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.

**1. Provide a Writeup / README that includes all the rubric points and how you addressed each one.**

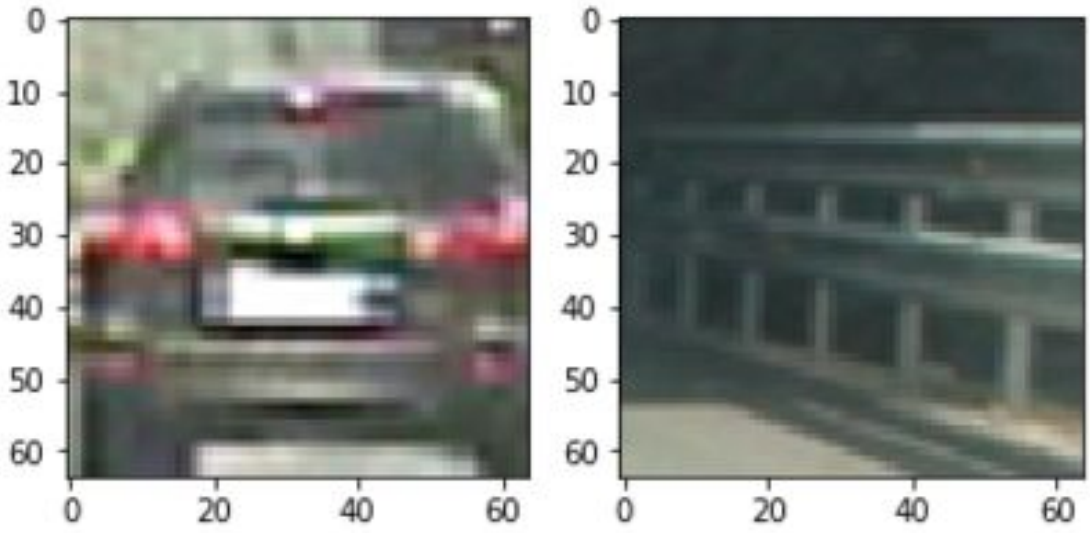
You're reading it!

## Histogram of Oriented Gradients (HOG)

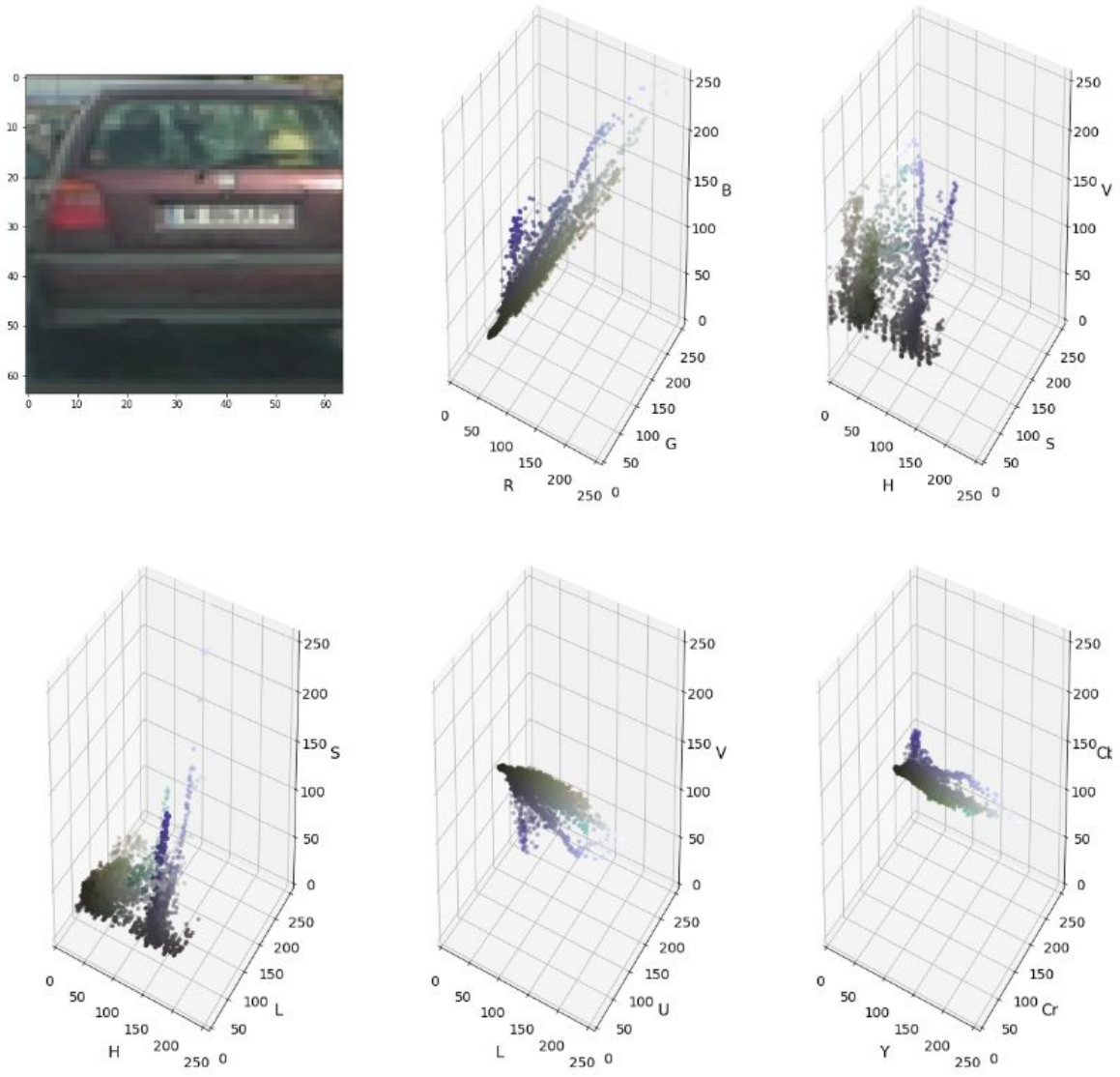
**1. Explain how (and identify where in your code) you extracted HOG features from the training images.**

