Vehicle Detection Project

###Histogram of Oriented Gradients (HOG)

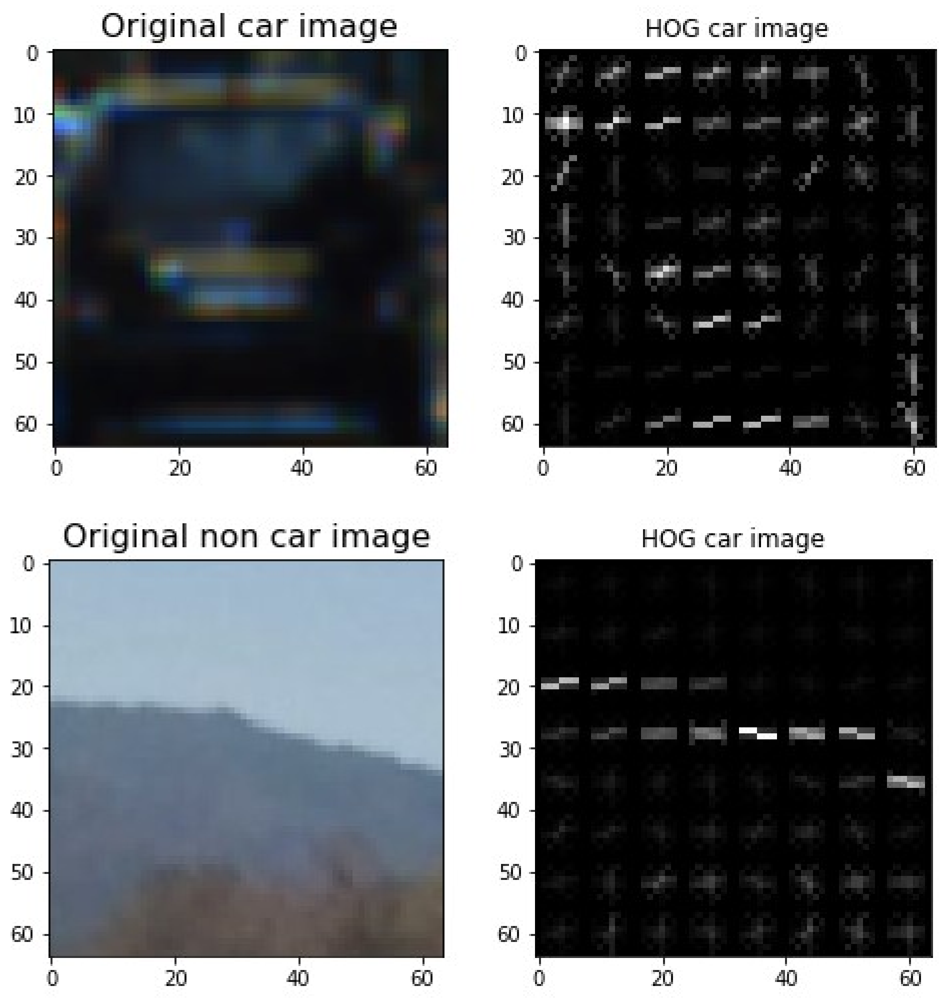
####1. Explain how (and identify where in your code) you extracted HOG features from the training images.

The code for this step is contained in the first code cell of the IPython notebook.

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:

In order to extract HOG features from an image I used the method ‘ get_hog_features ’, positioned at the 7th cell in the notebook.

The picture below shows a comparison between car image as well as a non-car image and the relative associated histogram of oriented gradients.



The method 'extract_features' (8th cell) accepts a list of images paths and HOG parameters and produces a flattened array of HOG features for each image in the list.

After that, with the features of both type of images, I produced a label vector for the entire dataset skimming car from not-cars; 1 was used for cars and 0 for not-cars.

Subsequently, the features and the labels are then shuffled and split into training and test sets, so that in turn they ended up to feed a SVM classifier.

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

I tried various combinations of parameters, accurately tuned, and I made my final choice upon the performance of the SVM classifier produced using the best ones among them. In the end it was the right compromise between accuracy and speed that made me choose for the YUV color space, 11 orientations, 16 pixels per cell, 2 cells per block and all channel for the color space.

####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).

