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.

Histogram of Oriented Gradients (HOG)

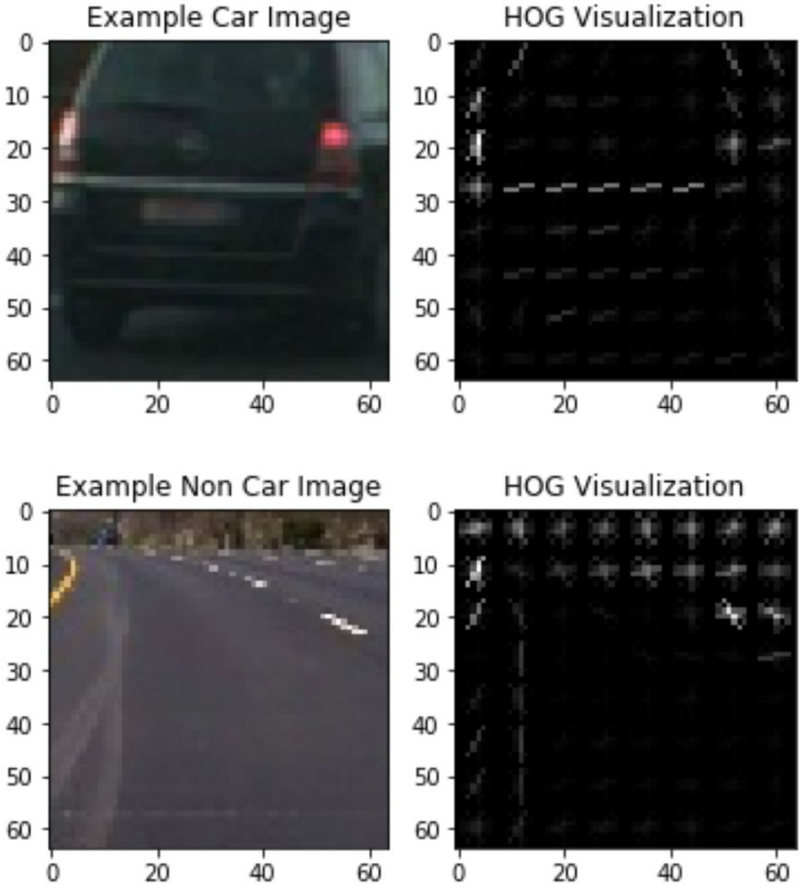
1. Explain how (and identify where in your code) you extracted HOG features from the training images.

The code for this step is contained in the first seven code cells of the IPython notebook CarND-Vehicle_Detection.ipynb and relies on skimage's hog function

I started by reading in all the `vehicle` and `non-vehicle` images. All images from the GTI dataset were used along with images cropped from the test video after resizing to 64x64 and classifying them as car and non car. All images were converted to .jpg.

Here are a few examples of each of the `vehicle` and `non-vehicle` classes:

After doing some testing with color histogram (cell 6) and HOG features I decided to focus on HOG features since they seemed to provide a reliable classifier. I explored different color spaces and different `skimage.hog()` parameters (`orientations`, `pixels_per_cell`, and `cells_per_block`). The variations of these tests are listed in the following section: Below is a sample of the HOG feature extractor with default settings.



2. Explain how you settled on your final choice of HOG parameters.

I tried various combinations of parameters. I started by exploring all the color spaces and channels with nine orientations and eight pixels per cell and two cells per block. All of the color spaces provided good results but YUV had the best. In all cases using all three channels provided better accuracy than any single channel as expected. After selecting YUV and all three channels, I scanned through various orientans and found they did not have much impact but settled on 12 before scanning through various pixels per cell and cells per block. After this extensive search I found good balance of speed and accuracy using eight pixels per cell and 4 cells per block

The table below shows all parameters tested.

