Carnd Term1 Proj5 Vehicle Detection

Joe Zhou - ibalpowr@gmail.com

This project on vehicle detection is the last project of Term1 in Udacity Car Nano Degree. My approach to this detection problem involves SVM classification, HOG features extraction, sliding window search, and heatmap filtration. An excellent reference paper is the classic Dalal-Triggs [2005].

I. Explore Datasets

The vehicles and non-vehicles datasets, provided by Udacity, are based on GTI and KITTI datasets. The vehicles dataset has 8792 of images in the shape of 64x64x3 and the majority images are in rear view. The non-vehicles dataset has 8968 of image in the same shape and the images are road side features. I like to point out that some non-vehicles images are false positives extracted by hard-mining. Below are ten random examples from each dataset.

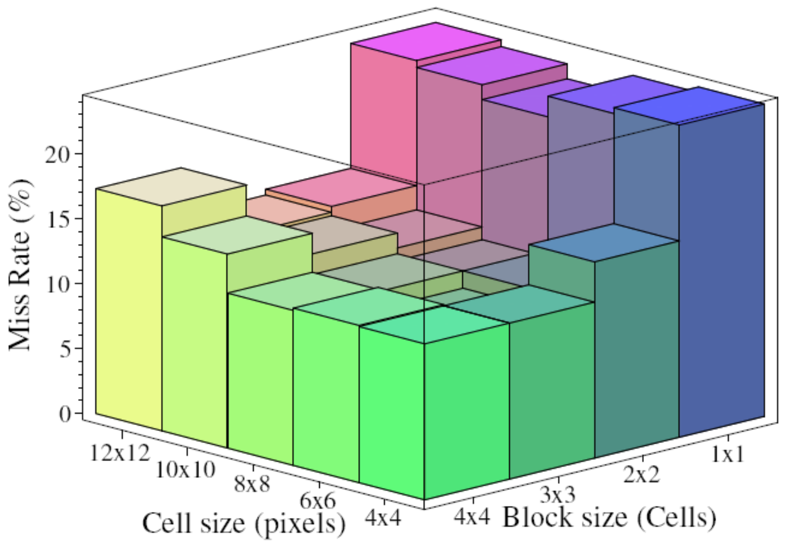
The ground truth labels are created by a hack, using two numpy functions np.ones( ) and np.zeros( ), since there are only two classes. These two datasets are stacked then split randomly into 80% for training and 20% for testing.

II. HOG features

HOG stands for Histogram of Oriented Gradients [Stanford CS231A Winter 2015 Lecture 13]. It is a feature descriptor, like SIFT. To get the HOG features, I select the hog function from SKImage package and it has three key parameters, pix_per_cell, cell_per_block and orientation, which should be chosen according to the sizes of important features in the training images. For example, in Dalal-Triggs [2005], "6-8 pixel wide cells do best irrespective of the block size – an interesting coincidence as human limbs are about 6-8 pixels across in our images." For a vehicle, tail lights, tires/wheels and window shapes are distinct features. After trying a few combinations as shown below, I settle with orient = 10, pix_per_cell = 8, cell_per_block = 2.

color space	orient	pix_per_cell	cell_per_block	SVM accuracy
YCrCb	10	8	2	0.9825
YCrCb	9	6	2	0.9775
YCrCb	8	8	1	0.978

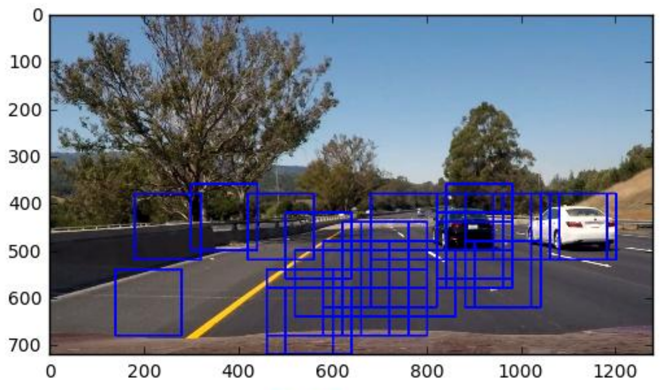
Furthermore, my choice of parameters fits into the analysis from the reference paper, as shown by the diagram below.

