Vehicle Detection Project

The goals / steps of this project are the following:

- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
- Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
- Note: for those first two steps don't forget to normalize your features and randomize a selection for training and testing.
- Implement a sliding-window technique and use your trained classifier to search for vehicles in images.
- Run your pipeline on a video stream (start with the test_video.mp4 and later implement on full project_video.mp4) and create a heat map of recurring detections frame by frame to reject outliers and follow detected vehicles.
- Estimate a bounding box for vehicles detected.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

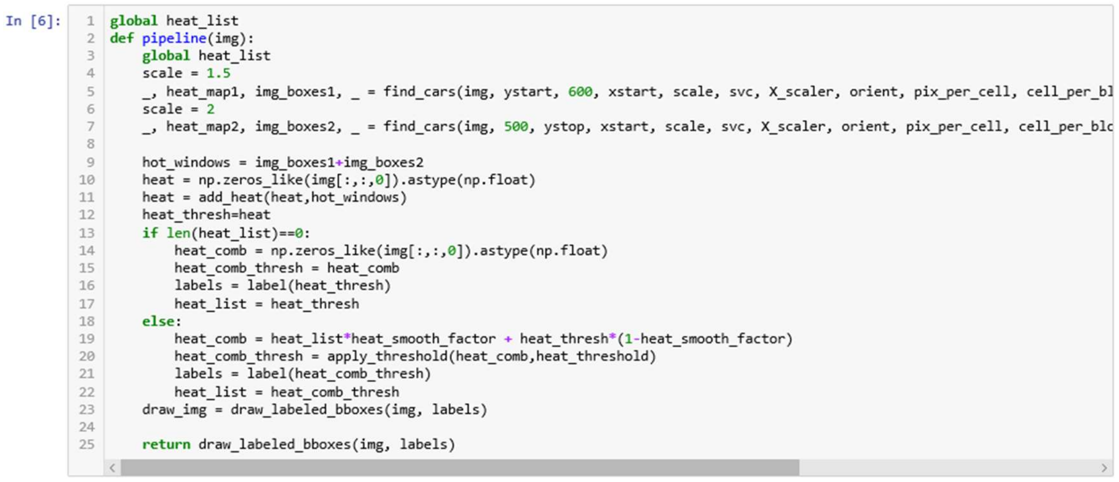
Video Implementation

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.)

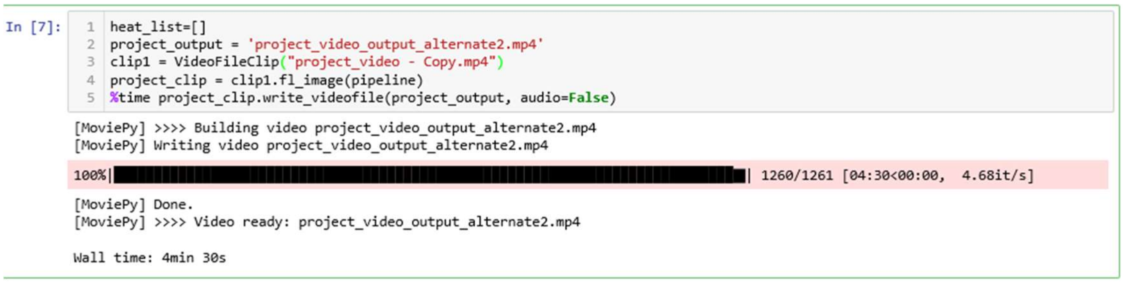
Here's a link to my video result:

https://youtu.be/D4QSVajtDyc

1. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.



```
In [6]:
global heat_list
def pipeline(img):
    global heat_list
    scale = 1.5
    _, heat_map1, img_boxes1, _ = find_cars(img, ystart, 600, xstart, scale, svc, X_scaler, orient, pix_per_cell, cell_per_bl
    scale = 2
    _, heat_map2, img_boxes2, _ = find_cars(img, 500, ystop, xstart, scale, svc, X_scaler, orient, pix_per_cell, cell_per_blo

    hot_windows = img_boxes1+img_boxes2
    heat = np.zeros_like(img[:,:,0]).astype(np.float)
    heat = add_heat(heat,hot_windows)
    heat_thresh=heat
    if len(heat_list)==0:
        heat_comb = np.zeros_like(img[:,:,0]).astype(np.float)
        heat_comb_thresh = heat_comb
        labels = label(heat_thresh)
        heat_list = heat_thresh
    else:
        heat_comb = heat_list*heat_smooth_factor + heat_thresh*(1-heat_smooth_factor)
        heat_comb_thresh = apply_threshold(heat_comb,heat_threshold)
        labels = label(heat_comb_thresh)
        heat_list = heat_comb_thresh
    draw_img = draw_labeled_bboxes(img, labels)

    return draw_labeled_bboxes(img, labels)
```



```
In [7]:
heat_list=[]
project_output = 'project_video_output_alternate2.mp4'
clip1 = VideoFileClip("project_video - Copy.mp4")
project_clip = clip1.fl_image(pipeline)
%time project_clip.write_videofile(project_output, audio=False)

[MoviePy] >>>> Building video project_video_output_alternate2.mp4
[MoviePy] Writing video project_video_output_alternate2.mp4

100%|████████████████████████████████████████| 1260/1261 [04:30<00:00,  4.68it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: project_video_output_alternate2.mp4

Wall time: 4min 30s
```