Color Space	Orientations	Pixels per Cell	Cells per Block	Channels	Accuracy	Extract Time (seconds)	Train Time (seconds)	Predict Time (seconds for 1000 images)
RGB	9	8	2	0	90.65%	34.81	11.54	0.00399
RGB	9	8	2	1	90.89%	36.88	12.9	0.00401
RGB	9	8	2	2	91.16%	35.34	12.66	0.00401
RGB	9	8	2	3	92.03%	113.46	29.99	0.00899
HSV	9	8	2	0	83.23%	37.71	27.97	0.003
HSV	9	8	2	1	90.38%	35.67	12.81	0.00299
HSV	9	8	2	2	91.56%	37.38	12.24	0.00301
HSV	9	8	2	3	94.38%	98.98	27.55	0.01
LUV	9	8	2	0	91.95%	36.92	11.09	0.00301
LUV	9	8	2	1	85.19%	30.81	13.69	0.00401
LUV	9	8	2	2	85.59%	34.36	16.27	0.00302

LUV	9	8	2	3	95.44%	93.9	22.47	0.01001
HLS	9	8	2	0	83.90%	35.98	23.69	0.00301
HLS	9	8	2	1	91.32%	30.84	11.28	0.003
HLS	9	8	2	2	91.56%	34.25	13.86	0.003
HLS	9	8	2	3	95.21%	98.19	23.88	0.00901
YUV	9	8	2	0	91.08%	35.59	10.76	0.00299
YUV	9	8	2	1	84.13%	29.29	18.51	0.00399
YUV	9	8	2	2	85.59%	33.13	14.64	0.00399
YUV	9	8	2	3	95.80%	92.59	7.84	0.00901
YCrCb	9	8	2	0	92.14%	35.59	11.66	0.003
YCrCb	9	8	2	1	86.57%	29.55	13.13	0.00299
YCrCb	9	8	2	2	84.09%	31.99	17.25	0.00399
YCrCb	9	8	2	3	95.13%	91.02	20.94	0.01201
YUV	6	8	2	3	94.38%	75.54	13.03	0.00601
YUV	7	8	2	3	93.75%	85.78	7.8	0.00802
YUV	8	8	2	3	95.76%	93.12	5.99	0.00902
YUV	9	8	2	3	95.48%	96.72	6.04	0.01202
YUV	10	8	2	3	95.64%	96.01	5.22	0.01
YUV	11	8	2	3	95.76%	92.03	29.36	0.01101
YUV	12	8	2	3	95.56%	91.16	5.85	0.01201
YUV	13	8	2	3	95.76%	96.44	31.62	0.01402

YUV	14	8	2	3	95.76%	87.69	6	0.01702
YUV	12	6	2	3	95.64%	122.61	8.82	0.02601
YUV	12	7	2	3	95.99%	100.2	7.02	0.016
YUV	12	8	2	3	96.19%	98.41	6.87	0.01201
YUV	12	9	2	3	95.05%	95.19	6.17	0.01002
YUV	12	10	2	3	95.29%	76.63	14.01	0.00651
YUV	12	12	2	3	95.95%	44.81	7.07	0.00399
YUV	12	16	2	3	96.58%	38.92	3.05	0.003
YUV	12	20	2	3	96.43%	32.25	2.66	0.00199
YUV	12	24	2	3	93.95%	25.67	3.38	0
YUV	12	28	2	3	95.64%	27.26	2.44	0
YUV	12	32	2	3	96.66%	29.24	1.77	0
YUV	12	8	3	3	97.21%	70.08	7.78	0.02602
YUV	12	12	3	3	96.50%	41.99	7.65	0.00601
YUV	12	16	3	3	97.45%	36.46	2.84	0.002
YUV	12	20	3	3	96.82%	28.59	1.67	0.00101
YUV	12	8	4	3	96.98%	58.35	11.09	0.03203
YUV	12	12	4	3	96.54%	33.9	5.53	0.00401
YUV	12	16	4	3	97.09%	31.16	1.15	0.001