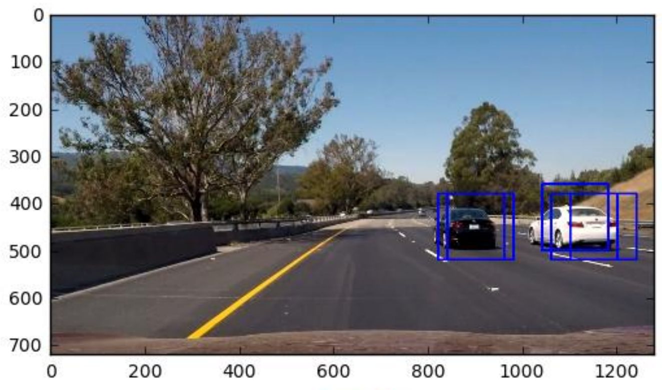


[Diagram from Dalal-Triggs 2005]

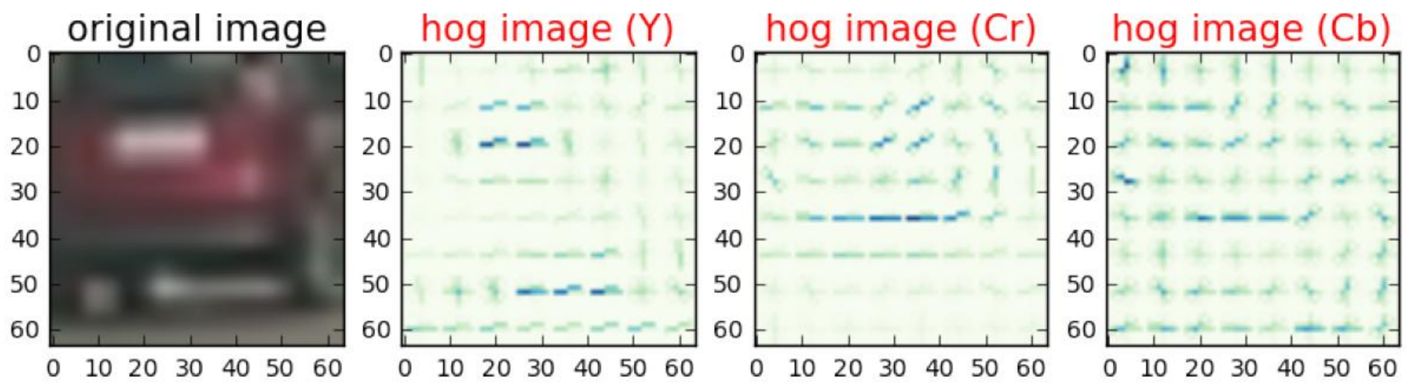
I have tried a few color space options and YCrCb provides the best accuracy. For example, using all three channels, YCrCb can reach accuracy of 98% while RGB has only 96% and HLS has the lowest as 95%.

Even though the difference in accuracy is small, the number of false positives can be large because the number of sliding windows is in the range of thousands. For example, as shown below, the number of positive detections in RGB is much higher than in YCrCb under the same settings except for their choices of color space.

