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 second code cell of the IPython notebook under the function by the name of get_hog_features. The hog function from skimage.feature library has been used to obtain the hog image and the features.

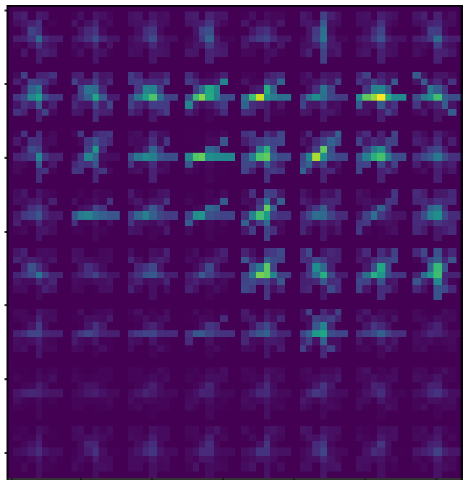
I started by reading in all the vehicle and non-vehicle images. Here is an example of one of each of the vehicle and non-vehicle classes:

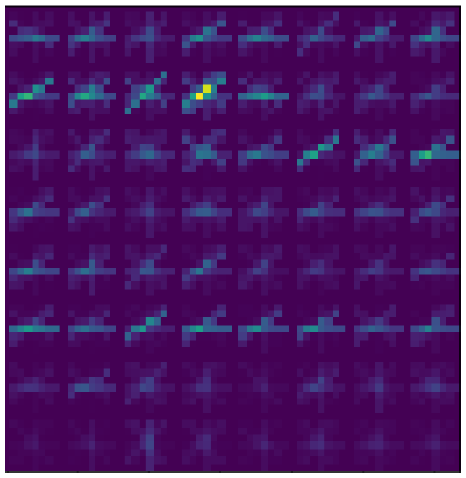
CAR

NOT CAR

I then explored different color spaces and different skimage.hog() parameters (orientations, pixels_per_cell, and cells_per_block).

Here is an example using the YCrCb color space and HOG parameters of orientations=8, pixels_per_cell=8 and cells_per_block=2. The images for car and not car are the pics shown above.





Extracted Features:

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

I tried various combinations of parameters for different color spaces and reported the accuracy and runtime for the classifier and chose the best set.

Colorspace	Orient	Pix per cell	Cell per block	Hog channel	Accuracy (%)	Time to run (secs)
RGB	6	8	2	All	98.00	10.97
RGB	8	8	2	All	97.75	14.55
RGB	10	8	2	All	98.34	18.01
YCrCb	6	8	2	All	99.04	8.10
YCrCb	8	8	2	All	99.04	10.96
YCrCb	10	8	2	All	99.10	2.50
HSV	6	8	2	All	98.73	8.72
HSV	8	8	2	All	99.04	2.12
HSV	10	8	2	All	99.35	2.99
LUV	6	8	2	All	98.73	9.94
LUV	8	8	2	All	98.96	13.55
LUV	10	8	2	All	98.99	18.16

YCrCb was chosen for a good speed and a very good accuracy.

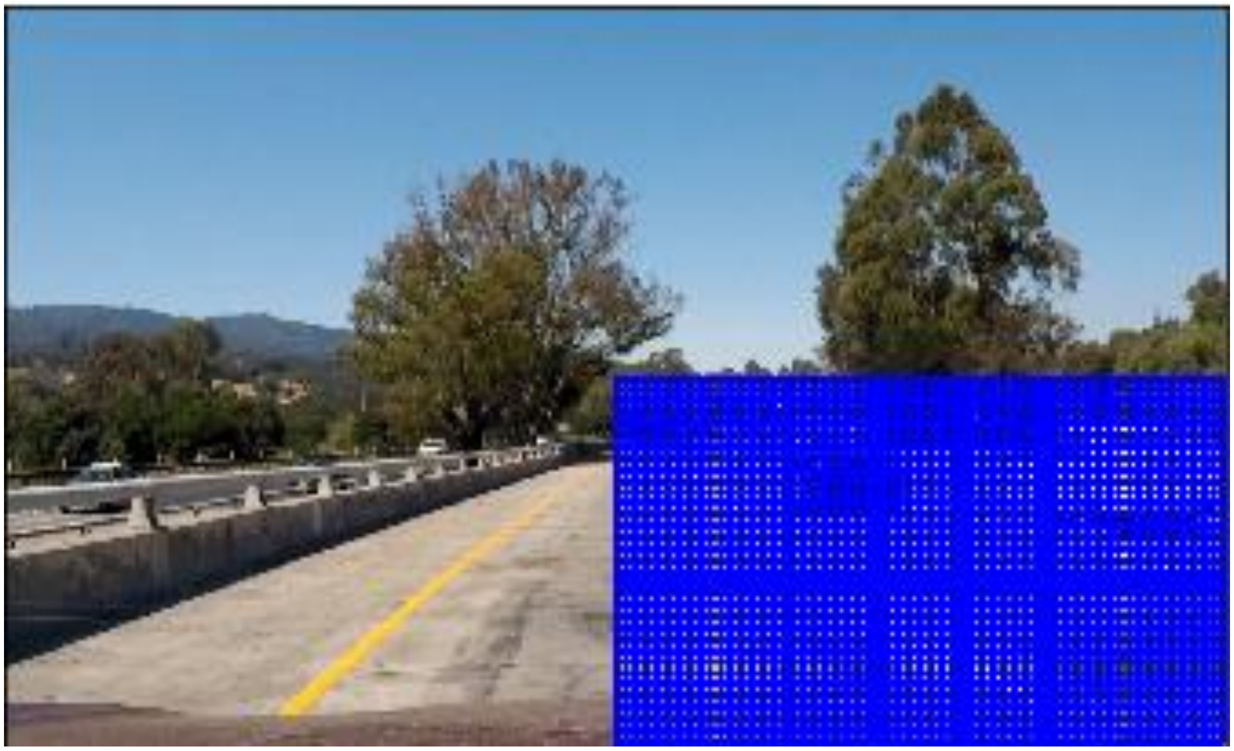
3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