3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

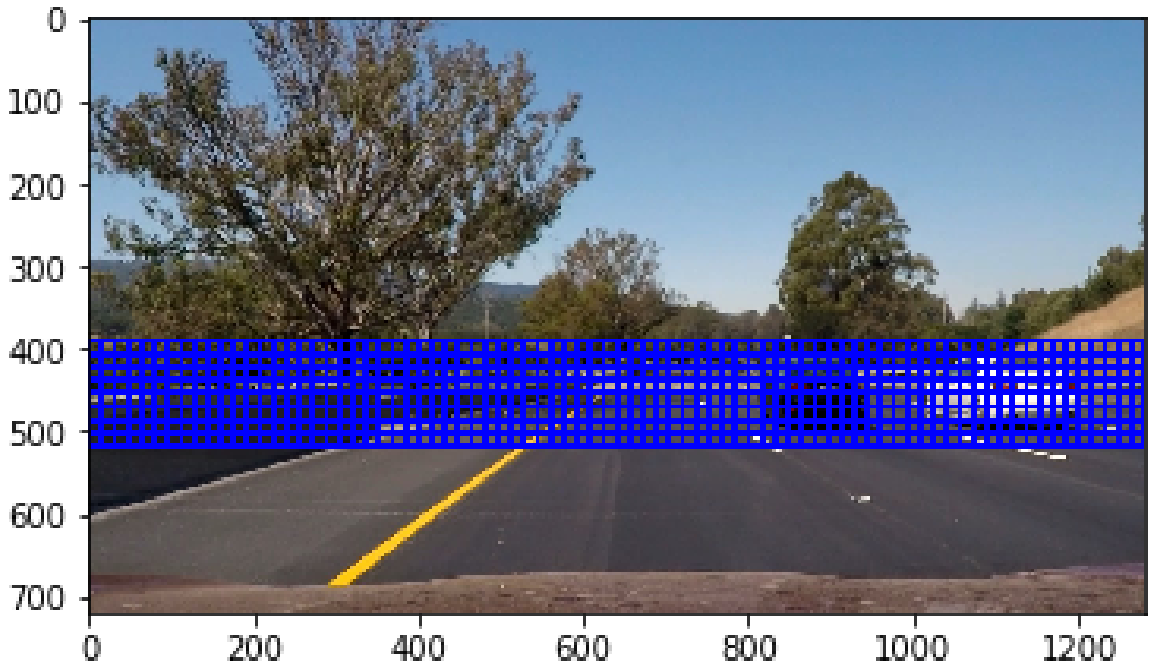
I trained a classifier using LinearSVC() in cell 9 of the code. This used a basic fit function on the training data. I only used HOG features.

Sliding Window Search

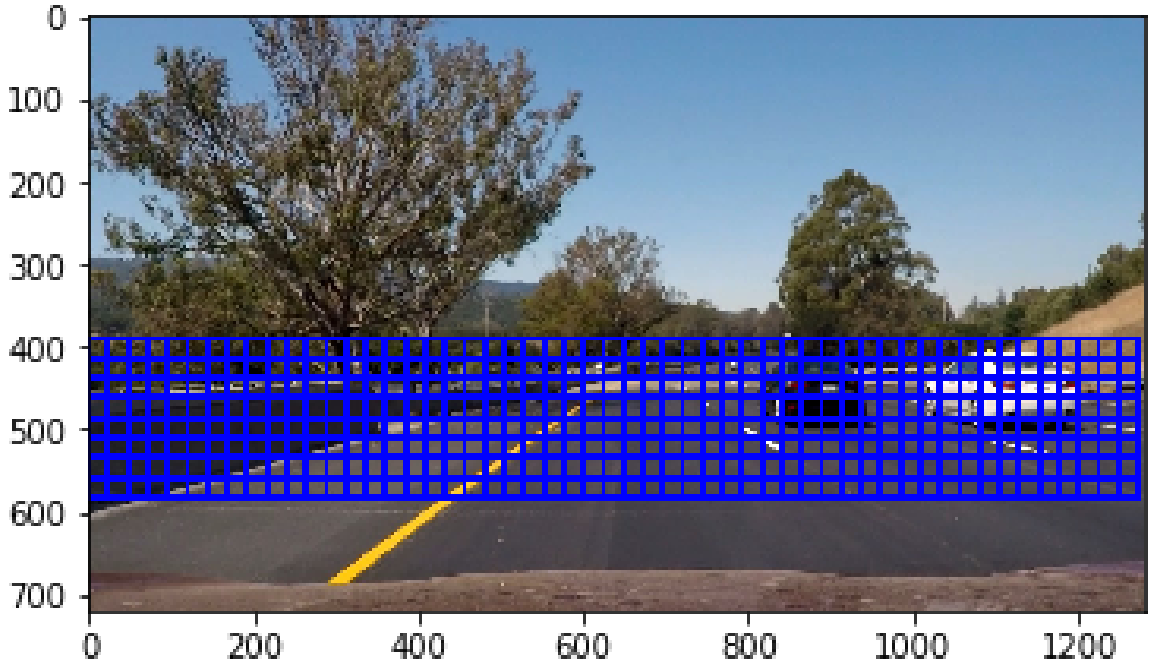
1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