The code for this is a little distributed throughout the ipython notebook.  The function definition to do this is in get_hog_features(), located in lines 123 - 133 of cell block 4.

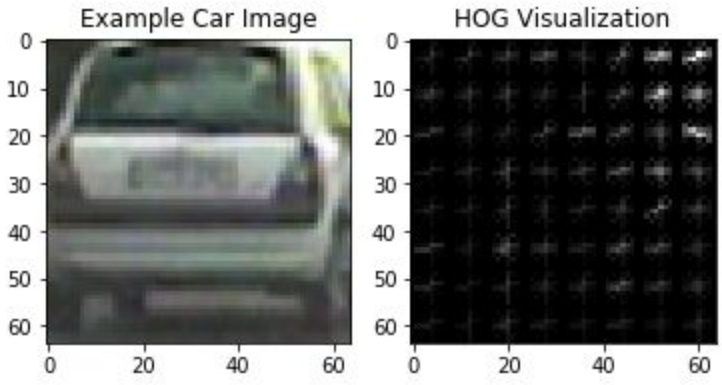
I started by reading in all the vehicle and non-vehicle images. Here is an example of one of each of the vehicle and non-vehicle classes:




I then explored different color spaces and different skimage.hog() parameters (orientations, pixels_per_cell, and cells_per_block). I grabbed random images from each of the two classes and displayed them to get a feel for what the skimage.hog() output looks like.  Of particular use was using my own modified version of the plot3d() function given in class.  Below is sample output from that:




Here is an example using the YCrCb color space and HOG parameters of orientations=8, pixels_per_cell=(8, 8) and cells_per_block=(2, 2):




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

I tried various combinations of parameters and color spaces, but it appears most publications for this type of thing recommended using the YCrCb color space. I initially thought that using fewer pixels per cell would yield a better result, but when I used too few the amount of time to run the video pipeline became unwieldy. I ultimately ended up using basically the same parameters as those tested in class.

**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).**

Training was first started using the extract_features() function defined in lines 138 - 187 of code block 4. This function is slightly modified from the one presented in class, in that it allows me to concatenate spatial color features, histogram features, and HOG features. These feature vectors are then passed to the prep_data_training() function in lines 275 - 288 of cell block 4, which returns scaled feature vectors. The scaled feature vectors and corresponding labels are then sent to the Train_LinearSVC() function defined in lines 299-308 of code block 4. The Train_LinearSVC() function simply calls

the scikit learn LinearSVC() function to train the linear support vector classifier, and returns time to train and prediction accuracy.