A linear SVM has been used as the classifier. The code for it is in the cell5 of the IPython notebook. The dataset provided was divided into car and non-car data. The next step was to extract the features out of the car and non-car data respectively using the extract_features function which were stacked and were converted to scalar features which was fed into the classifier.

Sliding Window Search

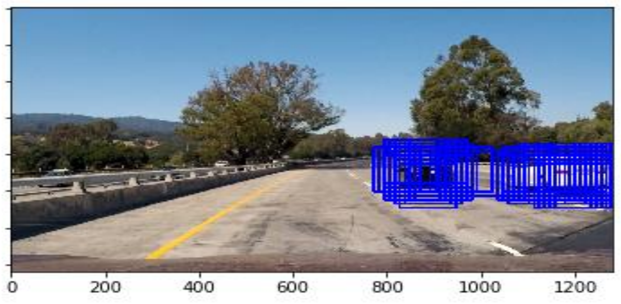
1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

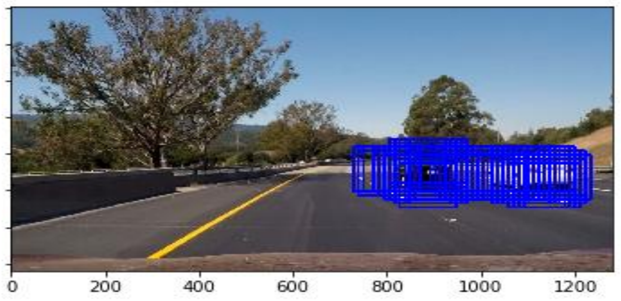
I defined the max and minimum limit on the y coordinate for searching in the window as we don't want to search for cars in the sky. The y_start was taken to be half of the height of the image (img.shape[0]*0.5)and y_stop was taken to be the lowest point in the image (image.shape[0]). An overlap of 0.9 was taken as I wasn't able to achieve a good result with a smaller overlap on the project video when the 2 cars in the video came closer. Two sizes of window were selected for the outlining box for vehicles: (100,100) and (120,120) to cater for varying size of boxes when the cars are close by and when they get far, the box size reduces.



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?

Ultimately I searched on two scales using YCrCb 3-channel HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. Here are some example images:





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.)