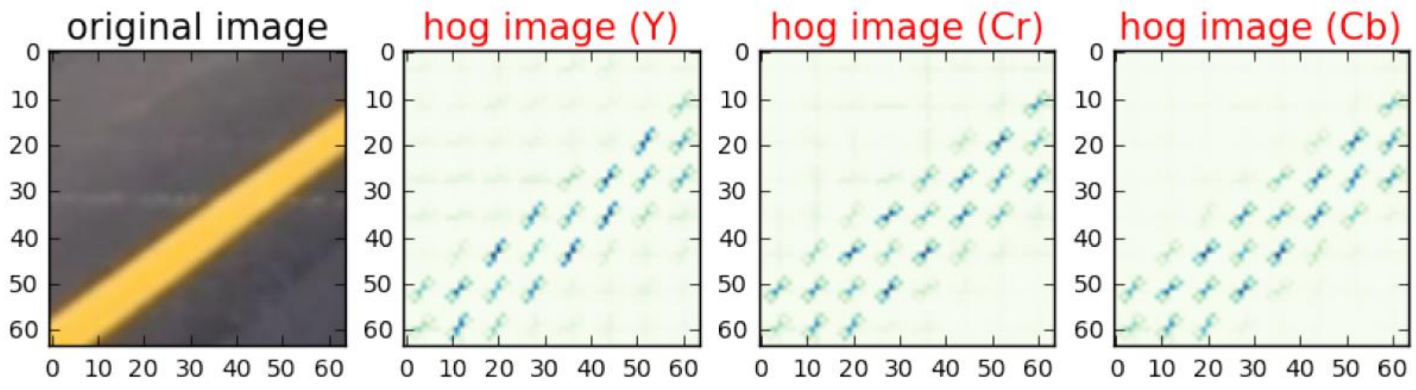

RGB


YCrCb

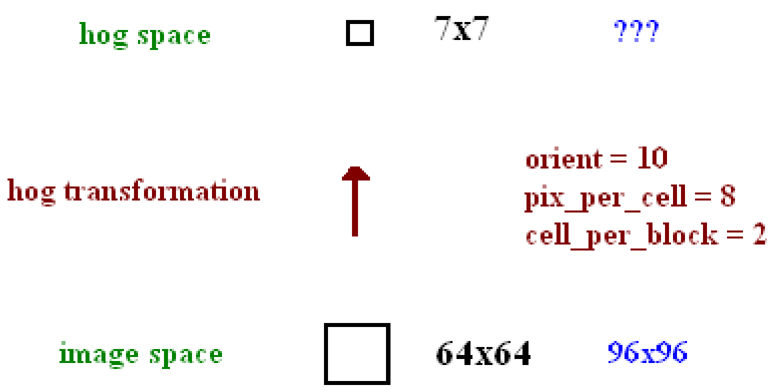
Also I use all three channels since each channel provides different features, as shown by an example of vehicle.



And the HOG image for a non-car sample image is shown below.



The sizing in image and hog spaces is interesting. For example, what is the size of the hog image in below if we change the kernel size from 64 to 96?



In order to simplify the HOG transformation process, I come up with two neat functions to automatically calculate the correct HOG and image sizes, as shown below.

```
# from hog-svm paper [2005]
# find hog size ... given image size, pix_per_cell, cell_per_block
def find_hogsize(imagesize, pix_per_cell, cell_per_block):
    block_overlap = np.ceil(cell_per_block/2)
    return np.int(np.floor(1+(imagesize//pix_per_cell-cell_per_block)/
                           (cell_per_block-block_overlap)))

# the inverse of find_hogsize()
# find image size
def find_imagesize (hogsize, pix_per_cell, cell_per_block):
    block_overlap = np.ceil(cell_per_block/2)
    return np.int(np.floor(((hogsize-1)*(cell_per_block-block_overlap)+
                           cell_per_block)*pix_per_cell))
```

So, for above question, the correct hog size is find_hogsize(96,8,2) = 11.

III. SVM Classifier

SVM stands for Support Vector Machine. It is a machine learning classifier. It computes a score S = W * x + b, where W and b are weight and bias, and x is an input image. Since there are only two classes: vehicle and non-vehicle, we can use the sign of the score S, sign(score) = label, to determine the class label.

Before applying the SVM classifier, I use SKImage's hog function to extract the hog features based on the parameters I select in the previous section. Below are the codes for extracting vehicles hog features. Codes for non-car hog features are similar.

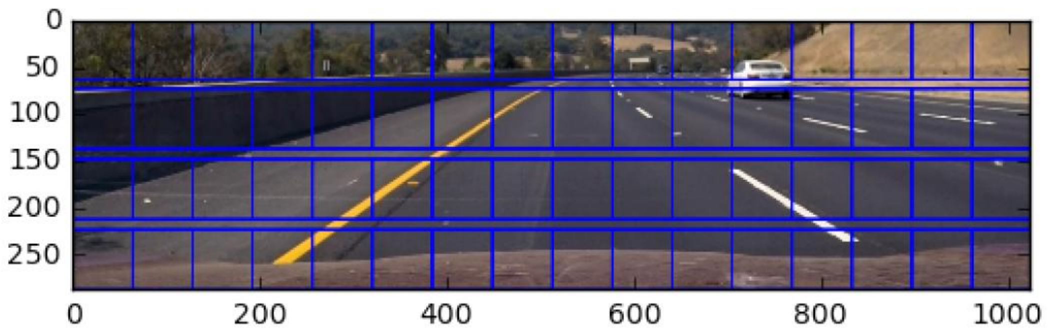
```
vehicle_features = []
for i in range(len(vehicles)):
    image = cv2.cvtColor(vehicles[i], cv2.COLOR_RGB2YCrCb)
    vehicle_channel_features = []
    for channel in range(image.shape[2]):
        vehicle_channel_features.append(hog(image[:, :, channel], \
                        orientations=orient, \
                        pixels_per_cell=(pix_per_cell,pix_per_cell), \
                        cells_per_block=(cell_per_block,cell_per_block), \
                        transform_sqrt=True, \
                        visualise=False, \
                        feature_vector=False, \
                        normalise=None))
    vehicle_channel_features = np.ravel(vehicle_channel_features)
    vehicle_features.append(vehicle_channel_features)
```

