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.

Writeup / README

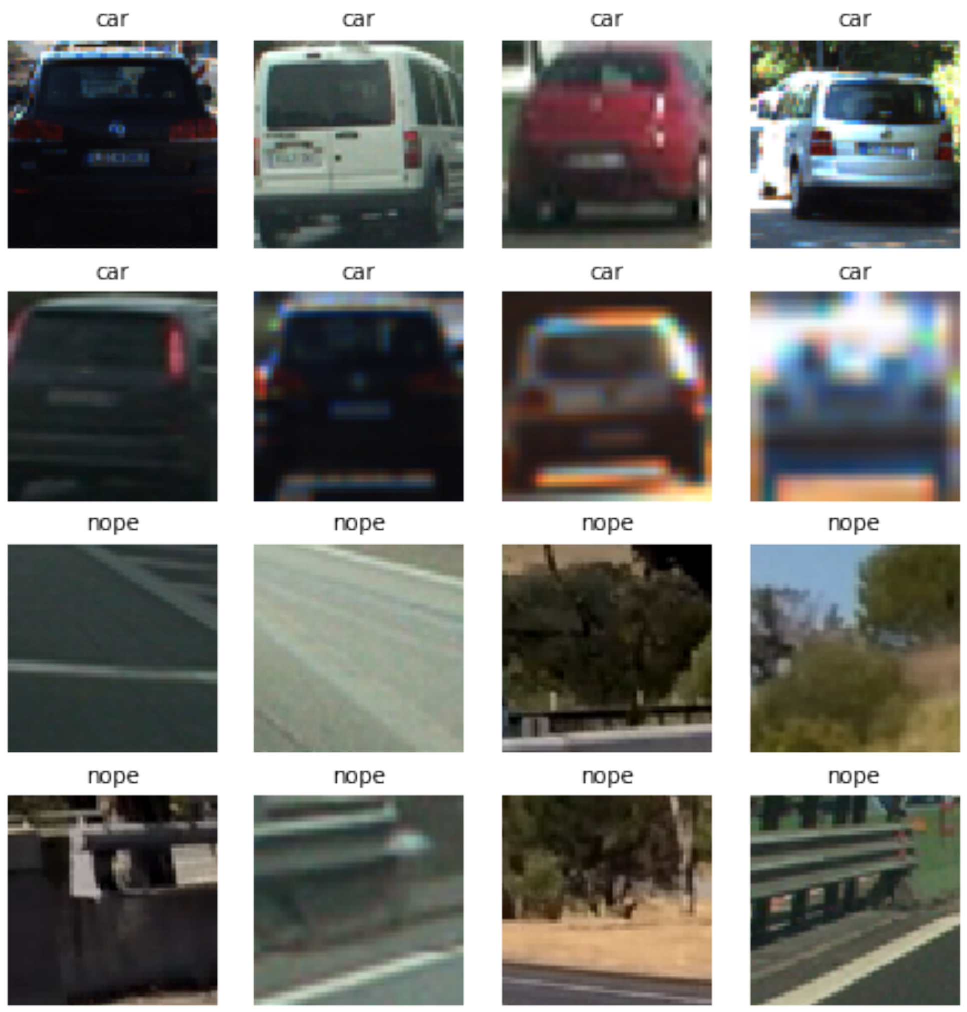
1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

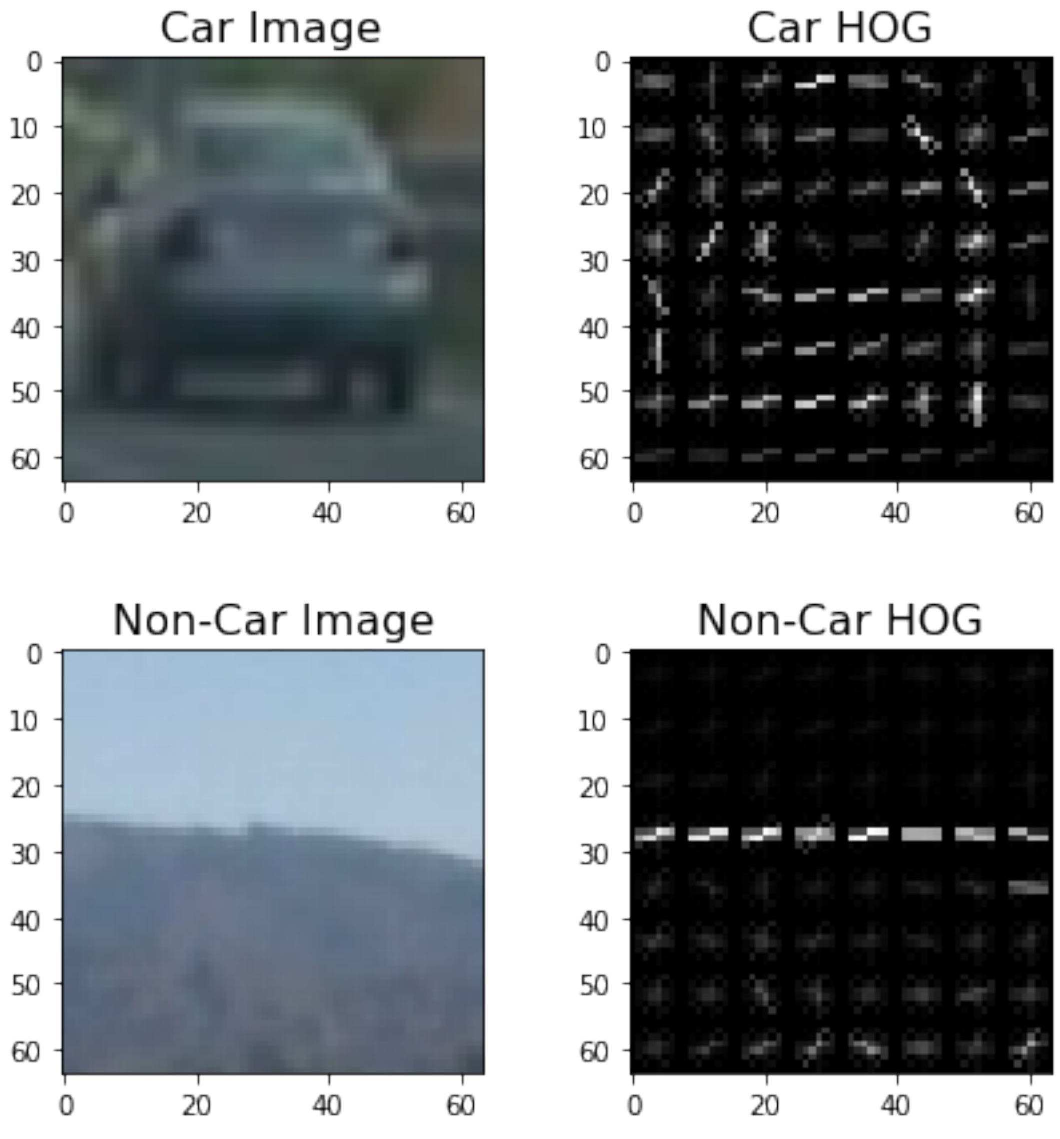
Histogram of Oriented Gradients (HOG)