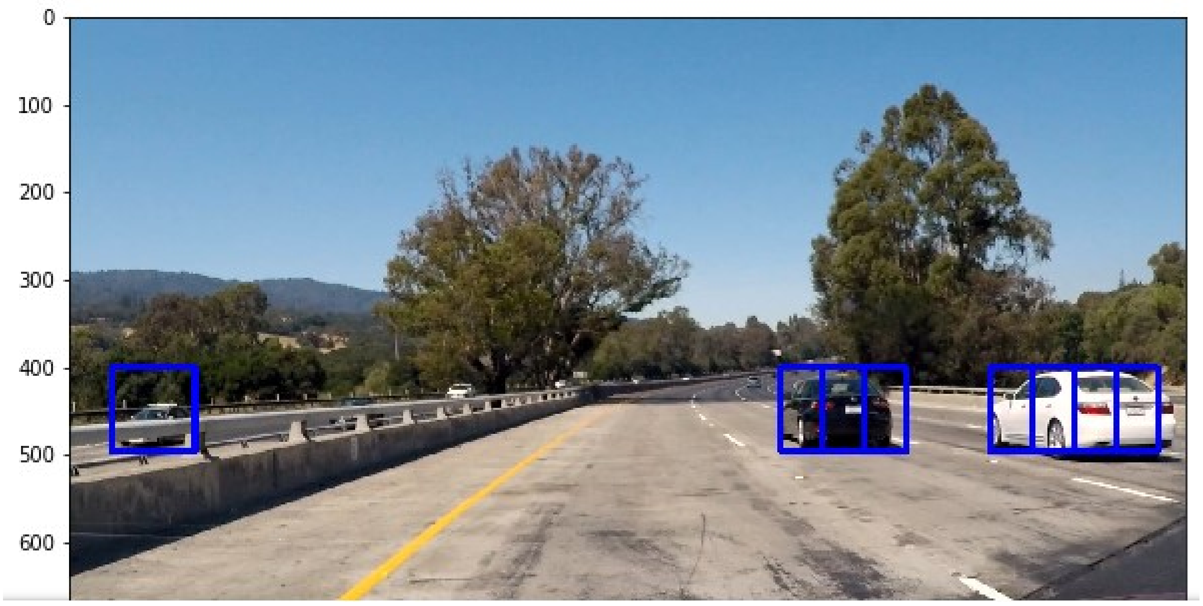
I trained a linear SVM using the default classifier parameters and just using HOG features alone, even though I was able to achieve a test accuracy of 98,2 %.

###Sliding Window Search

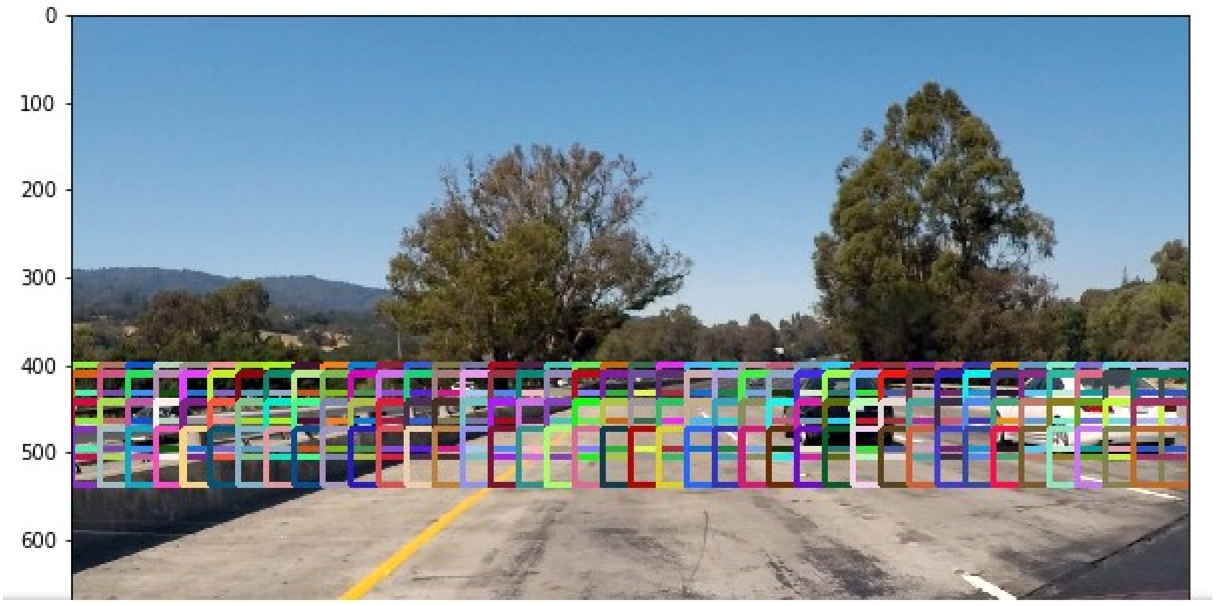
####1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