The code shown above is where I take a recent list of hot windows that have been estimated based on sliding window search and a trained classifier. Steps to eliminate false positives and multiple overlapping bounding boxes are:

1. Apply a threshold to the current hot window list

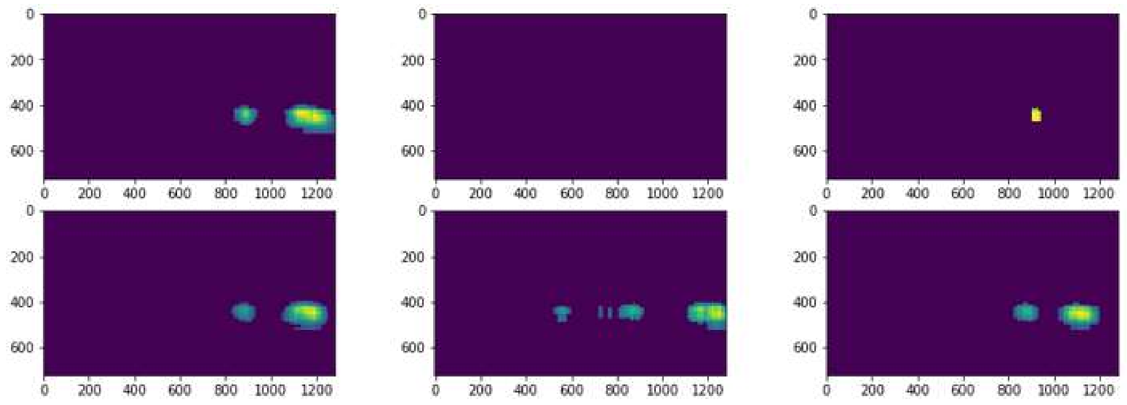
2. Add the current set of hot windows to a list

3. I chose to smooth out the multiple windows based on 5 previous frames. This is done with a heat_smooth_factor (0.5) as shown above.

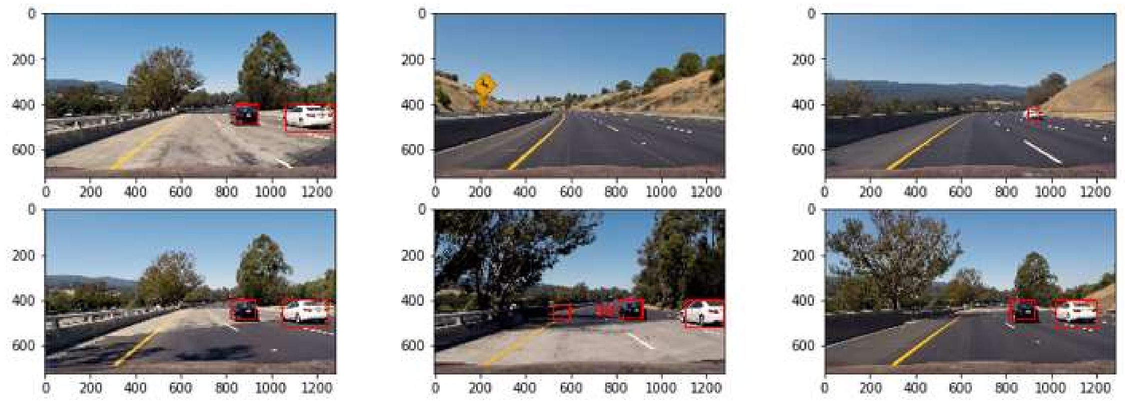
4. After smoothing the window position, I apply the threshold to eliminate the overlapping and false positive windows.