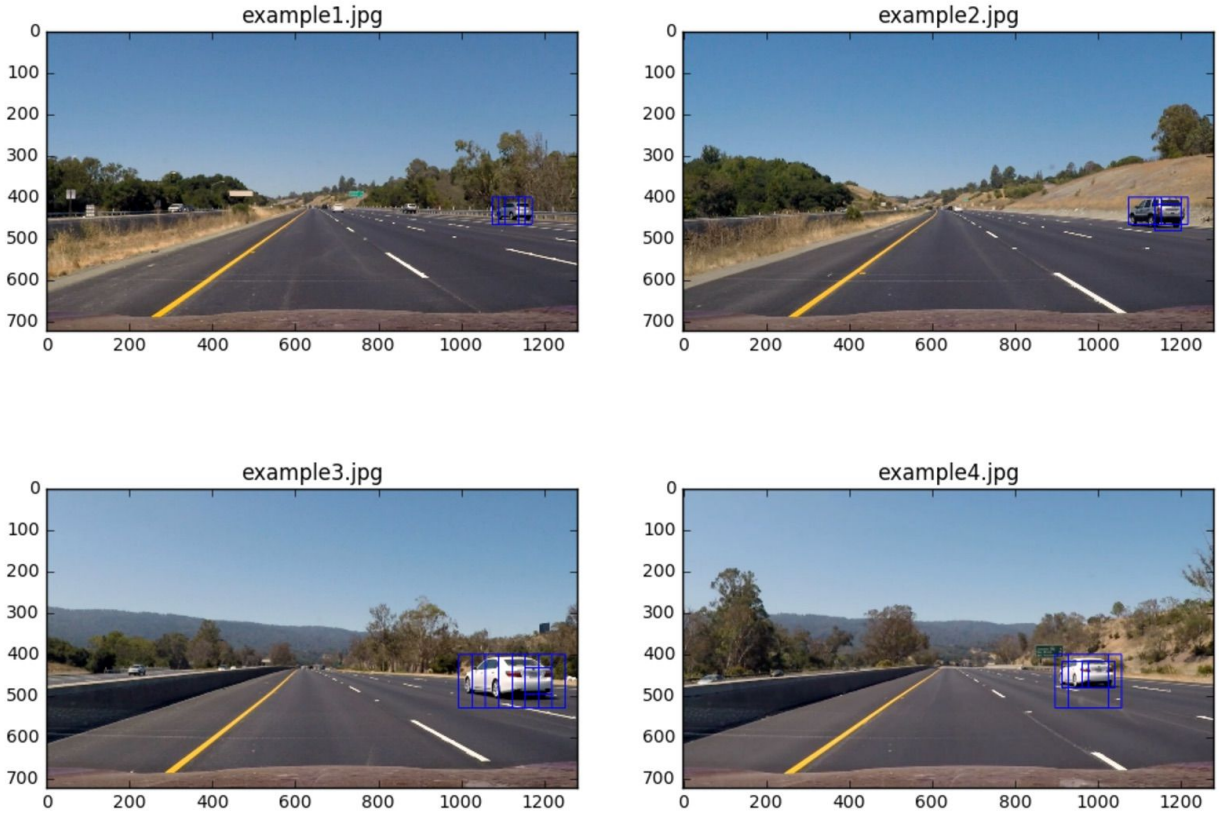
## Sliding Window Search

**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 order to save time I used fix scale sliding windows and through trial and error decided on an overlap of [0.85,0.85]. I found that if I was smart about the region of interest of the image that I searched through I did not need to use multi-scale for this. I think that moving forward using multi-scale windows would make the algorithm more robust and would help out at the end when the white car is very close to our car.