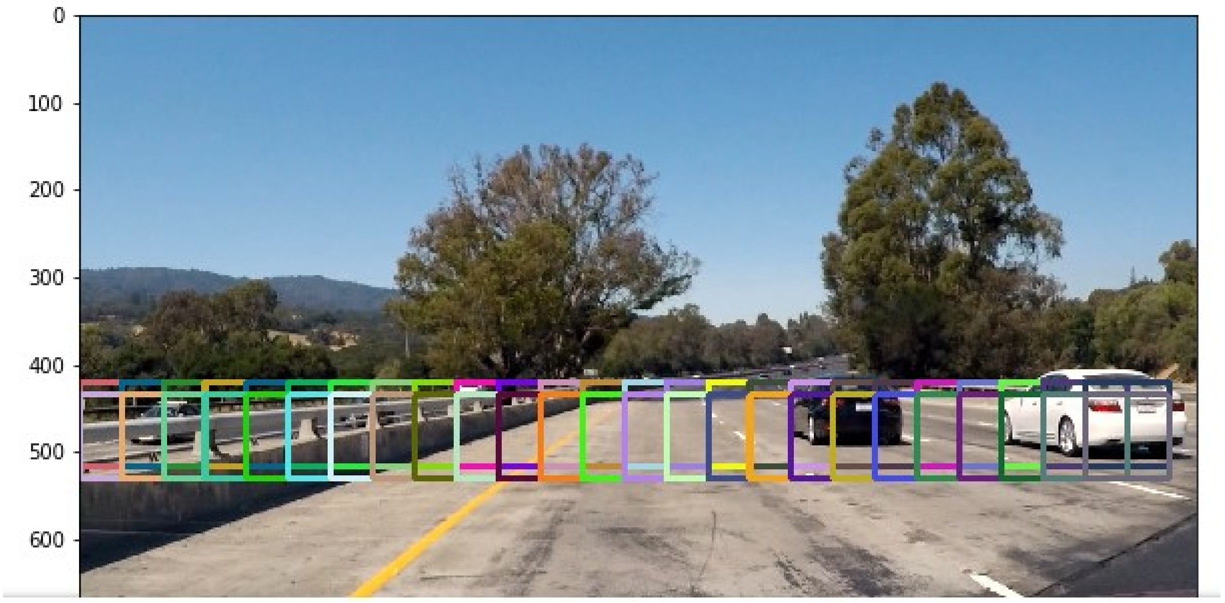
In the section " Method to find cars trough a classifier " I used 'find cars' function which combines HOG feature extraction with a sliding window search. The particularity of the approach consists to retrieve HOG features on a portion of the image subsampling the same features according to the size of the window and then pass them to the classifier. The method performs the classifier prediction on each window region and returns a list of rectangle objects corresponding to the windows that targeted a car prediction.

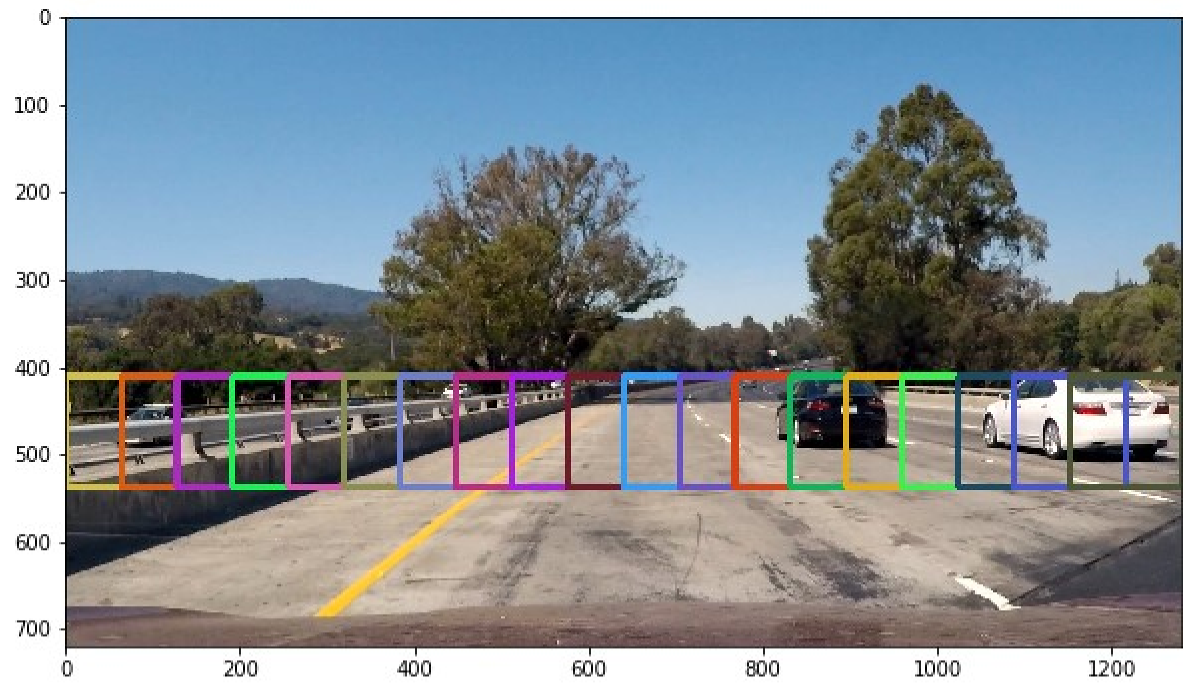
The image below shows the result.