I tested various scales and ranges of y position and ended up using a 1.0 scale in the upper portion of the road where the more distant cars were. I used a 1.5 scale, 2.0 scale and a 3.0 scale. Each time I increased the scale, I moved the scan window down to match the apparent scale of cars in the field of view. Each of these used 75% overlap since it provided good results while maintaining good processing speed. The images below show where all the test windows are at each scale.

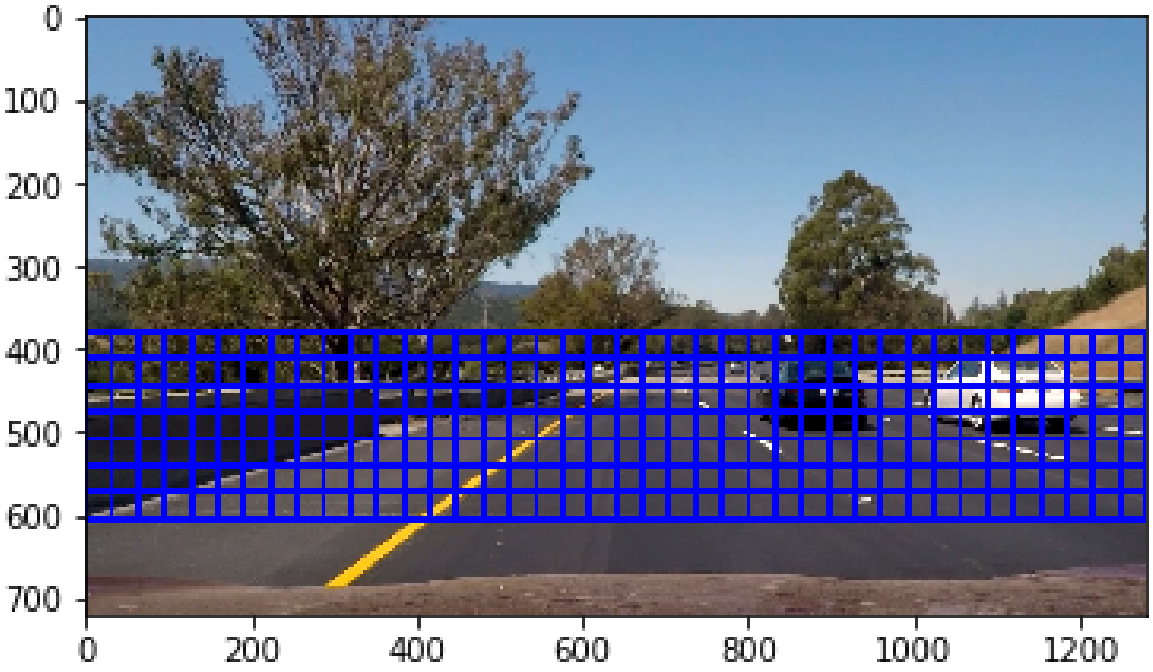
Scale = 1.0



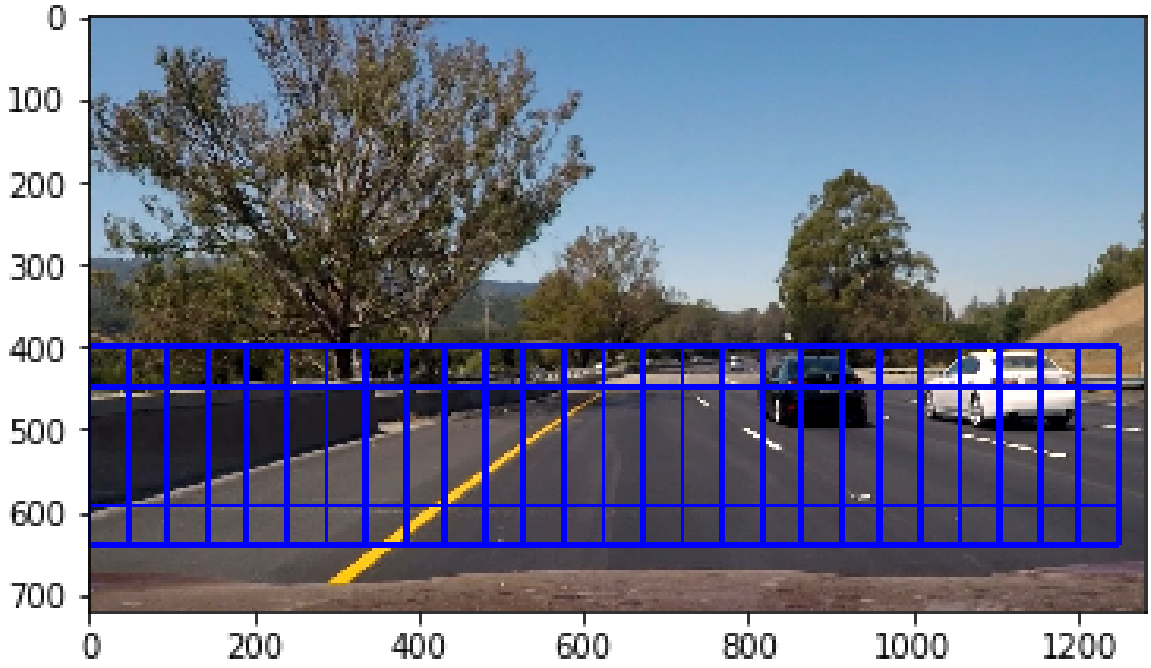
Scale = 1.5



Scale = 2

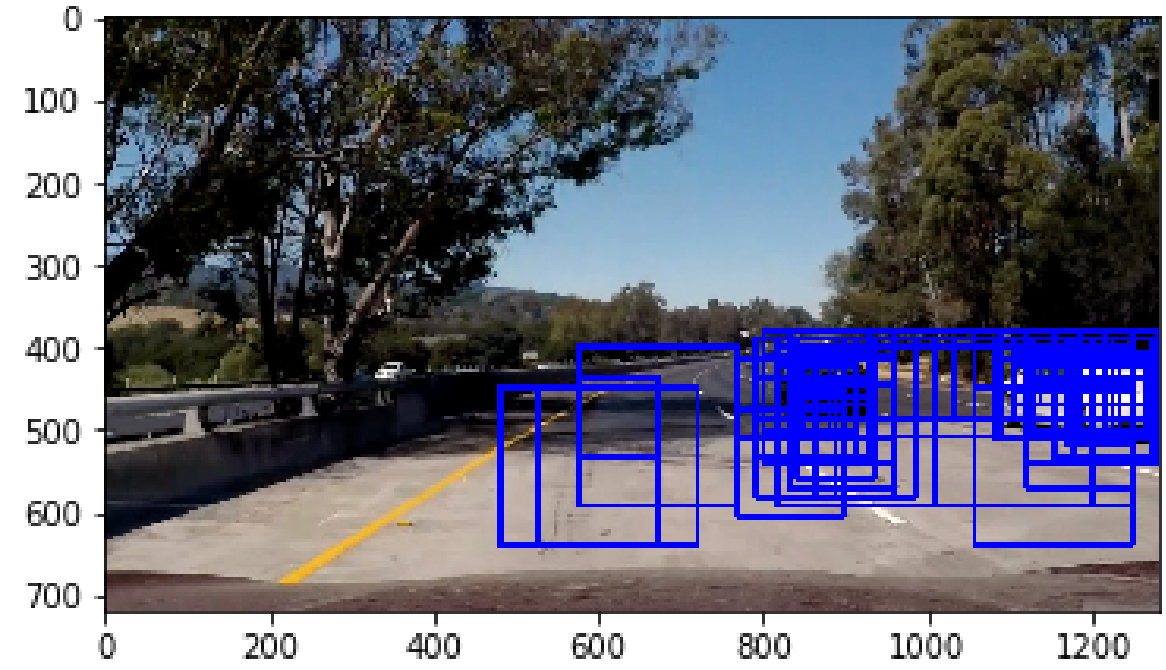


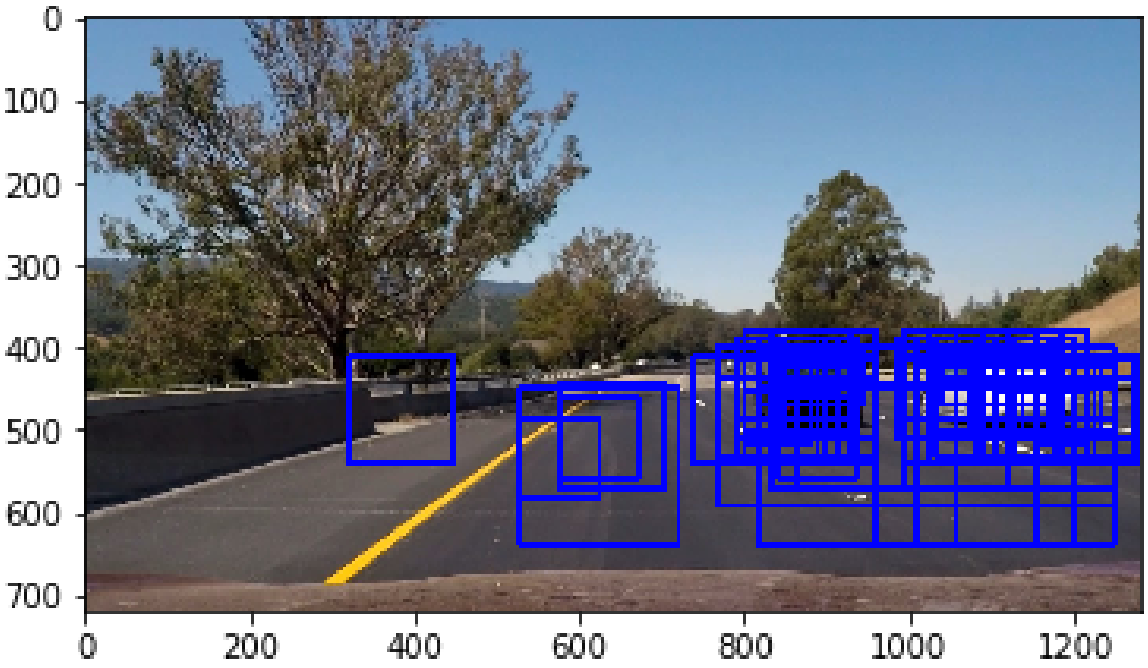
Scale = 3



2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

I use all three channels of YUV for my HOG features. I found four scales with 75% overlap provided a fast but reliable detector. Below are two example images with bounding boxes found by the pipeline. This is before heat map filtering. Most of the optimization came from testing many variations of features and also by cleaning the data and using false positives around the yellow lane line to retrain the SVM so it would correctly classify them as non cars.