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

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:



The code for extracting HOG features from an image is defined by the method `get_hog_features` and is contained in the cell titled "Convert Image to Histogram of Oriented Gradients (HOG)." The figure below shows a comparison of a car image and its associated histogram of oriented gradients, as well as the same for a non-car image.



The method `extract_features` in the section titled "Method to Extract HOG Features" accepts a list of image paths and HOG parameters (as well as one of a variety of destination color spaces, to which the input image is converted), and produces a flattened array of HOG features for each image in the list.

Next, in the section titled "Extract Features," I define parameters for HOG feature extraction and extract features for the entire dataset. In the next section titled “Combine Dataset”,these feature sets are combined and a label vector is defined (`1` for cars, `0` for non-cars). In the “Shuffle and Split”,the features and labels are then shuffled and split into training and test sets in preparation to be fed to a linear support vector machine (SVM) classifier.The table below documents the twenty-five different parameter combinations that I explored.

Configuration Label	Colorspace	Orientations	Pixels Per Cell	Cells Per Block	HOG Channel
1	RGB	9	8	2	ALL
2	HSV	9	8	2	1
3	HSV	9	8	2	2
4	LUV	9	8	2	0
5	LUV	9	8	2	1
6	HLS	9	8	2	0
7	HLS	9	8	2	1
8	YUV	9	8	2	0
9	YCrCb	9	8	2	1
10	YCrCb	9	8	2	2
11	HSV	9	8	2	ALL
12	LUV	9	8	2	ALL
13	HLS	9	8	2	ALL
14	YUV	9	8	2	ALL
15	YCrCb	9	8	2	ALL
16	YUV	9	8	1	0
17	YUV	9	8	3	0
18	YUV	6	8	2	0

19	YUV	12	8	2	0
20	YUV	11	8	2	0
21	YUV	11	16	2	0
22	YUV	11	12	2	0
23	YUV	11	4	2	0
24	YUV	11	16	2	ALL
25	YUV	7	16	2	ALL

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

I settled on my final choice of HOG parameters based upon the performance of the SVM classifier produced using them. I considered not only the accuracy with which the classifier made predictions on the test dataset, but also the speed at which the classifier is able to make predictions. There is a balance to be struck between accuracy and speed of the classifier, and my strategy was to bias toward speed first, and achieve as close to real-time predictions as possible, and then pursue accuracy if the detection pipeline were not to perform satisfactorily.

The final parameters chosen were those labeled "configuration 24" in the table above: YUV colorspace, 11 orientations, 16 pixels per cell, 2 cells per block, and `ALL` channels of the colorspace. The classifier performance of each of the configurations from the table above are summarized in the table below:

Configuration (above)	Classifier	Accuracy	Train Time
1	Linear SVC	97.52	19.21
2	Linear SVC	91.92	5.53
3	Linear SVC	96.09	4.29
4	Linear SVC	95.72	4.33
5	Linear SVC	94.51	4.51
6	Linear SVC	92.34	4.97
7	Linear SVC	95.81	4.04
8	Linear SVC	96.28	5.04
9	Linear SVC	94.88	4.69
10	Linear SVC	93.78	4.59
11	Linear SVC	98.31	16.03
12	Linear SVC	97.52	14.77
13	Linear SVC	98.42	13.46
14	Linear SVC	98.40	15.68
15	Linear SVC	98.06	12.86
16	Linear SVC	94.76	5.11
17	Linear SVC	96.11	6.71
18	Linear SVC	95.81	3.79
19	Linear SVC	95.95	4.84
20	Linear SVC	96.59	5.46
21	Linear SVC	95.16	0.52
22	Linear SVC	94.85	1.27