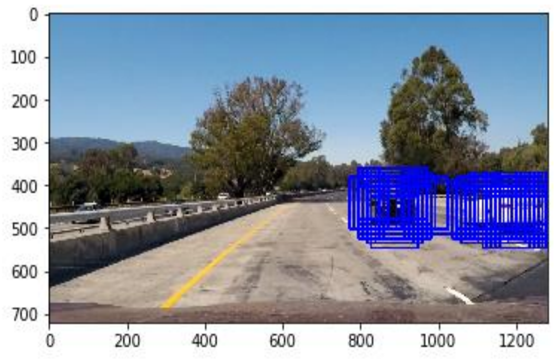
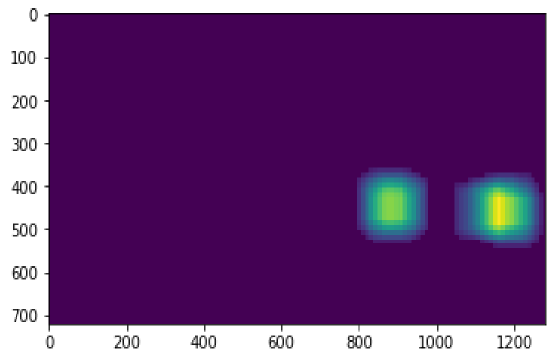
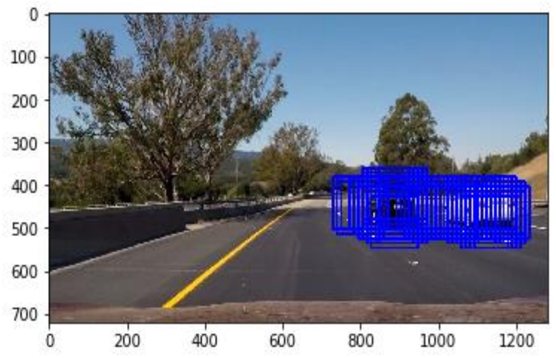
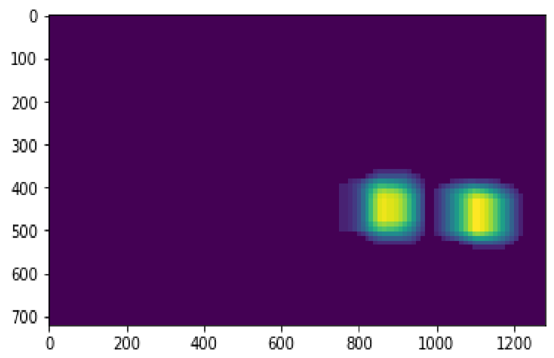
Please find the attached p5_3.mp4 to my project submission folder.

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.

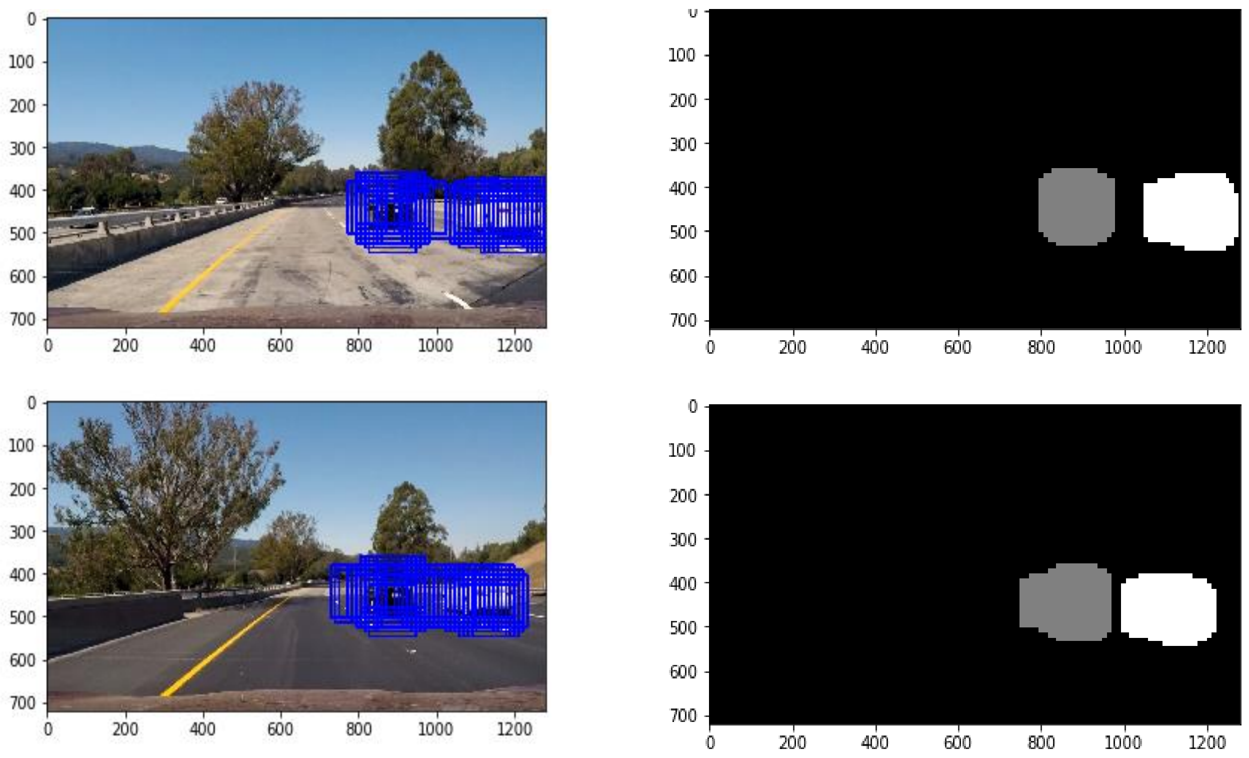
I recorded the positions of positive detections in each frame of the video. From the positive detections I created a heatmap and then thresholded that map to identify vehicle positions. I then used scipy.ndimage.measurements.label() to identify individual blobs in the heatmap. I then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected.