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 youtube link https://youtu.be/UWv1Ucp1hC8

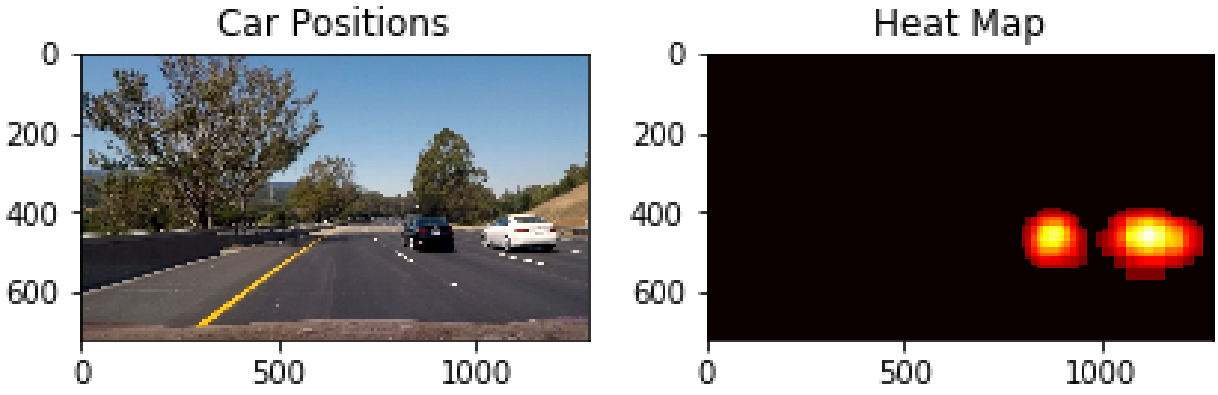
There is also a copy on the Github (final.mp4)

2. 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.

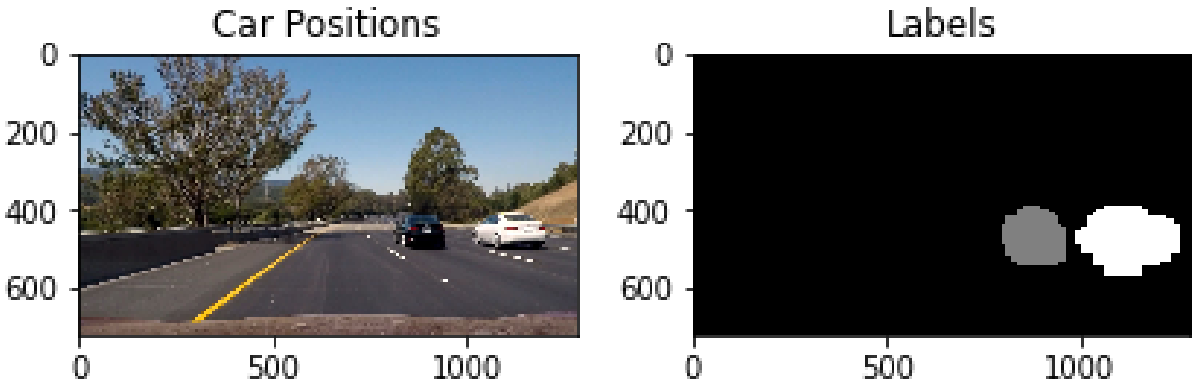
In cells 17-20 I created a heat map of the car positions. This was averaged/summed over 20 frames to reduce noise and excessive jittering of the windows. The averaged heatmap was then thresholded and fed into scipy's label function to pick out individual blobs which represent each car. Bounding boxes were drawn around the blobs and displayed on each frame of the video after the 20th frame.

Below is a sample image along with the heatmap, labels and bounding boxed.

Here is a sample frame and its corresponding heatmap:



Here is the output of scipy.ndimage.measurements.label() on the integrated heatmap :



Here the resulting bounding boxes are drawn onto the frame:

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?

The first time I ran the pipeline on the video I had a lot of false positives on the yellow lane line and trouble reliably detecting the white car. I collected several images of the cars in the video and the non car areas from the video and added them to the training data and retaught the classifier. This provided much better results. The bridges

continue to cause the classifier to falsely detect a car but the overall results were reasonable for this type of detection.

State of the art appears to be neural network based and algorithms like YOLO look like they are very robust. There are also many implementations of per-pixel classification/images segmentation that can not only find cars but identify many other items and features in the video in real time. The are areas I would like to explore further.