them to calibrate the camera

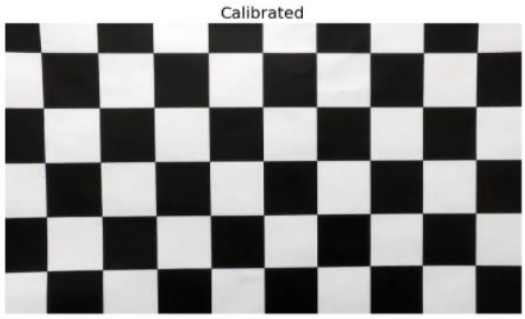
```
if f:
    pattern_points.append(pattern)
    image_points.append(corners)
    image_size = (image.shape[1], image.shape[0])
```

Finally, we calibrate the camera and get our matrix and distortion

```
matrix, dist = cv2.calibrateCamera(pattern_points, image_points, image_size, None, None)
```

Below you will find original and calibrated images shown.





Color transform

Here we use a series of image manipulation techniques to detect the edges, lines of an image.

We start by getting the **gradient absolute value** using OpenCV Sobel operator

```
sobel = np.absolute(cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=sobel_ksize))
```

We then calculate the **gradient magnitude**

```
x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=sobel_ksize)
y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=sobel_ksize)

magnit = np.sqrt(x**2 + y**2)
```

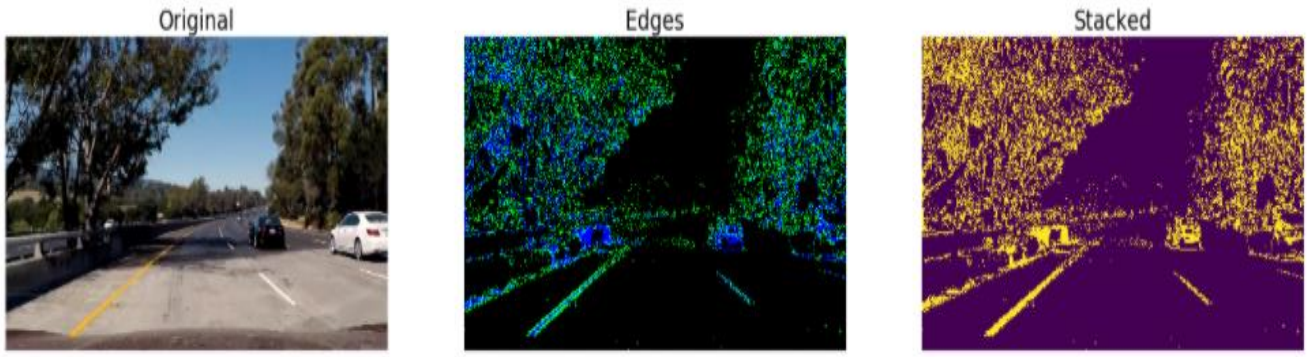
And the **gradient direction**

```
x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=sobel_ksize)
y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=sobel_ksize)

direct = np.arctan2(np.absolute(y), np.absolute(x))
```

Finally, we extract the **S** channel from the **HLS** color space and apply a threshold to it

```
# Convert to HLS space
hls = cv2.cvtColor(np.copy(image), cv2.COLOR_RGB2HLS).astype(np.float)
# Separate and get the S channel
s = hls[:, :, 2]

mask = np.zeros_like(image)
mask[(image >= threshold[0]) & (image <= threshold[1])] = 1
```

We apply all these transformations so we can identify the edges on the lane lines, here's an example:



Perspective transform

We defined a matrix of source and destination points in the images as to transform them to the "bird's eye view" using OpenCV getPerspectiveTransform function.
Here you can see the defined vertices for the "region of interest"

```
height = image.shape[0]
width = image.shape[1]
# Vertices coordinates in the source image
s1 = [width // 2 - 76, height * 0.625]
s2 = [width // 2 + 76, height * 0.625]
s3 = [-100, height]
s4 = [width + 100, height]
src = np.float32([s1, s2, s3, s4])
# Vertices coordinates in the destination image
d1 = [100, 0]
d2 = [width - 100, 0]
d3 = [100, height]
d4 = [width - 100, height]
dst = np.float32([d1, d2, d3, d4])
```

And here the actual transformation:

```
# Given src and dst points we calculate the perspective transform matrix
M = cv2.getPerspectiveTransform(src, dst)
# Warp the image
warped = cv2.warpPerspective(image, M, (width, height))
# We also calculate the oposite transform
unwrap_m = cv2.getPerspectiveTransform(dst, src)
```

Here's an example of the transformation:

Lane pixel detection

The next challenge is by using the transformed image, identify the lane line pixels. To accomplish this we'll use a method called "Peaks in a Histogram" where we analyze

the histogram of section of the image, window, and identify the peaks which represent the location of the lane lines.

To abstract this we've create a two classes:

- **SlidingWindow** - Where we represent "these" sections of the image where it's more likely to find a lane line. The class defines the top and bottom coordinates for the vertices of the rectangular window.
- **LaneLine** - This class represents each of the lane lines in the image, in this case the `left` and `right` lines. It calculates the equation of the curve of the line and returns `radius_of_curvature` and `camera_distance` from the center line. This class also includes all the assumptions like the width of the lanes, `3.7` meters, and the length of the lane in the image, `30` meters.

Image augmentation

In order to augment the video, and its respective frames, we've created a series of overlays to add additional information to the stream.