**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.)**

[Here is my video]

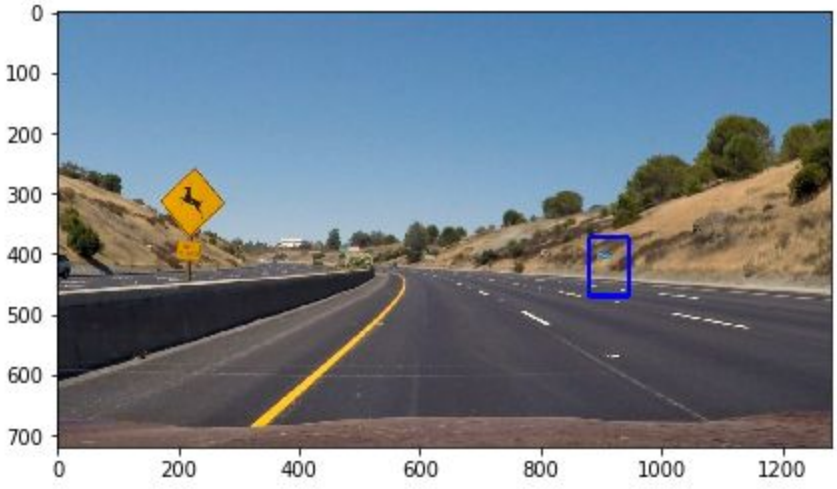
**2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.**

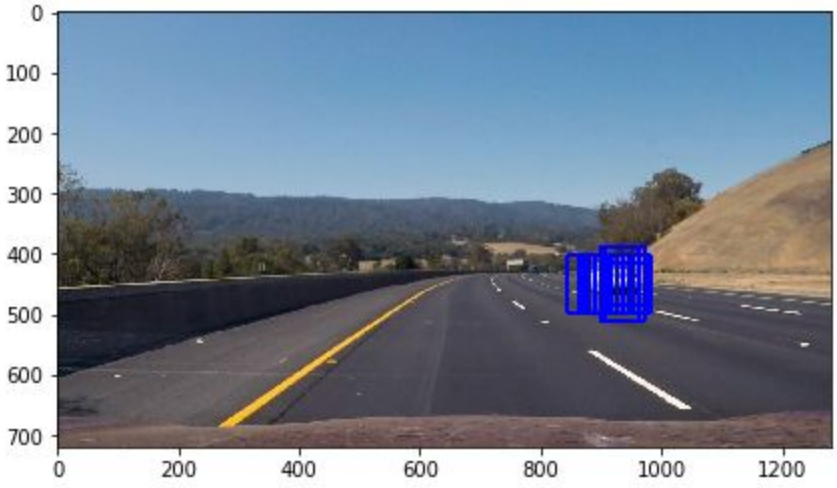
The test code that I used to develop code for use in the video pipeline is located in the *test_heatmap()* function defined in lines 522 - 547 of code block 4.

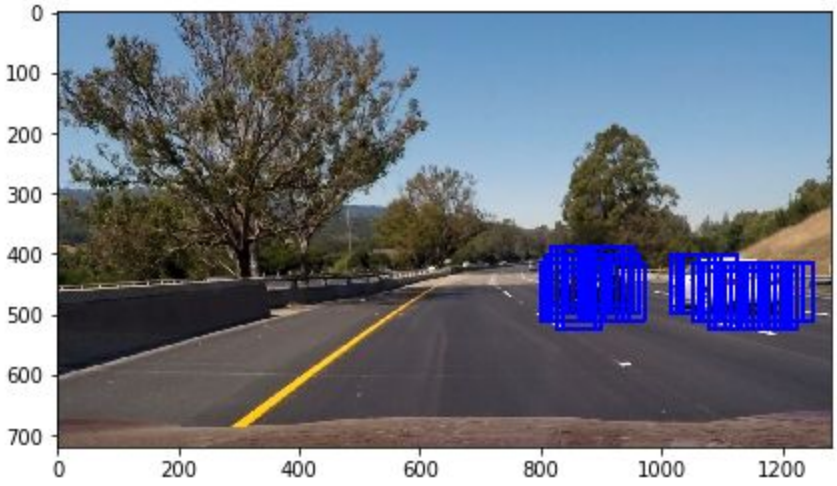
I recorded the positions of positive detections in each frame of the video. From the positive detections I created a heatmap and then thresholded that map to identify vehicle positions; I settled on using a threshold of 5. I then used `scipy.ndimage.measurements.label()` to identify individual blobs in the heatmap, and then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected using the draw_labeled_bboxes() function given in class and defined in lines 507 - 520 of code block 4.