5. The windows are then drawn on the image with draw_labeled_bboxes.

Below are the heatmaps achieved with my classifier



Below are the bounding boxes drawn based on the heatmap and label function:



HOG (Histogram of Oriented Gradients):

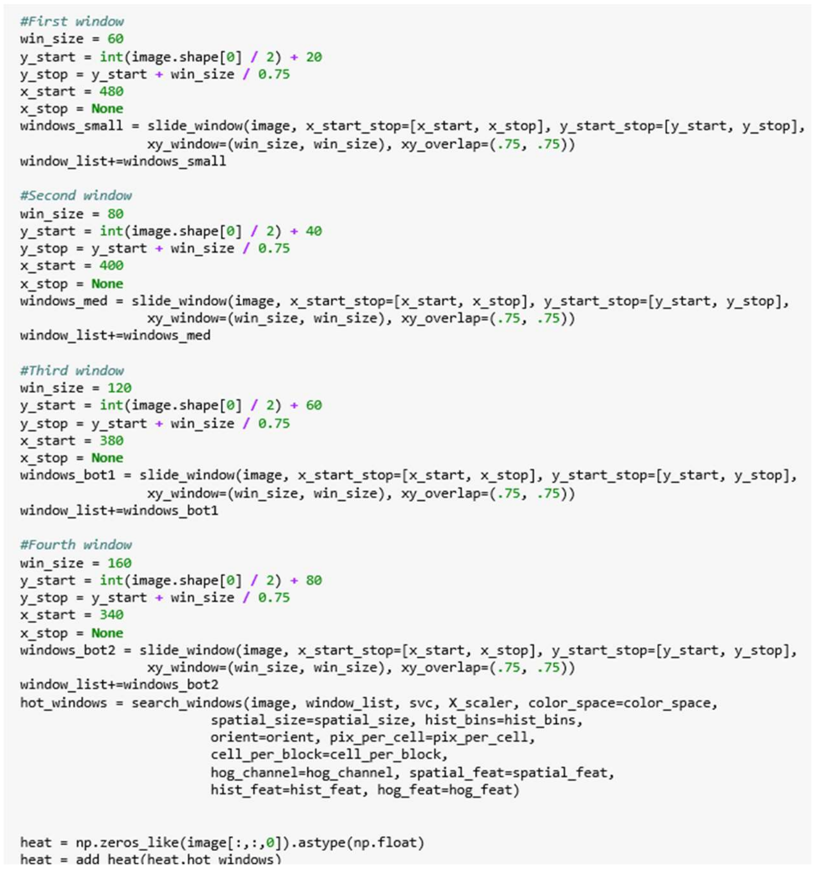
My get_hog_features is implemented in Functions.py line 50 to 63. During training, hog features are extracted using extract_features function lines 91 to 100 in functions.py. During video pipeline process, find_cars calls get_hog_features directly to extract hog features once before subsampling them as per sliding window function coordinates.

- To start of with, I tested various LinearSVC and SVC with rbf trainings using only 1 channel fed to the HOG feature extractor function hog.sdf
    - When I trialed only channel 0, I got very sketchy performance with actual vehicle detection.
    - With 2 channels, I got only ~50% of the detections correct.
- Based on all the discussion on slack and my own tests, only feeding all the channels to extract hog features gave me above 99% test accuracy during training.
- I also tested using orientations setting of 9, 12 and 32.

    - For all orient settings there was no substantial improvement in the test accuracy.
    - But during video pipeline runs, orient of 9 missed out detecting the black car (in combination with using only 1 channel for hog) that comes into the frame. And as I increased the orient alone while sticking with single channel, the accuracy of recognizing the black car improved but not entirely.
    - Eventually in combination with feeding all the channels to hog, orient of 32 proved to be the most reliable for video pipeline.
- I did not test a lot of variations of pix_per_cell and cell_per_block. Initially all my testing was done at pix_per_cell of 8. The video processing time for this was 11 minutes with finding_cars function. When the pix_per_cell was upped to 16, the time to process video was 4min 30 sec. But the difference in quality of vehicle detection was not substantial. I settled on pix_per_cell of 16. This variable has direct impact on the nxblocks and nyblocks variables (I tried this in the excel sheet first to gauge impact – the extract from the sheet is shown in the sliding window section).
    - I left cell_per_block at 2 for overlap of 75% while sliding windows across the image.