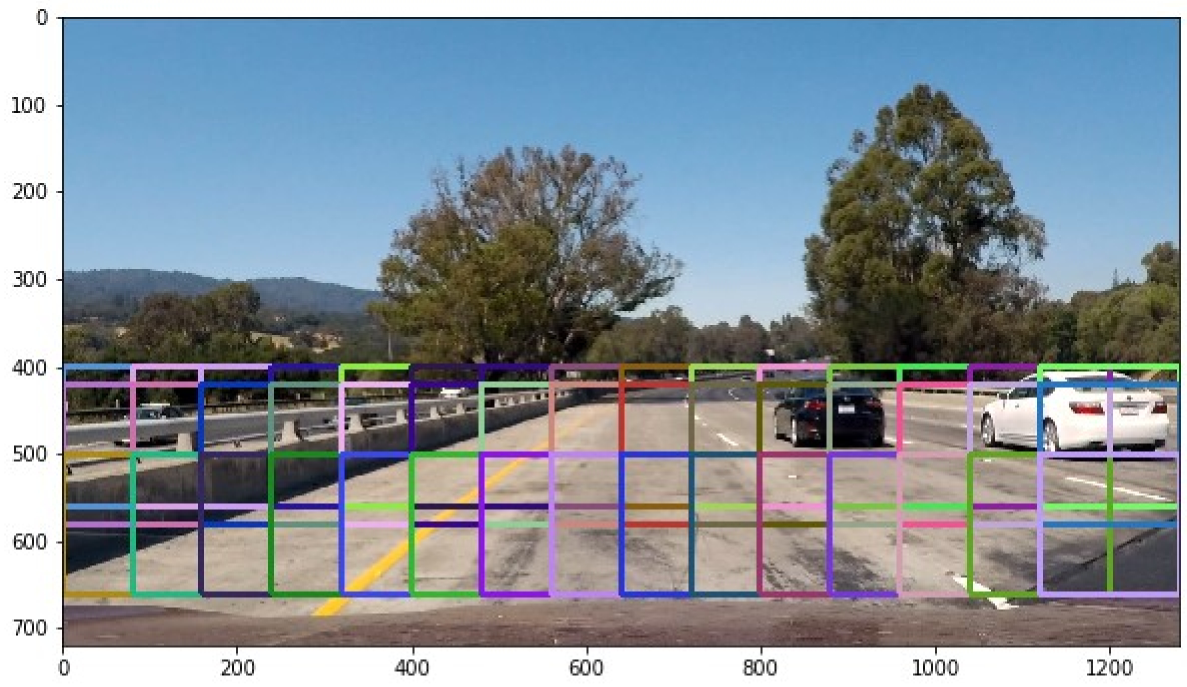
Therefore, I tried to explore several combinations of windows sizes and positions, among a precise sections of the image. Below few examples of images with all the spatial possibilities.