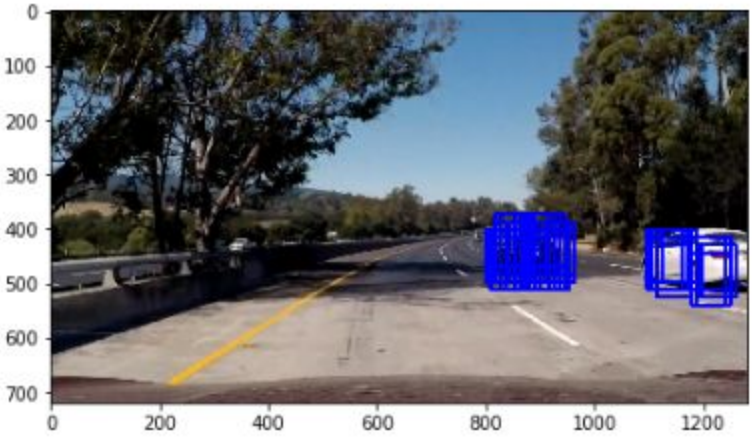
The video pipeline primarily does this using the two methods within the VehicleDetector class (lines 556 - 609), with the help of the add_heat() function given in class, which is defined in lines 489 - 497 of code block 4.

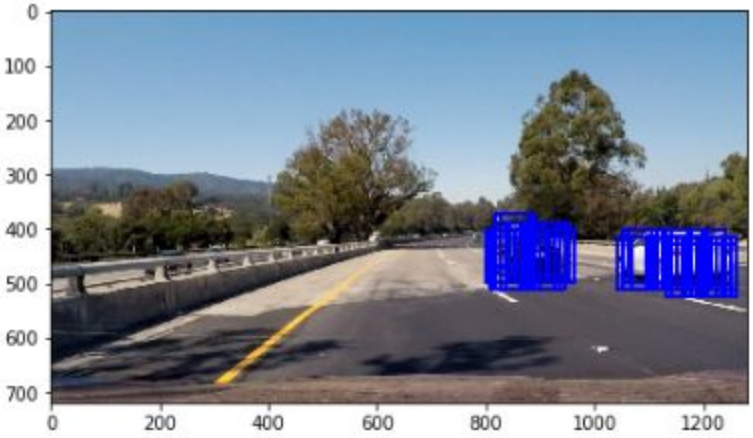
## Here are six frames and their corresponding bounding boxes, heatmaps, and resulting bounding boxes:

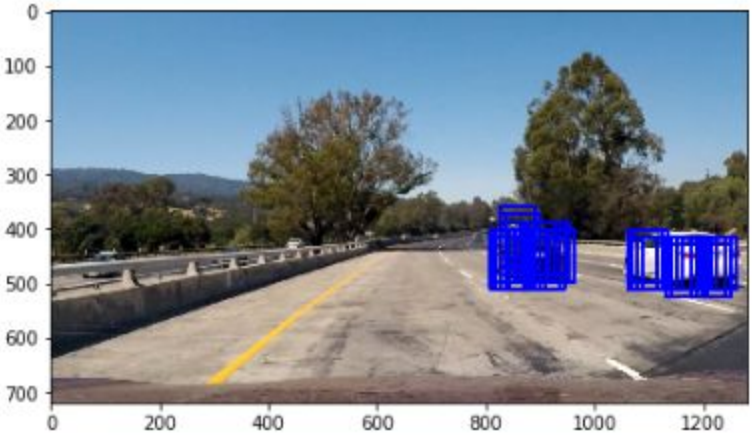

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

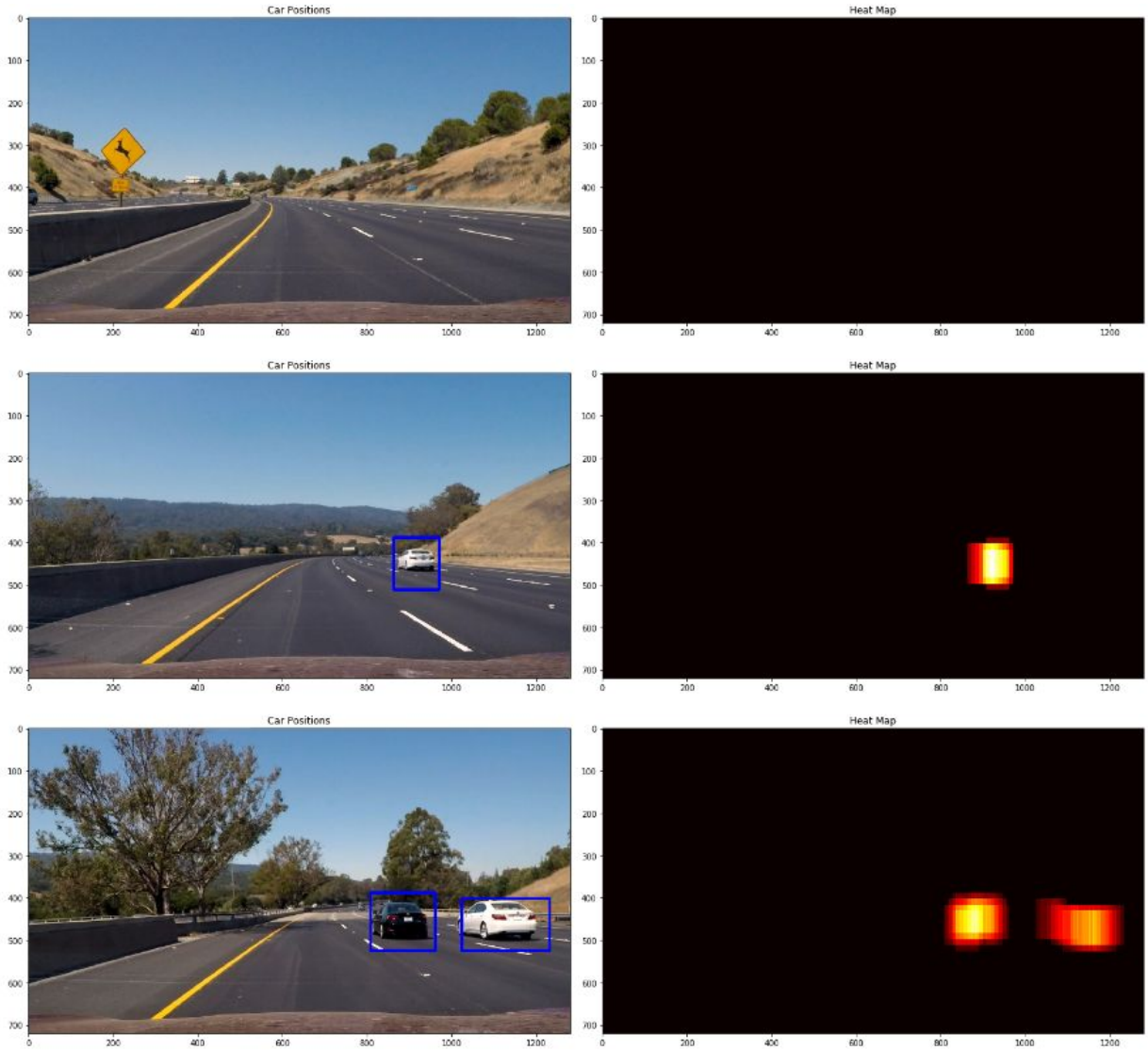

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

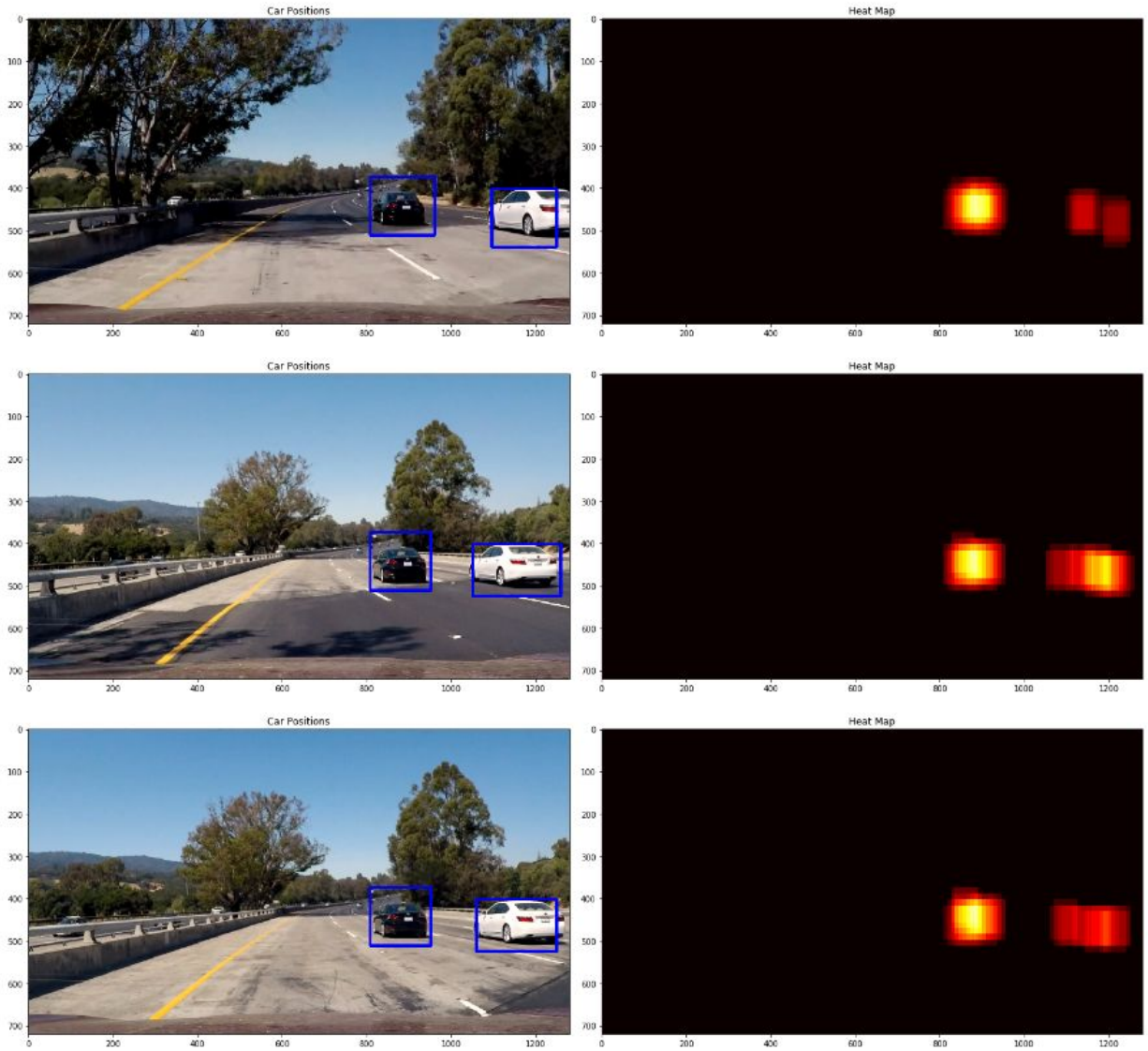

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


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


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


Car Positions
Heat Map
Car Positions
Heat Map
Car Positions
Heat Map


Car Positions
Heat Map
Car Positions
Heat Map
Car Positions
Heat Map

## 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?

Ultimately I didn't have many problems worth mentioning getting the video pipeline up and working; mostly due to the large number of helper functions that were given in class and the large amount of skeleton code we already had.  The data sets provided were very balanced so I didn't have to do any data set preparation before training.  The lighting in the project video was good, so that was not a concern, and the car stayed in one lane the whole time.

I believe my pipeline is most likely to fail when vehicles are up close and very large due to the fact that I chose not to use multi-scale sliding windows, as well as when our vehicle is not in the far left lane.  If our test and project video had cars that were directly in front of our car, or our car was in a different lane, I would have been forced to search over a larger region of interest in the image and would have had to use multi-scaled windows.  Other difficulties would arise when our vehicle is at an intersection or traffic light because cars will appear in much different positions.

I think one thing that would be really interesting to do to improve this would be to couple this with data from the car such as throttle and steering angle, and then fit a dynamical model to the detected cars to see if they are ever getting close to hitting the car we are riding in.  Other improvements could come from this model creation as well that would allow us to still predict that a car is there even though it may be occluded by something (since cars have inertia and can't instantly disappear).