Sliding Window:

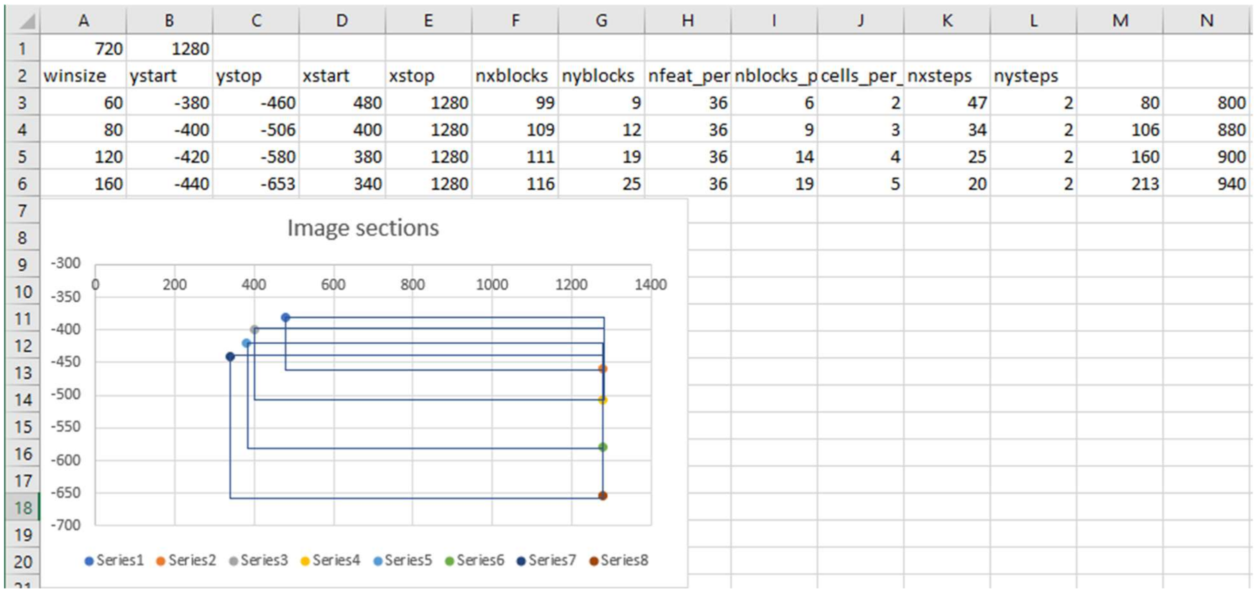
This is where I spent a lot of time. I initially wanted to see how fast the video pipeline will be if I implemented the slide_window and search_window algorithms where the hog features are extracted for each image snippet. The code for this is now included in the submission. Here is an extract:



```python
#First window
win_size = 60
y_start = int(image.shape[0] / 2) + 20
y_stop = y_start + win_size / 0.75
x_start = 480
x_stop = None
windows_small = slide_window(image, x_start_stop=[x_start, x_stop], y_start_stop=[y_start, y_stop],
                xy_window=(win_size, win_size), xy_overlap=(.75, .75))
window_list+=windows_small

#Second window
win_size = 80
y_start = int(image.shape[0] / 2) + 40
y_stop = y_start + win_size / 0.75
x_start = 400
x_stop = None
windows_med = slide_window(image, x_start_stop=[x_start, x_stop], y_start_stop=[y_start, y_stop],
                xy_window=(win_size, win_size), xy_overlap=(.75, .75))
window_list+=windows_med

#Third window
win_size = 120
y_start = int(image.shape[0] / 2) + 60
y_stop = y_start + win_size / 0.75
x_start = 380
x_stop = None
windows_bot1 = slide_window(image, x_start_stop=[x_start, x_stop], y_start_stop=[y_start, y_stop],
                xy_window=(win_size, win_size), xy_overlap=(.75, .75))
window_list+=windows_bot1

#Fourth window
win_size = 160
y_start = int(image.shape[0] / 2) + 80
y_stop = y_start + win_size / 0.75
x_start = 340
x_stop = None
windows_bot2 = slide_window(image, x_start_stop=[x_start, x_stop], y_start_stop=[y_start, y_stop],
                xy_window=(win_size, win_size), xy_overlap=(.75, .75))
window_list+=windows_bot2
hot_windows = search_windows(image, window_list, svc, X_scaler, color_space=color_space,
                        spatial_size=spatial_size, hist_bins=hist_bins,
                        orient=orient, pix_per_cell=pix_per_cell,
                        cell_per_block=cell_per_block,
                        hog_channel=hog_channel, spatial_feat=spatial_feat,
                        hist_feat=hist_feat, hog_feat=hog_feat)


heat = np.zeros_like(image[:,:,0]).astype(np.float)
heat = add_heat(heat,hot_windows)
```

The video processed with this method is pretty much the same as what I got from my final find_cars function but took almost 1hr 30 minutes to process!!!