0
100
200
300
400
500
600


0
100
200
300
400
500
600

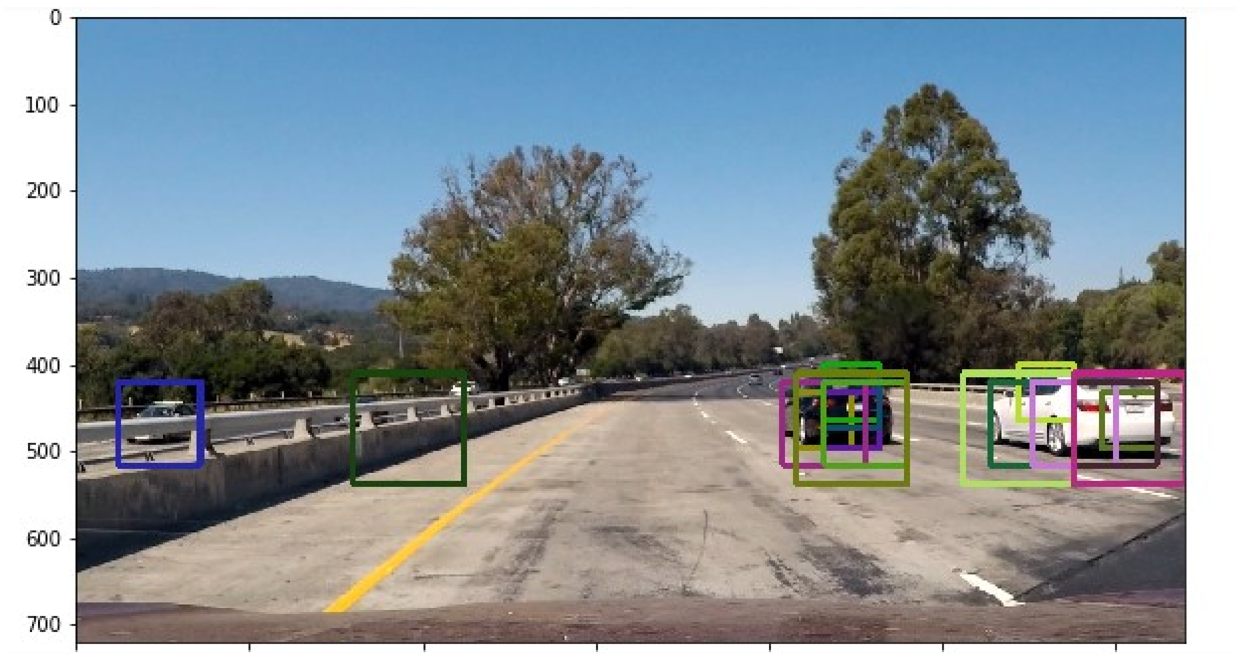


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



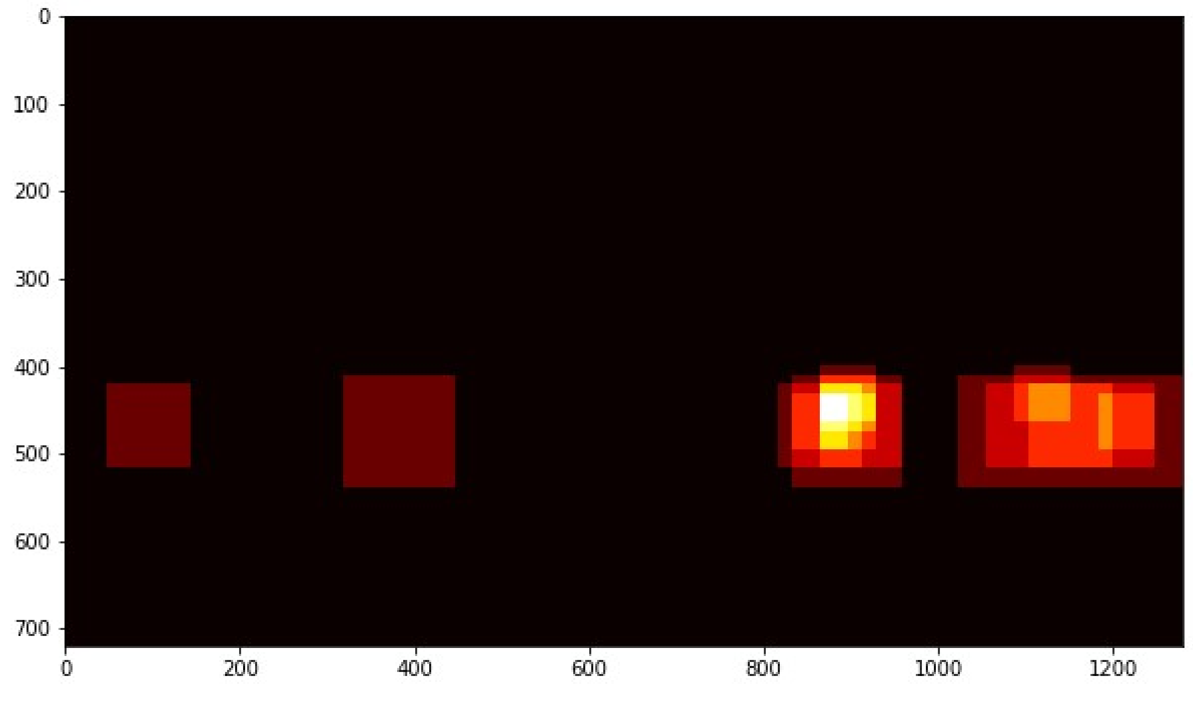
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

The image below shows just the rectangles that should identify cars.

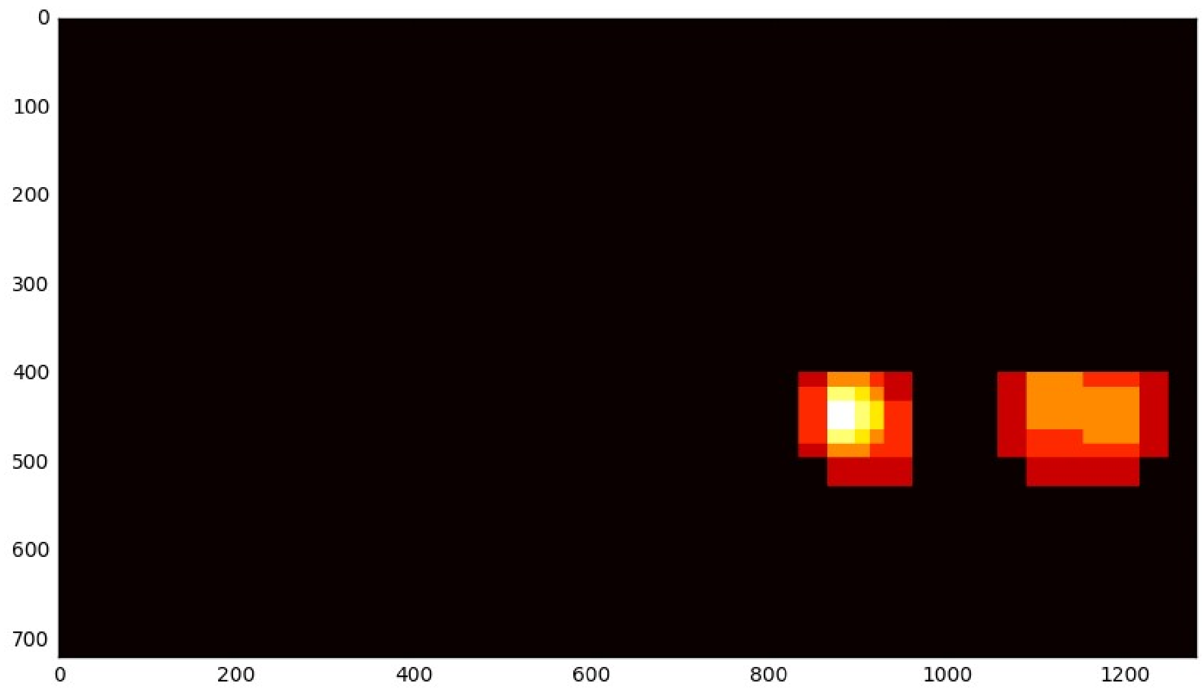


Regarding this last picture we can notice the presence of a couple of false positives. In order to avoid this phenomenon is recommended to implement a further step.