Here's an example result showing the heatmap from a series of frames of video, the result of scipy.ndimage.measurements.label() and the bounding boxes then overlaid on the last frame of video:

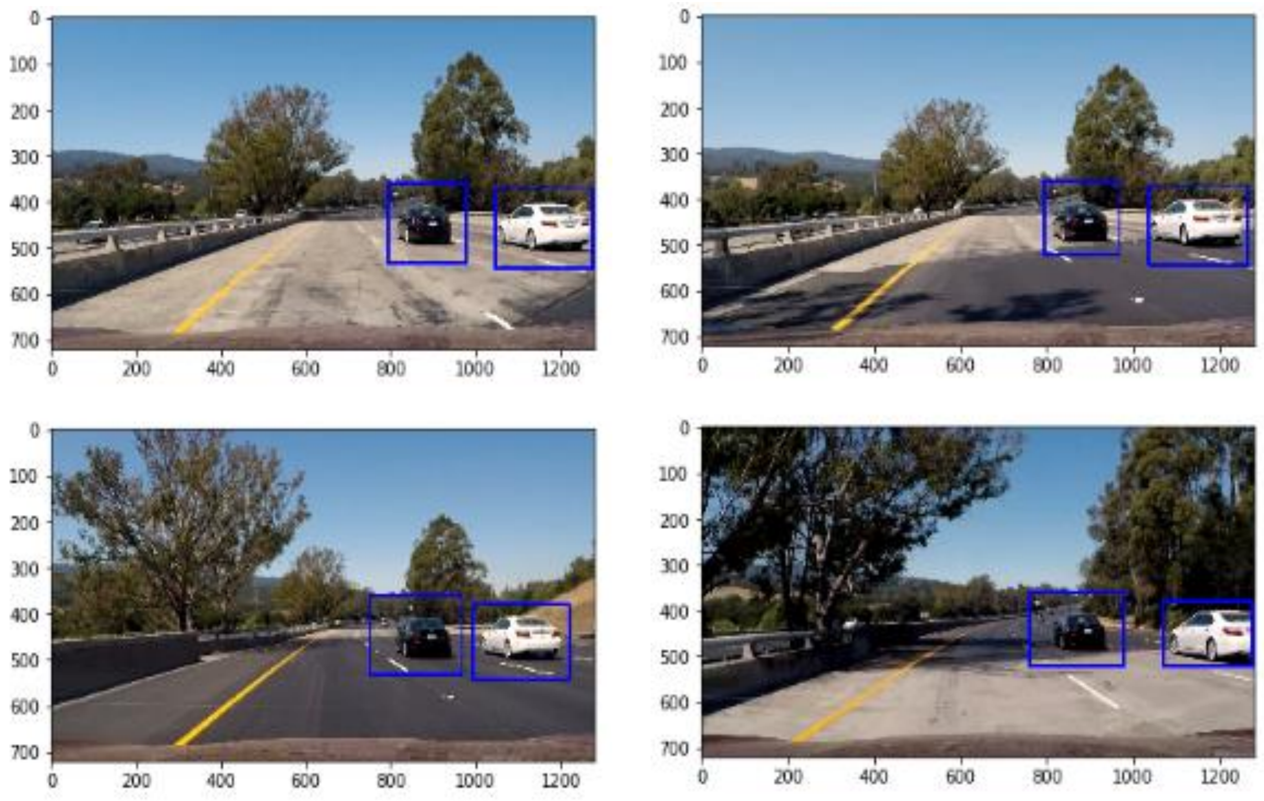
Here are two frames and their corresponding heatmaps:

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



Here the resulting bounding boxes are drawn onto the last frame in the series:



Smoothing:

Averaging of frames performed over 10 frames to smooth out the boxes and remove the false positives that might occur in a single frame. The boxes over 10 frames were stored and a heatmap was applied to those 10 frames to get an averaged heatmap.

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?

Detecting and tracking moving vehicles is a tough task. There are a lot of false positives which are a pain to eliminate. The method will fail in a scenario where it would encounter new types of vehicles with different sizes and shapes upon which it hasn't been trained. Sometimes the classifier gets confused between a white car and white lanes. The processing time is very high and it takes 15-20 minutes to process the project video.

It can be made more robust by training it on newer scenarios. Techniques which are better than Linear SVM can be tried out which I didn't do. More color spaces can be explored.