In this method as can be seen, I am getting a list of windows for sections of the images shown below in an excel chart I made:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	720	1280												
2	winsize	ystart	ystop	xstart	xstop	nxblocks	nyblocks	nfeat_per	nblocks_p	cells_per_	nxsteps	nysteps		
3	60	-380	-460	480	1280	99	9	36	6	2	47	2	80	800
4	80	-400	-506	400	1280	109	12	36	9	3	34	2	106	880
5	120	-420	-580	380	1280	111	19	36	14	4	25	2	160	900
6	160	-440	-653	340	1280	116	25	36	19	5	20	2	213	940

Image sections
-300
-350
-400
-450
-500
-550
-600
-650
-700
0
200
400
600
800
1000
1200
1400
● Series1 ● Series2 ● Series3 ● Series4 ● Series5 ● Series6 ● Series7 ● Series8

This was done to work out exact dimensions of the image to be fed to the slide_window function. This excel sheet provided with all the required ystart, ystop and xstart, xstop variable values along with a visualization of which sections of the image are being used for feature extraction. The window list extracted from this is fed to the search_window to obtain the hot_window list which is then processed for rejecting false positives and duplicate detections using the label function.

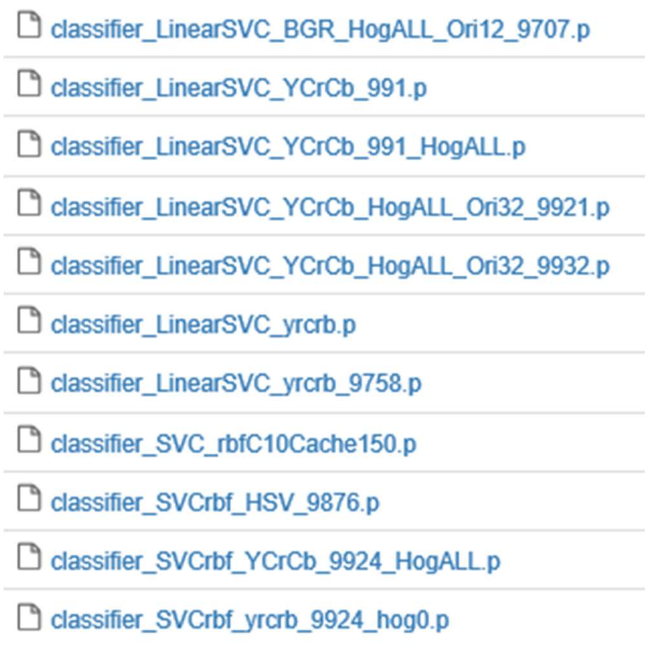
Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Mostly time was spent trying to improve the efficiency of the algorithm to speed up the pipeline. Initially the same result took about 1.5 hrs. using the sliding window and search_window functions! The efficiency was improved by using the find_cars function where the HOG features are extracted beforehand for the whole image and later subsampled while prediction is being done.

Another substantial problem was whether to use the YCrCb or RGB/BGR color spaces. Both the color spaces have their pluses and minuses. Eventually based on many hrs spent training and retraining, I decided to go with YCrCb although BGR color space was a very close second.

I also tested SVC with rbf and Linear SVC classifiers. The SVC takes a lot longer to process the image as compared to the LinearSVC. I settled on LinearSVC for speed and efficiency despite of slightly lower test accuracy.



classifier_LinearSVC_BGR_HogALL_Ori12_9707.p
classifier_LinearSVC_YCrCb_991.p
classifier_LinearSVC_YCrCb_991_HogALL.p
classifier_LinearSVC_YCrCb_HogALL_Ori32_9921.p
classifier_LinearSVC_YCrCb_HogALL_Ori32_9932.p
classifier_LinearSVC_yrcrb.p
classifier_LinearSVC_yrcrb_9758.p
classifier_SVC_rbfC10Cache150.p
classifier_SVCrbf_HSV_9876.p
classifier_SVCrbf_YCrCb_9924_HogALL.p
classifier_SVCrbf_yrcrb_9924_hog0.p

These pickle dumps are from various permutations and combinations I trialed.

I also tested some variations of smoothing out the hot windows before settling on the implementation as given in the Video pipeline.

Eventually, after I had completed the vehicle detection portion, I fed the video to my previous project for lane detection. The link shared above is the result of this.

My next task is to use the LeNet neural network to train the model and see how that performs as compared to LinearSVC.