23	Linear SVC	95.92	21.39
24	Linear SVC	97.69	3.01
25	Linear SVC	97.61	1.42

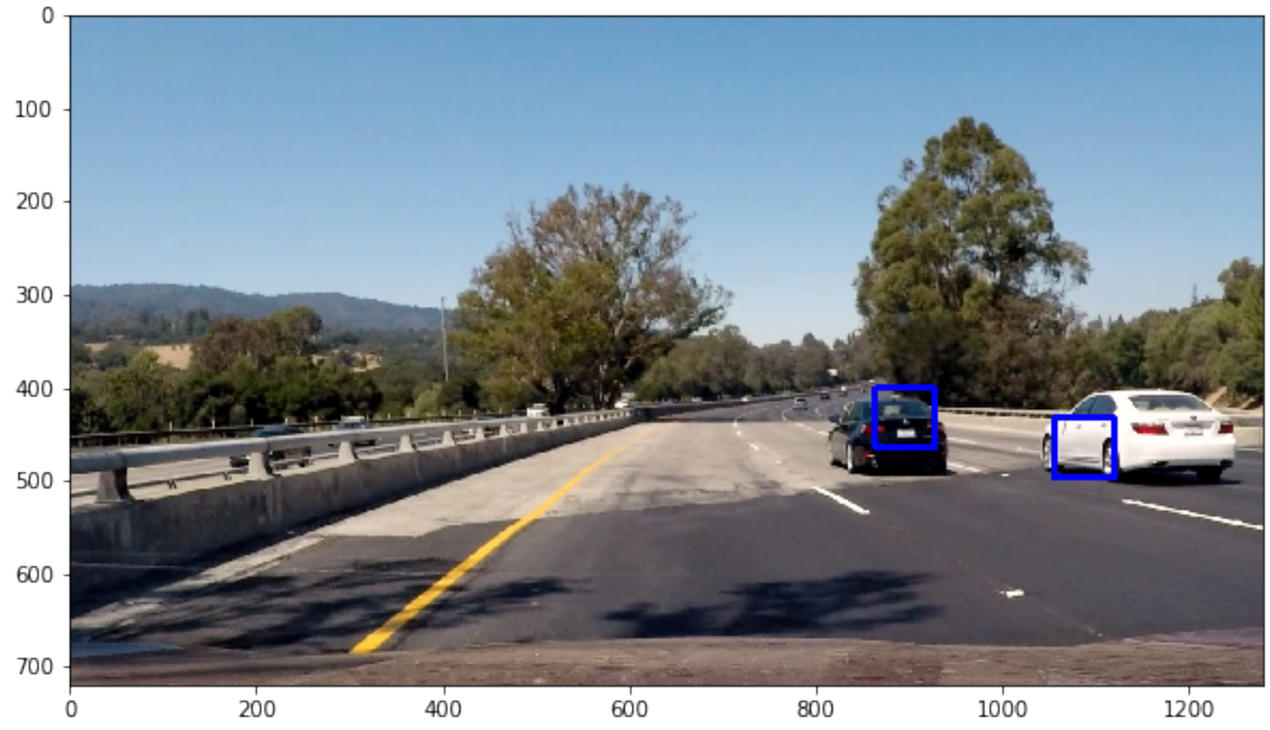
3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

In the section titled "Train a Classifier" I trained a linear SVM with the default classifier parameters and using HOG features alone (I did not use spatial intensity or channel intensity histogram features) and was able to achieve a test accuracy of 97.69%.