In this project, I use SKLearn's Linear SVC as the classifier and SKLearn's StandardScaler to remove mean and scale the image to have unit variance in the preprocessing stage. And their codes are below.

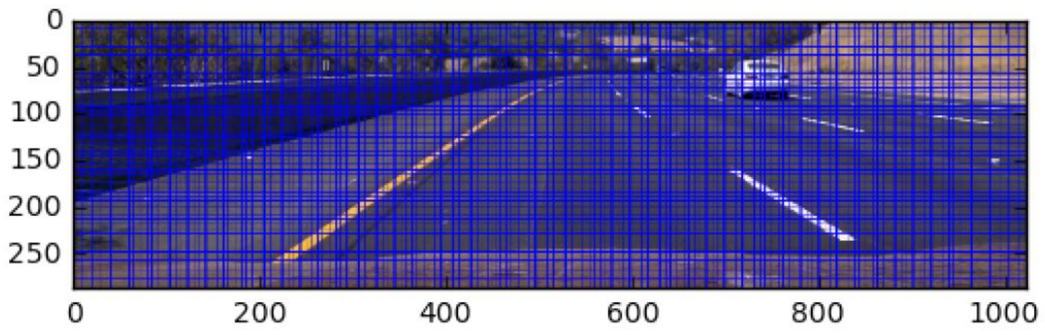
```
# for normalizing dataset
from sklearn.preprocessing import StandardScaler

# for SVM classifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split

# compute the mean and std to be used for later scaling
X_scaler = StandardScaler().fit(X)
scaled_X = X_scaler.transform(X)

X_train, X_test, y_train, y_test = \
    train_test_split(scaled_X,labels,test_size=0.2,random_state=86)
svc = LinearSVC()
svc.fit(X_train, y_train)
```

IV. Sliding Window Search

Sliding window search is a brute force approach. It is inefficient comparing with region proposal method [Stanford CS231N Winter 2016 Lecture8]. To alleviate this problem a bit, I choose to scan only the lower half of the image. And to keep it simple, I choose to have all sliding windows (aka kernels) the same size as 64x64. Now, I only have parameters to play with, scale and overlap. I start with no overlap and select a scale in which the distance vehicle can fit into a 64x64 sliding window. Below is an image in which the scale is set to 0.8. The white vehicle fits nicely inside the 64x64 so I choose 0.8 as one of the scale.



After several trial and errors, I settle with two scales 0.7 and 0.8 and the overlap as 6 to produce one of best detection results. Below is a test image example overlaying with a total of 2304 sliding windows under the parameter setting as scale = 0.8 and overlap = 6.

