Advanced Lane Lines Detection

Camera Calibration:
Most of the cameras with convex lenses are associated with some distortion at the edges of images. To mitigate this issue, camera had to be calibrated so that the output image can be undistorted. Following functionality is available in 'camera_calibration.py' file.

- I've made use of different images of a chessboard (from various angles) against a flat surface for calibration purposes.
- Using OpenCV's 'cv2.findChessboardCorners()' function, corners of chessboard are identified. To visualize the detected corners, OpenCV provides a function called 'cv2.drawChessboardCorners()'.
- I computed the object and image points based on detected corners, which are in-turn fed into a function 'cv2.calibrateCamera(...)' to get the calibration metrics like distortion coefficients and transformation matrix along with rotation and translational vectors.
- Finally, I used matrix, distortion coefficients along with the image to feed into 'cv2.undistort(…)' function to get the undistorted image.

Detect Lines & Curvature:
To identify the lane lines, I made use of different gradient and color thresholding techniques to detect edges in an image. Perspective transform is applied to a particular region (polygon to fit the lane) of the resultant image from previous step to get a birds-eye view of the road. A Histogram is used to identify the location of lane lines along the Y direction and identify the 2 peaks corresponding to the left and right lines of a line. Then, a 'sliding window' approach is used to determine the curvature of the lane and identify the polygon area for lane. Once a lane has been identified, then we apply an inverse perspective transform on the image to be able to draw the lane on original image. Following steps describe in detail the pipeline employed

- Employed different gradients using OpenCV's 'Sobel' operator to detect edges in the image. Seems like this is the underlying method used for Canny edge detection used in the 'Finding Lane Lines' project. Following functions are included in gradient.py file of 'src' directory. Various thresholds (for each of the gradients) have been applied to filter out the noise and to identify more defined edges in the image
    - Gradient along X/Y axis (Low: 20, High: 100)
    - Gradient of magnitude (Low: 30, High: 100)
    - Gradient of direction (Low: 0.3, High: 1.1)
- When computing the above gradients, given image has to be converted to gray scale before we apply Sobel operator. Because of this, we lose valuable color information that might be useful in detecting lane lines. In order for lane lines to be detected accurately under different conditions of lighting and shadow, different color spaces (other than RGB) like HLS, HSV have been

explored. Among them, HLS color space seemed to do a better job in retaining the color information. I've chosen one of the channels ('S') in the HLS space that can retain the helpful color information, and thresholds (Low: 170, High: 255) have been applied to filter out the noise. 'color.py' in 'src' contains function called 'hls_color_binary(…)' that returns a color binary.

- In the pipeline, we combine both gradient and color thresholds to identify the edges in image. Even after applying thresholds a lot of noise exists in terms of tress, leaves and other objects that can have well defined edges. Our region of interest is on the road, in front of car, that can be marked using a polygon and mask out the rest of the region in image. For given image of size (1280, 720), I've used a polygon with following vertices:
    - Bottom Left: (100, 720)
    - Top Left: (600, 410)
    - Top Right: (690, 410)
    - Bottom Right: (1230, 720)
- We next apply Perspective (warp) transform using OpenCV's 'warpPerspective(…)' to the resultant masked image to get a birds-eye (top-down) image. In order to get a transform matrix, we need to choose source and destination points from original and transformed images respectively. This functionality can be found in perspective_transform.py. Following coordinates have been used
    - Source points:
        - Bottom Left: (200, 700)
        - Top Left: (590, 450)
        - Top Right: (700, 450)
        - Bottom Right: (1130, 700)
    - Destination points:
        - Bottom Left: (300, 720)
        - Top Left: (320, 0)
        - Top Right: (980, 0)
        - Bottom Right: (980, 720)
- Warped image obtained from the previous step has been used to identify the lane lines in 2 steps:
    - Using histogram of white pixel density (along X-axis), identify the 2 peaks associated with a lane's left and right lines.
    - Sliding window is used to identify the curvature of lane lines by traversing from the bottom of image with the above 2 peaks as starting points. Entire image is divided into small windows along the height into N windows and with a width of 100px. Note that 2 sets of windows are used for 2 lane lines. Next, we loop through each of the windows and identify the area where there is highest density of white pixels and extract left and right line pixel positions. Using these values, a second order polynomial fit is made to determine curvature of the lane lines.

    - Using the above polynomial fit values, we draw polygon to identify the lane area. Radius of curvature, displayed on the output video is also computed using these values. Using the polygon identified in previous step and applying inverse Perspective transform on the warped image, position of car (from center) in the lane can be determined.
- All of the above steps are implemented in pipeline.py file.

Discussion:
This has been very interesting and exciting project to work on. Learned a lot about applying computer vision techniques for the self-driving car application. Pipeline described above works for the most of the frames except when there is change in lighting conditions/shadow. I have made use of previous 10 values to smoothen out the rendering of lane detection algorithm. For some of the frames where the detection seems faulty, discarded that value and use the best-fit value computed from previous detections.