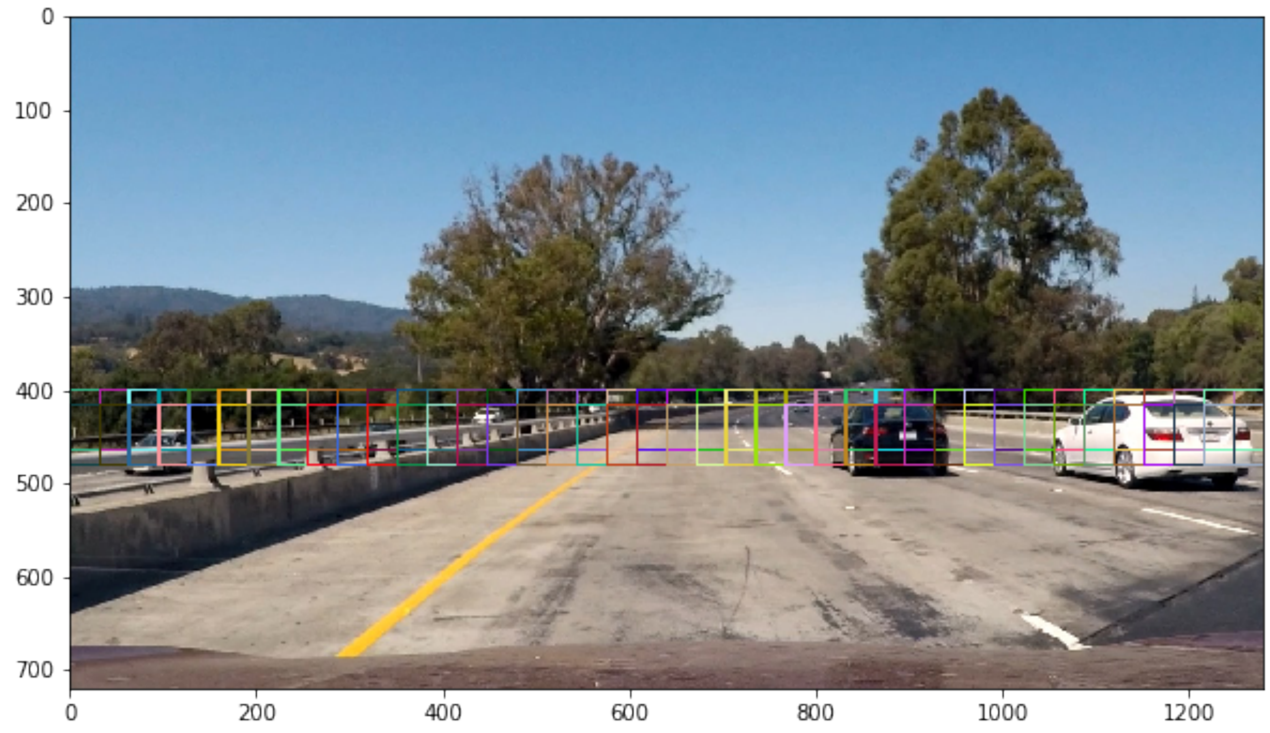
Sliding Window Search

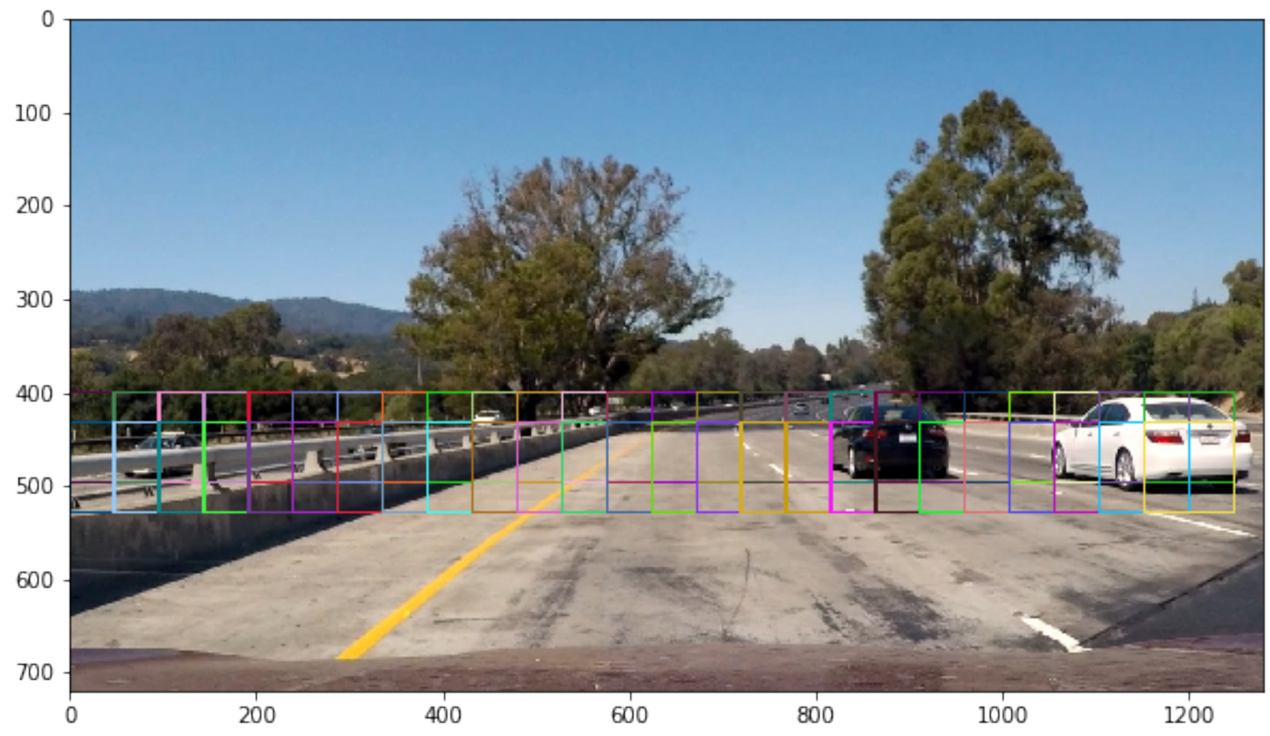
1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

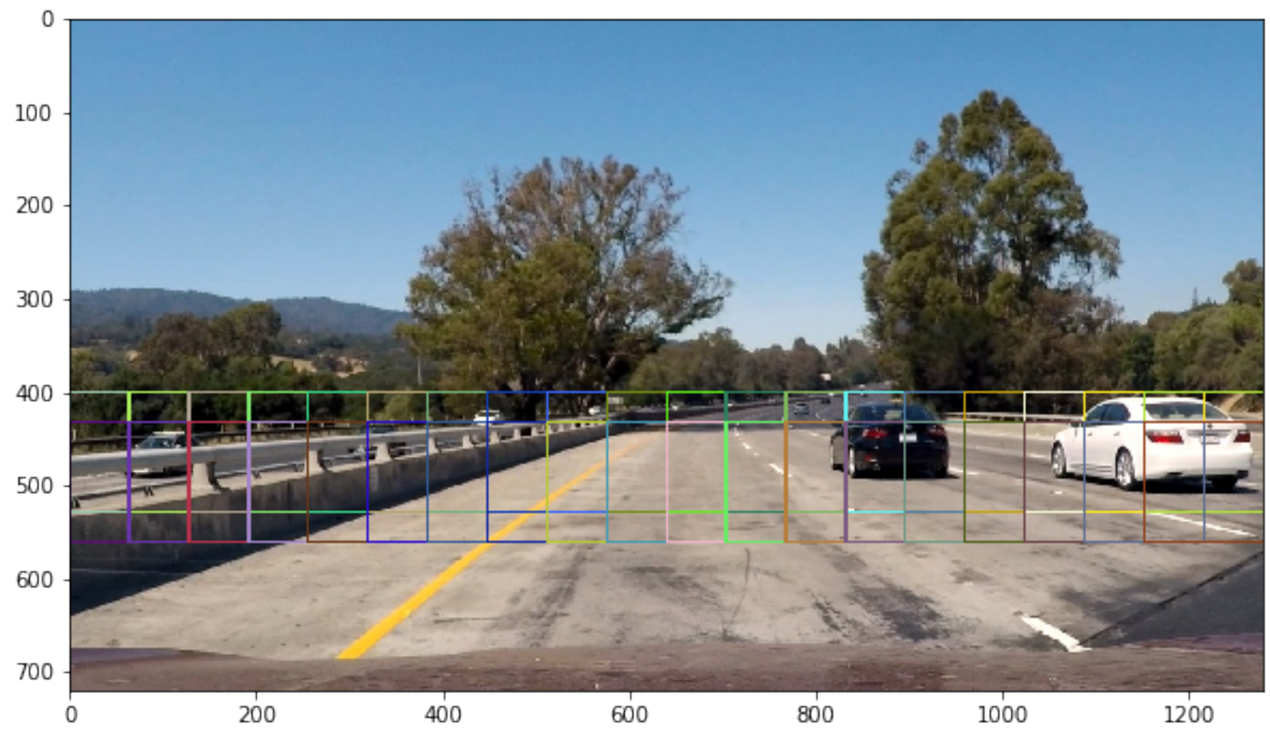
In the section titled "Method for Using Classifier to Detect Cars in an Image" I adapted the method `find_cars` from the lesson materials. The method combines HOG feature extraction with a sliding window search, but rather than perform feature extraction on each window individually which can be time consuming, the HOG features are extracted for the entire image (or a selected portion of it) and then these full-image features are subsampled according to the size of the window and then fed to the classifier. The method performs the classifier prediction on the HOG features for each window region and returns a list of rectangle objects corresponding to the windows that generated a positive prediction. The image below shows the first attempt at using `find_cars` on one of the test images, using a single window size:



I explored several configurations of window sizes and positions, with various overlaps in the X and Y directions. The following four images show the configurations of all search windows in the final implementation, for small (1x), medium (1.5x, 2x), and large (3x) windows:

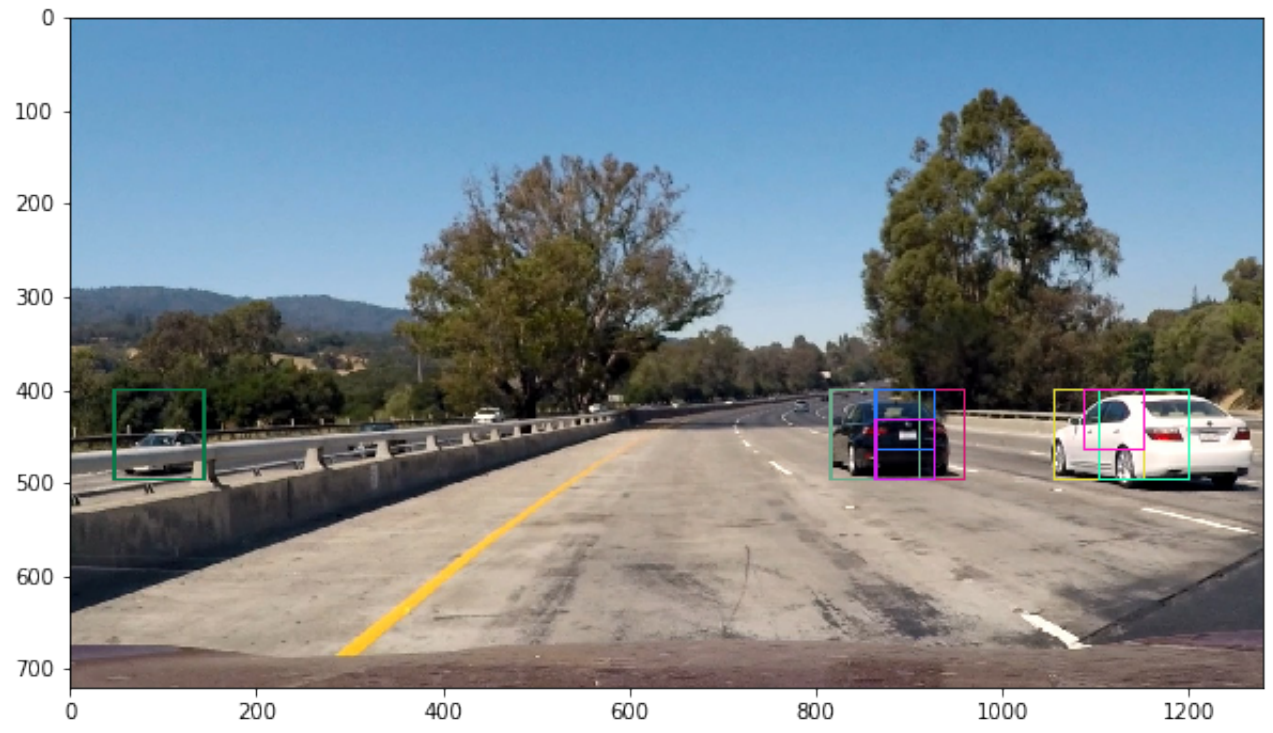




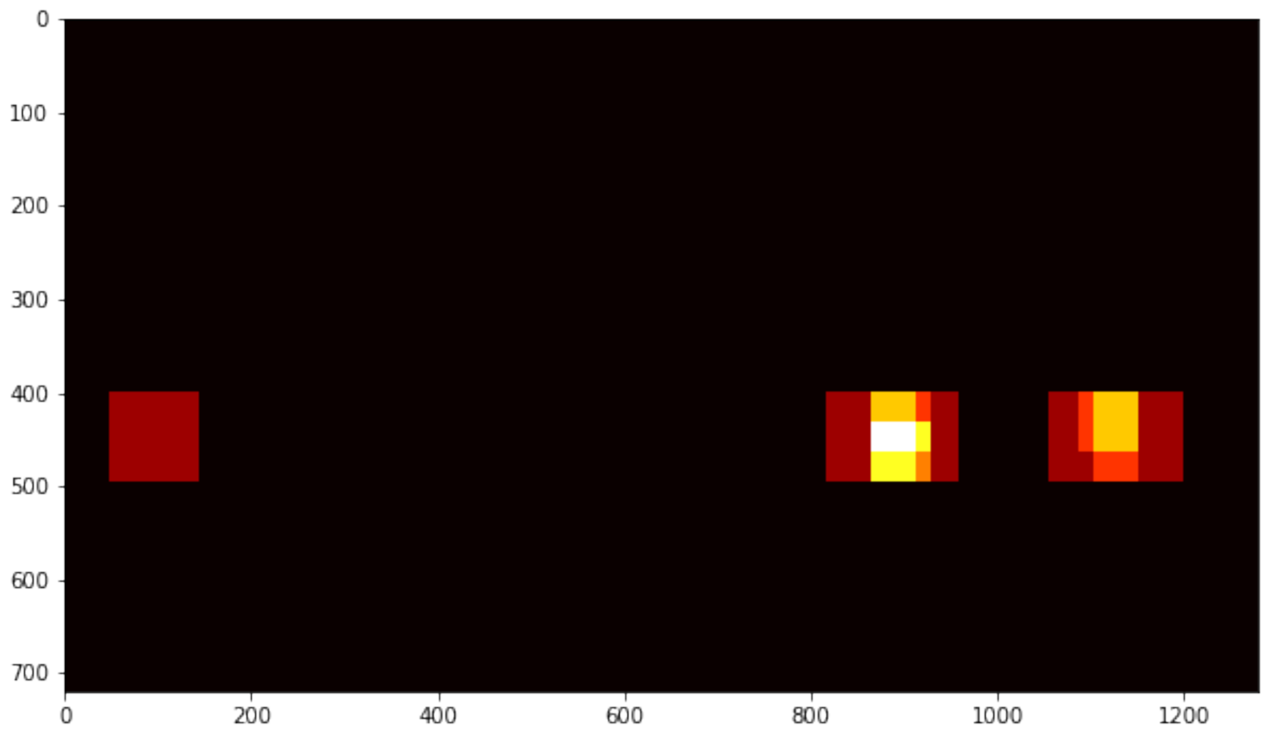


The final algorithm calls `find_cars` for each window scale and the rectangles returned from each method call are aggregated. In previous implementations smaller (0.5) scales were explored but found to return too many false positives, and originally the window overlap was set to 50% in both X and Y directions, but an overlap

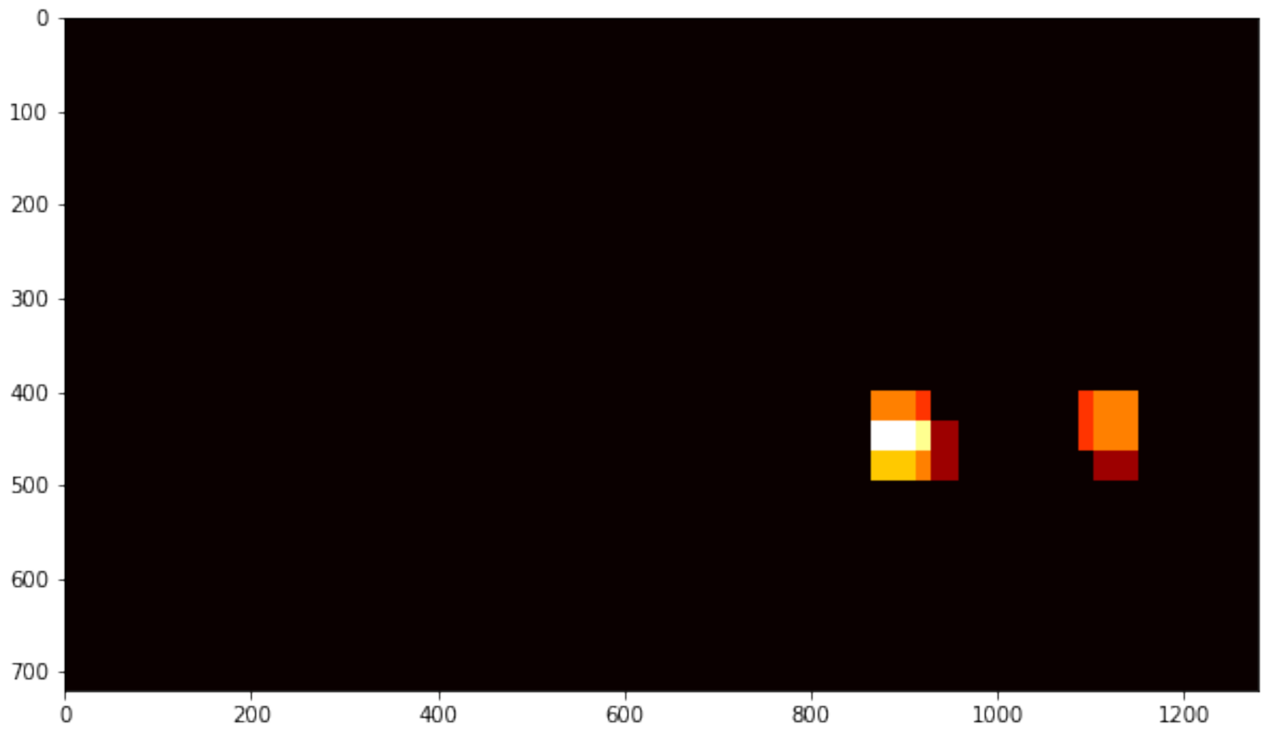
of 75% in the Y direction (yet still 50% in the X direction) produced more redundant true positive detections, which were preferable given the heatmap strategy described below. Additionally, only an appropriate vertical range of the image is considered for each window size (e.g. smaller range for smaller scales) to reduce the chance for false positives in areas where cars at that scale are unlikely to appear. The final implementation considers 190 window locations, which proved to be robust enough to reliably detect vehicles while maintaining a high speed of execution.
The image below shows the rectangles returned by `find_cars` drawn onto one of the test images in the final implementation. Notice that there are several positive predictions on each of the near-field cars, and one positive prediction on a car in the oncoming lane.