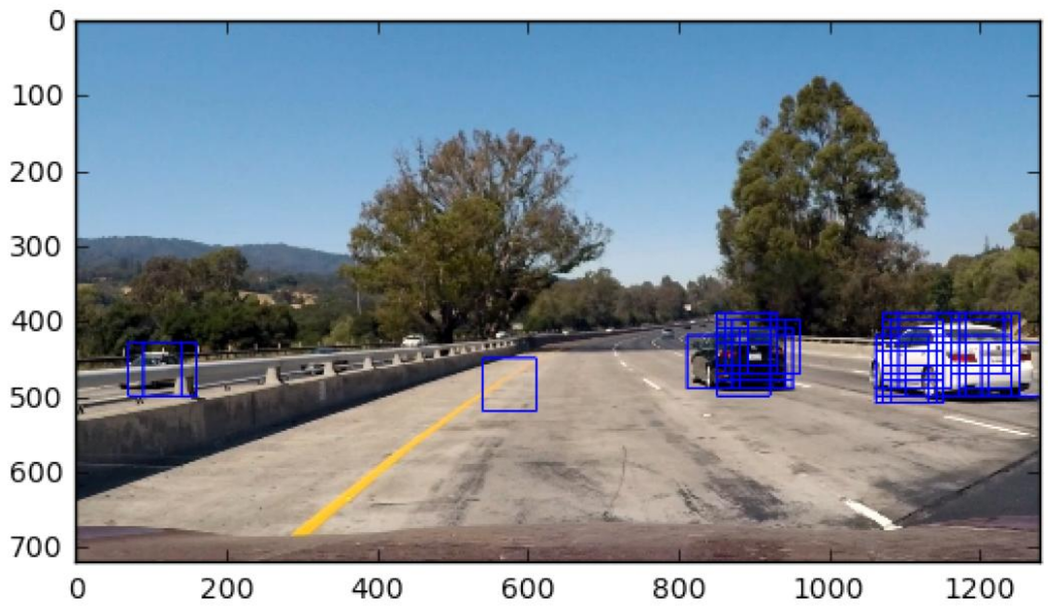


V. Use heatmap to suppress false positives

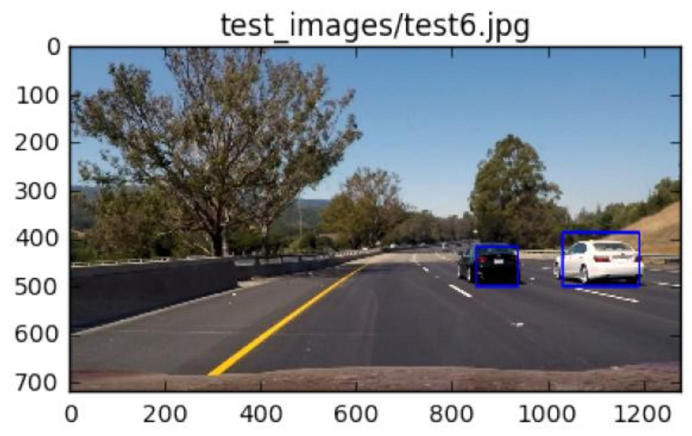
To remove false positives, I apply heatmap, which a method of non-maxima suppression. To implement heatmap, I choose the label function from scipy.ndimage.measurements package. Below is the code.

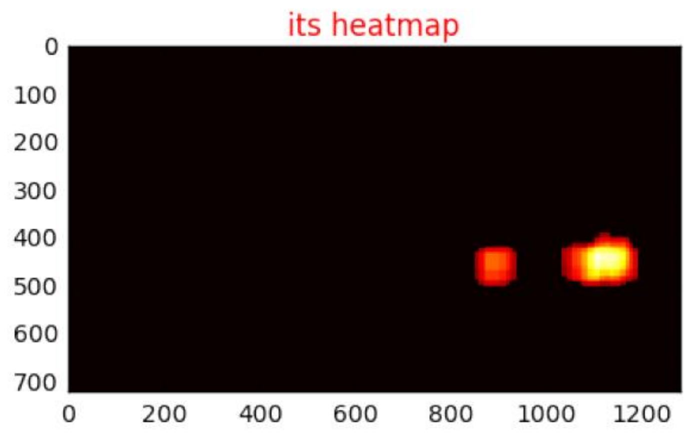
```python
def car_cluster(detections_history, image, threshold=1):

    # create a image-size heatmap
    heatmap = np.zeros((image.shape[0],image.shape[1])).astype(np.float)

    # update the heatmap
    for bbox in detections_history:
        (x1,y1,x2,y2) = (bbox[0],bbox[1],bbox[2],bbox[3])
        heatmap[y1:y2, x1:x2] += 1

    # set lower bound
    heatmap[heatmap < threshold] = 0
    # set upper bound
    heatmap = np.clip(heatmap, 0, 255)

    # make car clusters
    labeled_heatmap, car_number = label(heatmap)

    cars = np.empty([0, 4], dtype=np.int64)
    for car in range(car_number):
        nonzero = (labeled_heatmap == (car+1)).nonzero()
        (x1,y1,x2,y2) = (np.min(nonzero[1]),np.min(nonzero[0]),
                          np.max(nonzero[1]+1),np.max(nonzero[0]+1))
        cars = np.append(cars,[[x1, y1, x2, y2]], axis=0)

    # (x1,y1,x2,y2)   ...   image_size heatmap
    return (cars, heatmap)
```

Below is the detection result before heatmap applied.