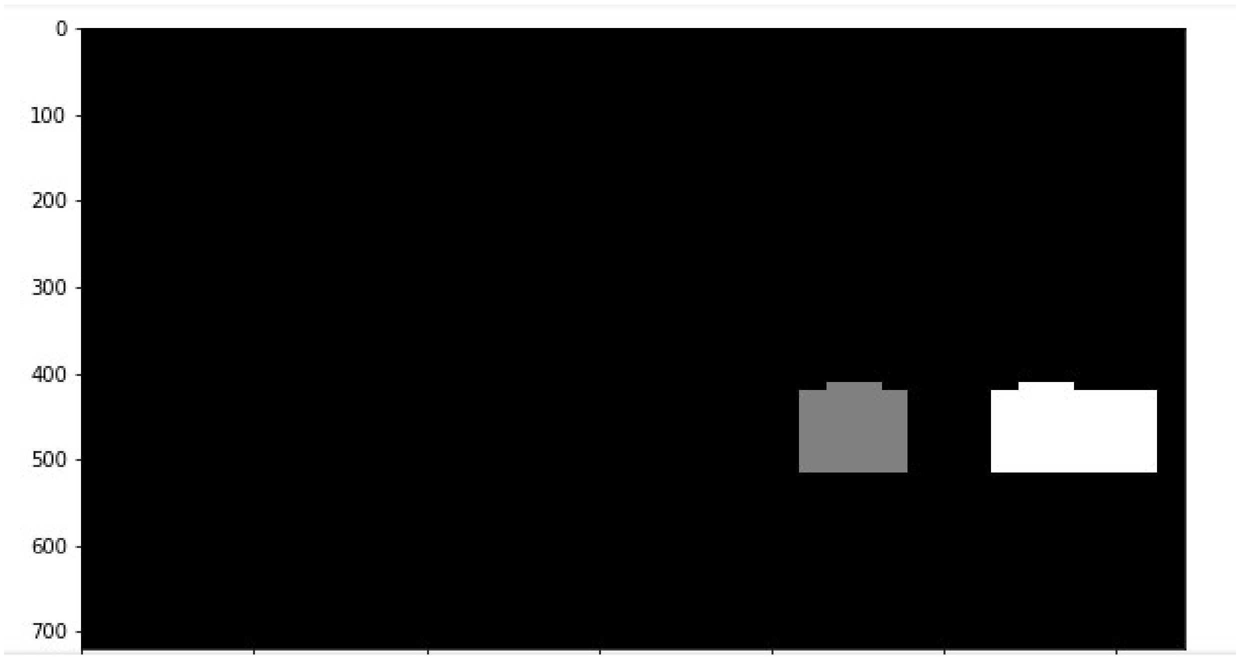
The synergy of a heatmap and its relative threshold can eliminate the unnecessary elements from the image. About the heatmap, I used the 'add_heat' function in order to stand out the multiple rectangles which identify with all likelihood cars. The image below is an example of this concept. You can notice more bright colours on the overlapped regions of interest.



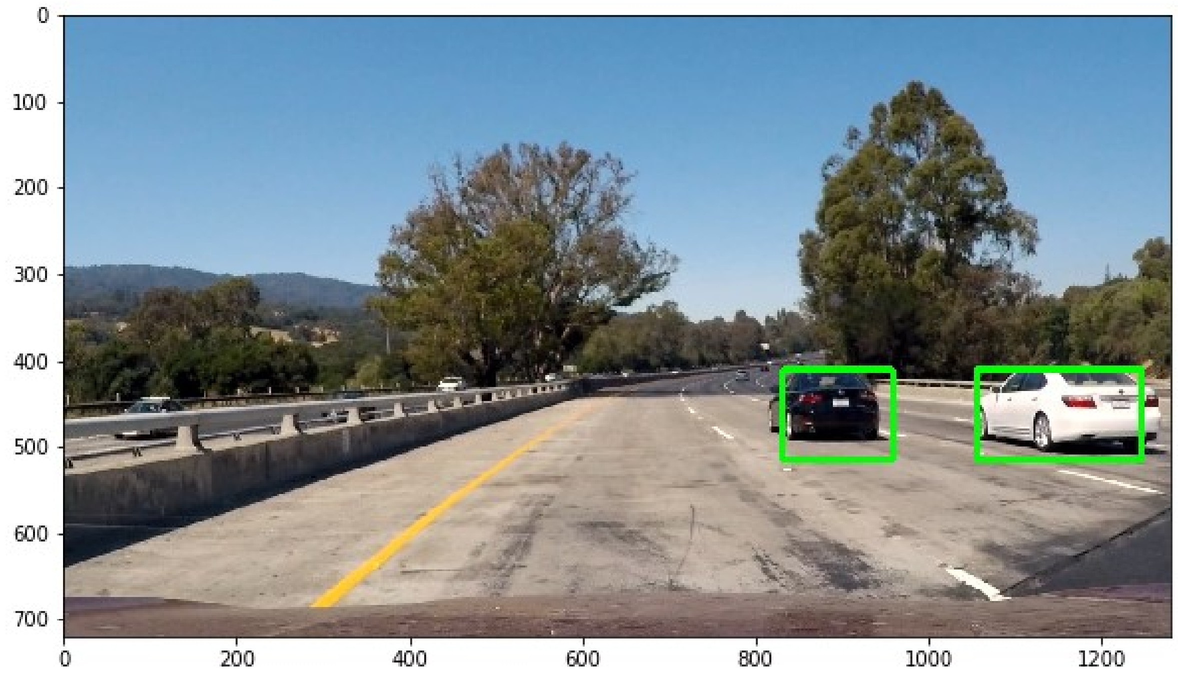
Follow the same image above after applying a threshold of 1.