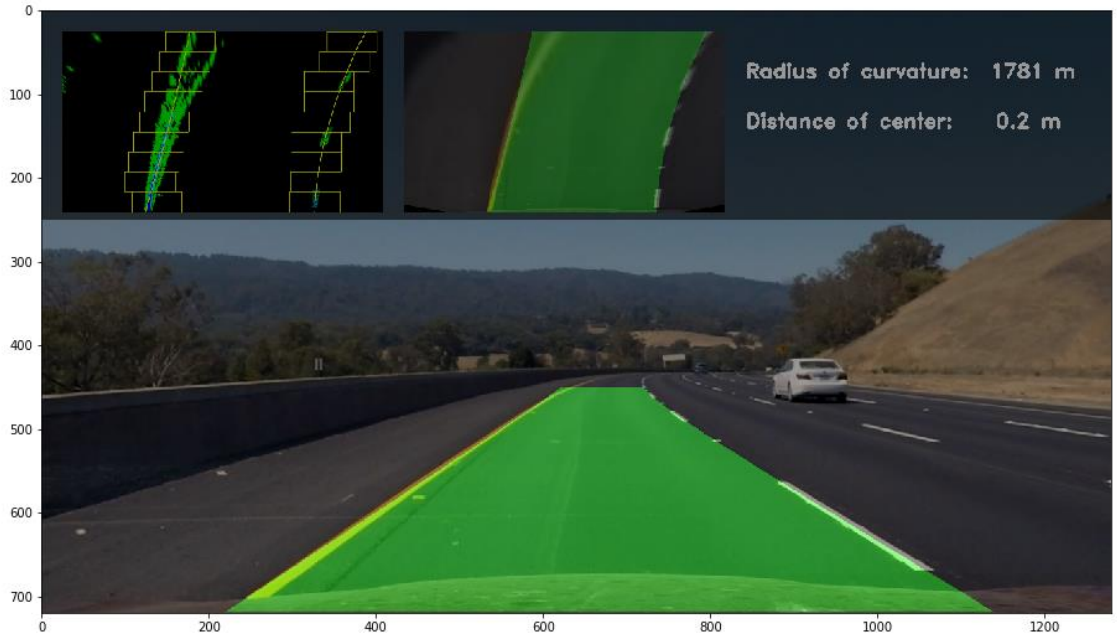
Top left corner - Adds a representation of the windows in the transformed image and identifies the lanes and its curvature

Top center - A "bird's eye" view with the lane identified

Top right corner - Metrics on the radius of curvature and distance to the center line

Lower two thirds - The actual image with the lane boundaries identified and colored for effect.

Here's an example augmented image:



Pipeline

Finally, we put it all together with the Pipeline class. We start by getting the edges of the lane lines

```
edges = stack_edges(frame)
```

Then we apply the perspective transform

```
transformed_egdes, _ = perspective_transform(edges)
```

Get the histogram of the transformed image which we'll use later to initialize the SlidingWindow objects

```
histogram = np.sum(transformed_egdes[int(self.height / 2):, :], axis=0)
```

Initialize the left and right SlidingWindow instances with a total of 9 per side.

```
for i in range(self.nwindows):
    # initialize each Window object
    if len(self.left_wins) > 0:
        l_x_center = self.left_wins[-1].x
        r_x_center = self.right_wins[-1].x
    else:
        l_x_center = np.argmax(histogram[:self.width // 2])
        r_x_center = np.argmax(histogram[self.width // 2:]) + self.width // 2

    left_win = SlidingWindow(y_low=self.height - i * window_height,
                             y_high=self.height - (i + 1) * window_height,
                             x_center=l_x_center)
```

```
    right_win = SlidingWindow(y_low=self.height - i * window_height,
                              y_high=self.height - (i + 1) * window_height,
                              x_center=r_x_center)
```

And lastly initialize each of the left and right laneLine instances

```
self.left_lane = LaneLine(nonzero[1][left_lane_inds], nonzero[0][left_lane_inds], self.height, self.width)
self.right_lane = LaneLine(nonzero[1][right_lane_inds], nonzero[0][right_lane_inds], self.height,
self.width)
```

We then run the pipeline for each of the test images:

```
for image_path in glob.glob('test_images/test*.jpg'):
    image = mpimg.imread(image_path)
    calibrated = calibrated_camera(image)
    pipeline = Pipeline(calibrated)
    overlay = pipeline.run(calibrated)
```

And finally, for each of the frames in the video:

```
from moviepy.editor import VideoFileClip

def build_augmented_video(video_path_prefix):
    output_video_name = 'videos/{}_augmented.mp4'.format(video_path_prefix)
    input_video = VideoFileClip("videos/{}.mp4".format(video_path_prefix))

    calibrated = calibrated_camera(input_video.get_frame(0))
    pipeline = Pipeline(calibrated)

    output_video = input_video.fl_image(pipeline.run)
```

We use moviepy lib to read each frame of the video and subsequently save each of the augmented frames into an "augmented" video.

Discussion

The pipeline works well for a "simple" video stream, because of slight variation in elevation, lighting/shadows or any steep bends.

If we analyze the slightly harder video, We can clearly see that the pipeline start to struggle, this is because this video adds features intense shadow into the lane and parallel to the lane lines. Close to the camera the model works relatively well but near the escape point we can see the lane detection starting to fail.

If we then look at the more challenging video, it's obvious that the model does not perform well. The lane boundary is "all over the place" and could not keep the car inside the lane. This

is most likely due to the nature of the road, single line for each traffic direction, steeper bends and a higher proximity to the side of the road.