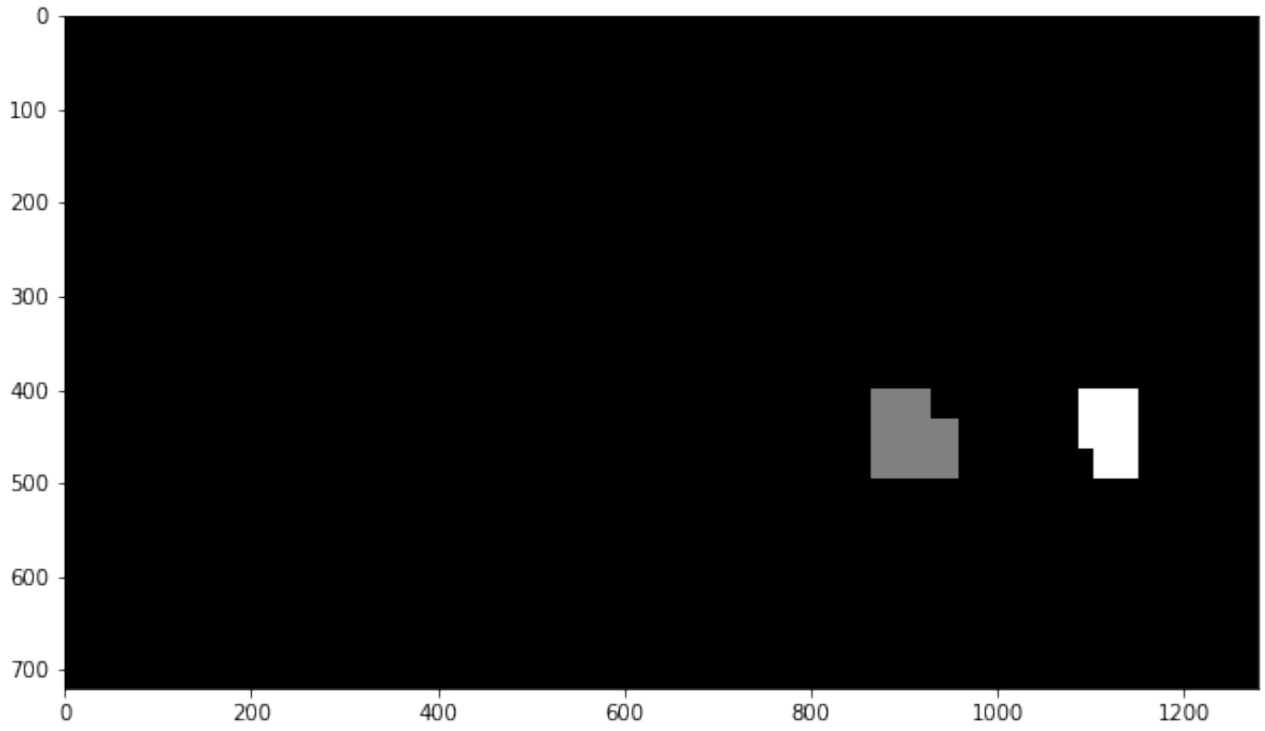
Because a true positive is typically accompanied by several positive detections, while false positives are typically accompanied by only one or two detections, a combined heatmap and threshold is used to differentiate the two. The `add_heat` function increments the pixel value (referred to as "heat") of an all-black image the size of the original image at the location of each detection rectangle. Areas encompassed by more overlapping rectangles are assigned higher levels of heat. The following image is the resulting heatmap from the detections in the image above:



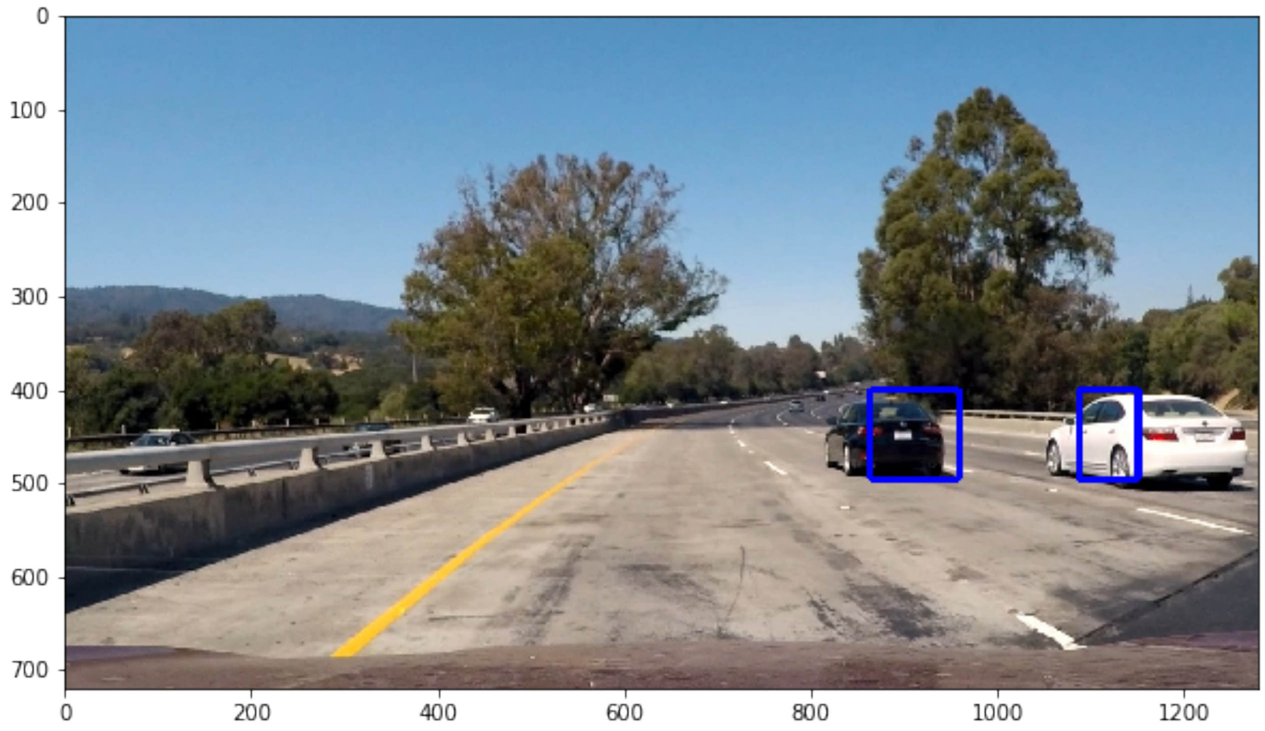
A threshold is applied to the heatmap (in this example, with a value of 1), setting all pixels that don't exceed the threshold to zero. The result is below:



The `scipy.ndimage.measurements.label()` function collects spatially contiguous areas of the heatmap and assigns each a label:

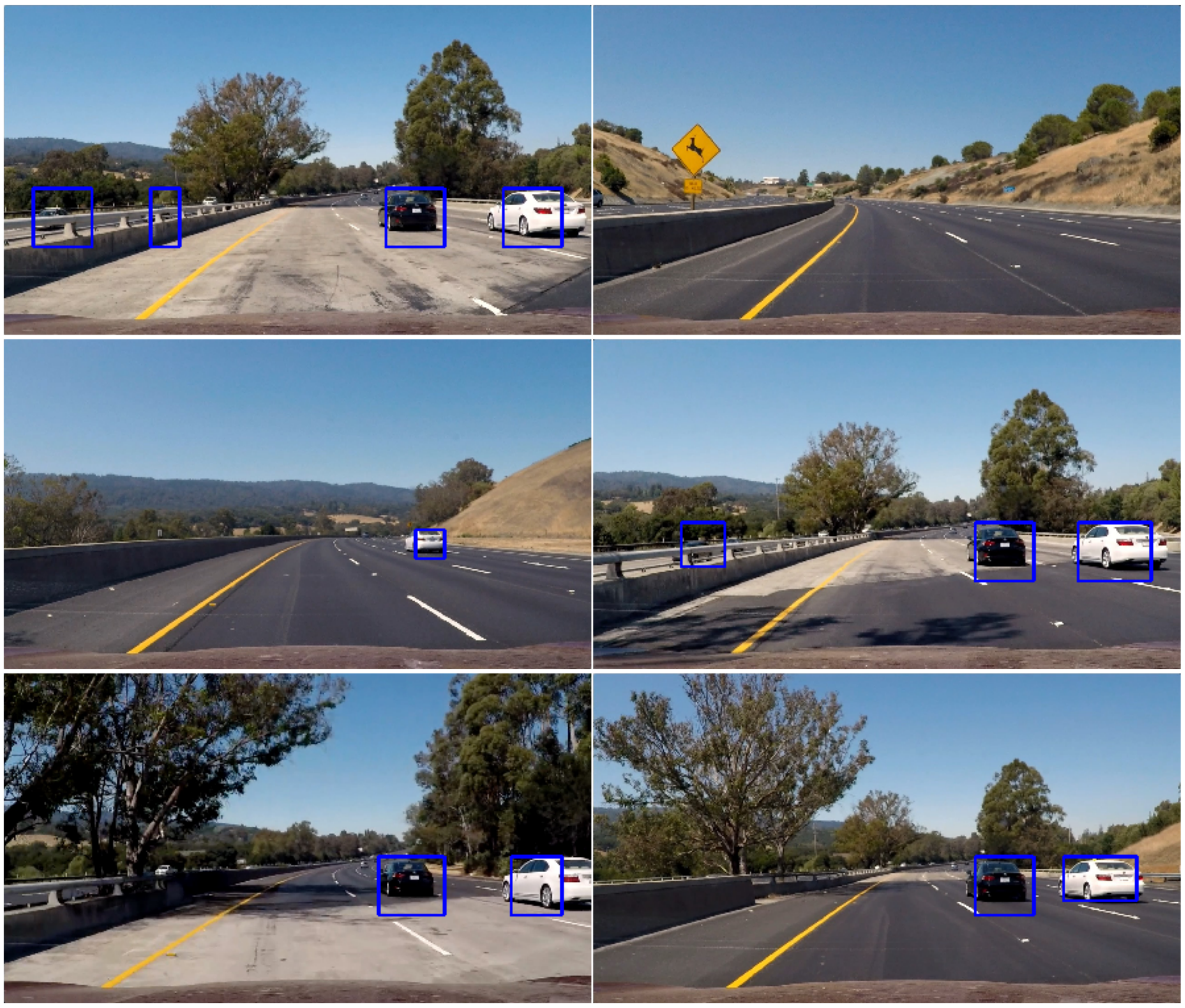


And the final detection area is set to the extremities of each identified label:



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?

The results of passing all of the project test images through the above pipeline are displayed in the images below:



The final implementation performs very well, identifying the near-field vehicles in each of the images with no false positives.
The first implementation did not perform as well, so I began by optimizing the SVM classifier. The original classifier used HOG features from the YUV Y channel only, and achieved a test accuracy of 96.28%. Using all three YUV channels increased the accuracy to 97.69%, but also tripled the execution time. However, changing the `pixels_per_cell` parameter from 8 to 16 produced a

roughly ten-fold increase in execution speed with minimal cost to accuracy.

Other optimization techniques included changes to window sizing and overlap as described above, and lowering the heatmap threshold to improve accuracy of the detection (higher threshold values tended to underestimate the size of the vehicle).

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.)

The Video can be found at the same folder as the Writeup File.

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.

The code for processing frames of video is contained in the cell titled "Pipeline for Processing Video Frames" and is identical to the code for processing a single image described above, with the exception of storing the detections from the previous 15 frames of video using the `prev_rects` parameter from a class called `Vehicle_Detect`. Rather than performing the heatmap/threshold/label steps for the current frame's detections, the detections for the past 15 frames are combined and added to the heatmap and the threshold for the heatmap is set to `1 + len(det.prev_rects)//2` (one more than half the number of rectangle sets contained in the history) - this value was found to perform best empirically (rather than using a single scalar, or the full number of rectangle sets in the history).

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 problems that I faced while implementing this project were mainly concerned with detection accuracy. Balancing the accuracy of the classifier with execution speed was crucial. Scanning 190 windows using a classifier that achieves approx 98% accuracy should result in around 4 misidentified windows per frame. Of course, integrating detections from previous frames mitigates the effect of the misclassifications, but it also introduces another problem: vehicles that significantly change position from one frame to the next (e.g. oncoming traffic) will tend to escape being labeled.

The pipeline is probably most likely to fail in cases where vehicles don't resemble those in the training dataset, but lighting and environmental conditions might also play a role As stated above, oncoming cars are an issue, as well as distant cars.

I believe that the best approach, given plenty of time to pursue it, would be to combine a very high accuracy classifier with high overlap in the search windows. The execution cost could be offset with more intelligent tracking strategies, such as:

- determine vehicle location and speed to predict its location in subsequent frames
- begin with expected vehicle locations and nearest (largest scale) search areas, and preclude overlap and redundant detections from smaller scale search areas to speed up execution.