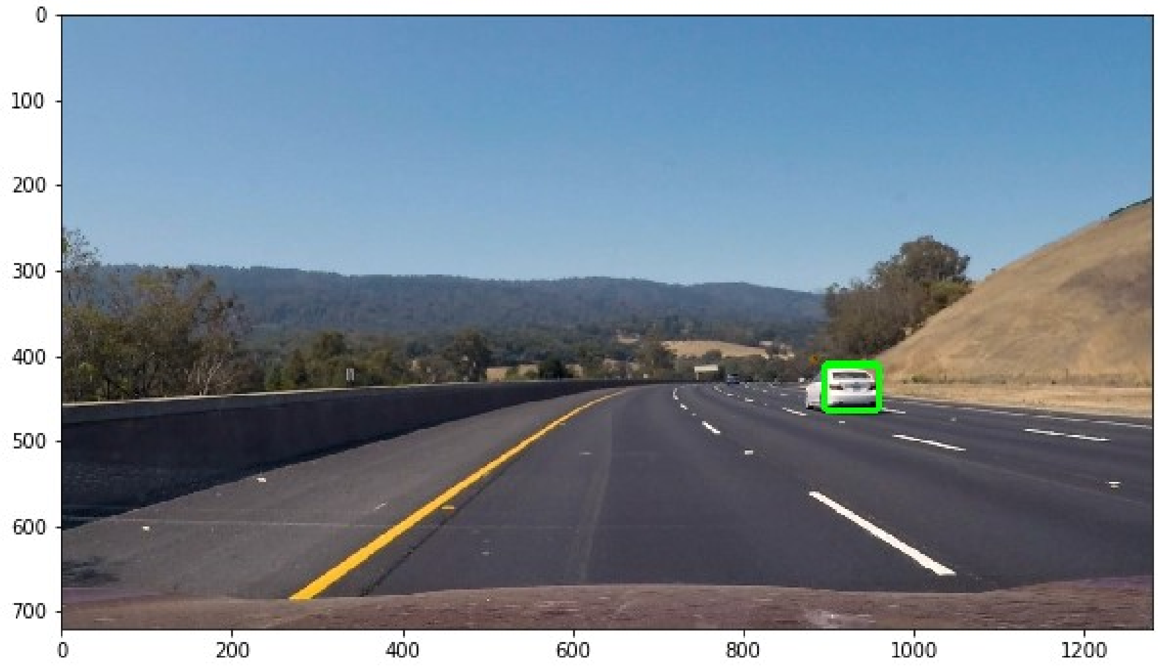
With the ‘scipy.ndimage.measurements.label()’ is possible to collect spatially contiguous areas of the heatmap and assign each label.



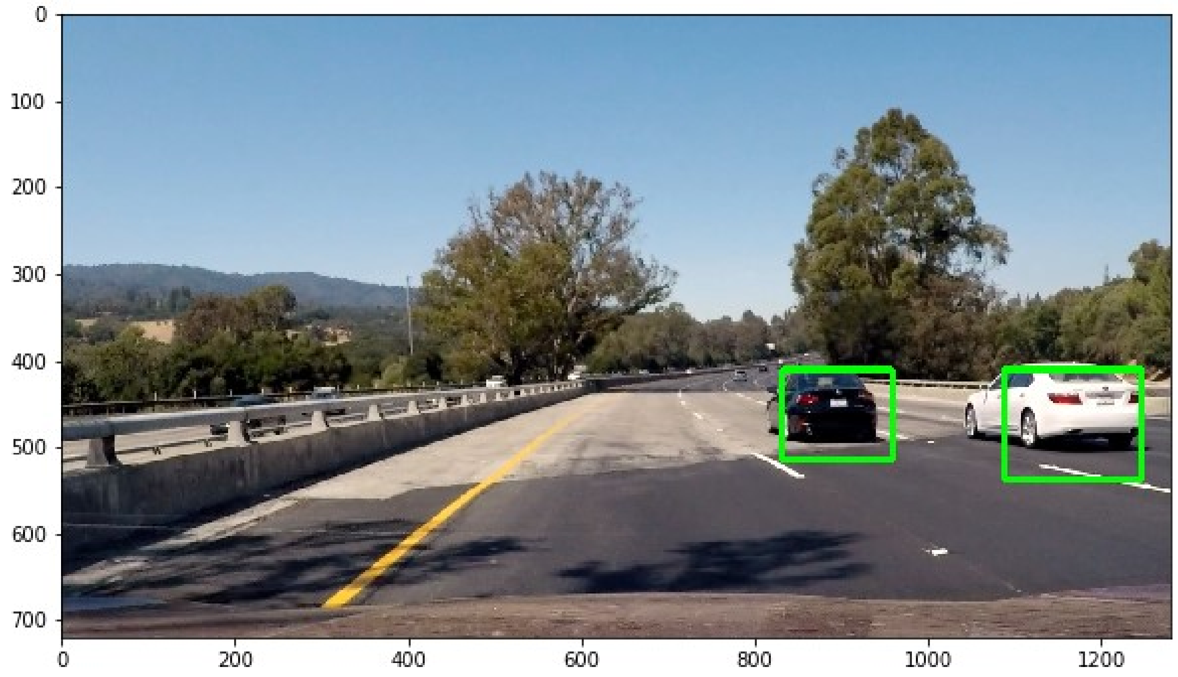
The final result is the following:



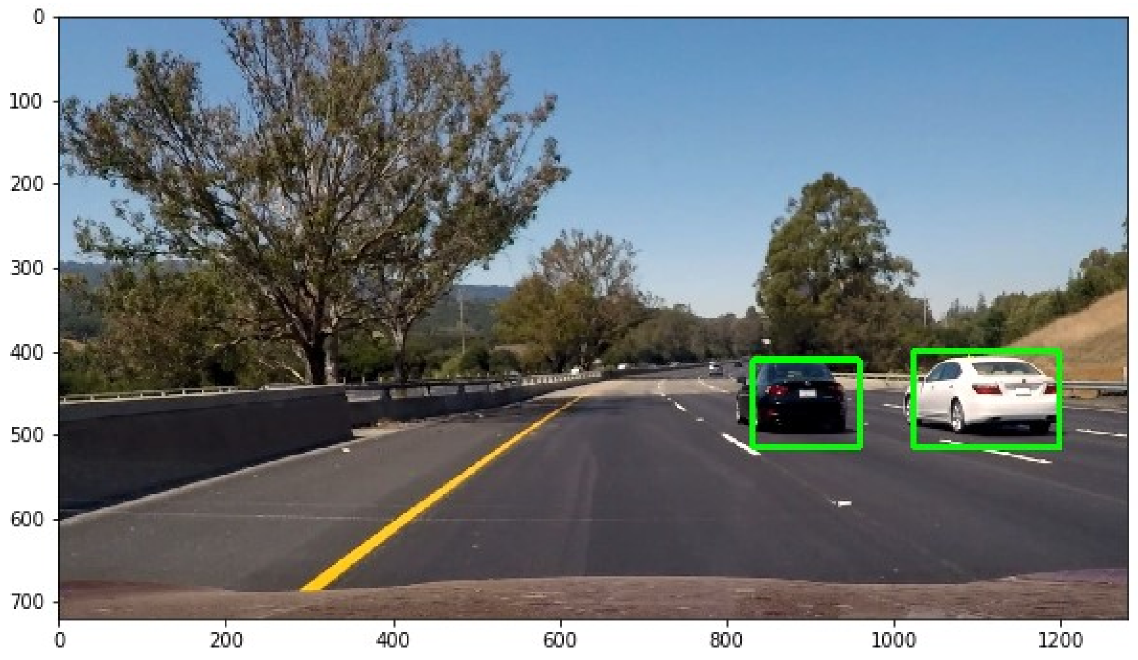
####2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?



0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



The result on the images worked pretty well, identifying the vehicles without false positives.

To be honest, the first implementation didn’t work that good, I had to tune several combinations of parameters and especially taking into account all the YUV channels, and ramping up the pixels per cell from 8 to 16; which subsequently represented a good compromise, as mentioned at the beginning, in term of execution speed and accuracy.

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 is attached along with the rest of the project material.

####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 detect vehicles on videos is situated in the ‘Pipeline for video‘ (32th cell) which is the same previous described, with the difference that I introduced a store for rectangles detections through a class. The detection of the last 16 frames are combined and then added along with a variable threshold of the heatmap, so that the latter could depend on the recent history of the previous rectangles.

###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 main issue was all about to try reach an acceptable detection accuracy without overlooking the speed of execution of the algorithm. A positive measure to cope with this situation was to include previous frame in order to reduce missclassification, though some cars could haven’t been right labelled when they change positions from one frame to the next.

Another cumbersome problem was represented when the background environment and lighting conditions are closer to the color of the car, in this case the algorithm fails since it found difficulties to skim the vehicle from the rest.

Moreover, I noticed that distant cars were difficult to detect, so the detection ability reduces along with the distance of the objectives.

Improvements:

- Use a convnet to preclude the sliding window search.
  (I wasn’t able to perform this approach seeing that I don’t have the right hardware to do that)
- Predict the locations of the vehicles in the next frames