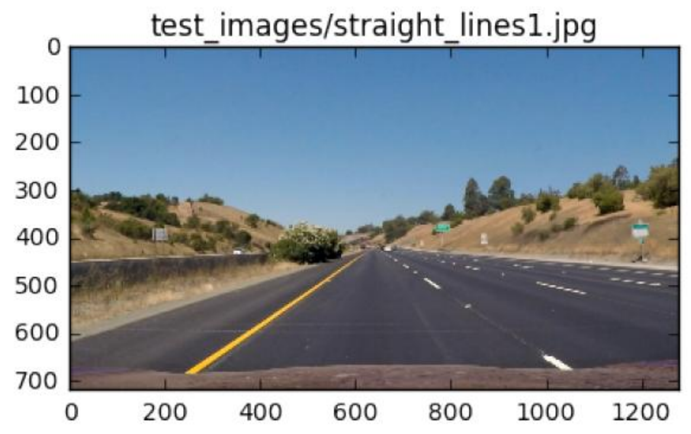


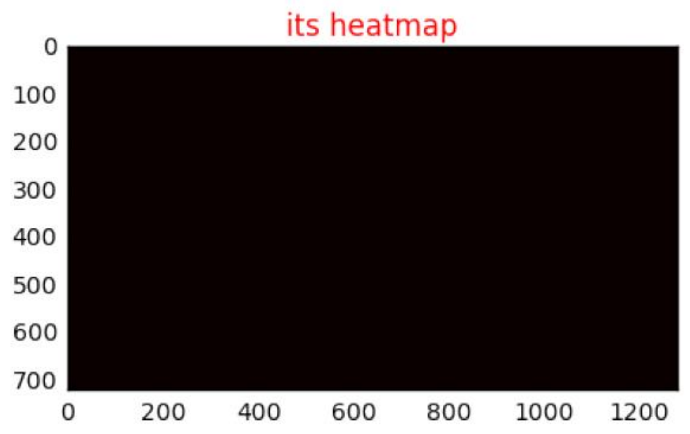
In order to suppress the false positives in the left side of the image, I set the threshold to 6 after several trial and errors. Below are the results after heatmap applied.

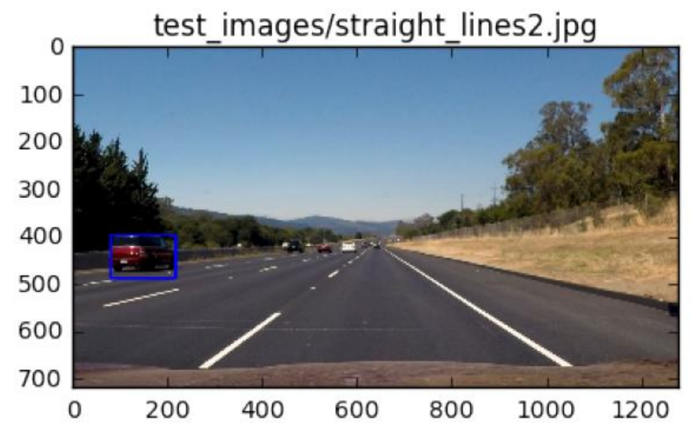


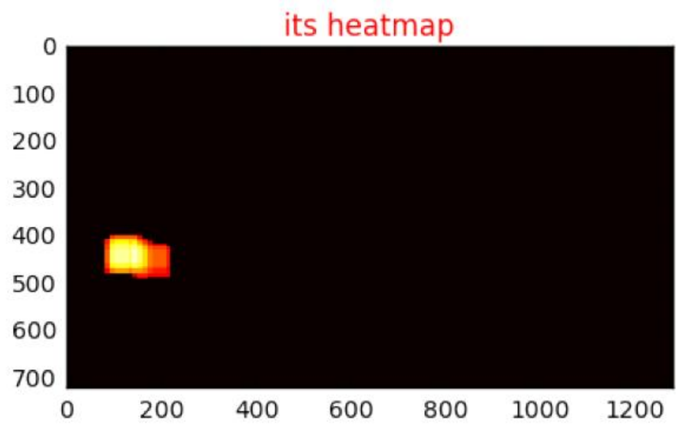


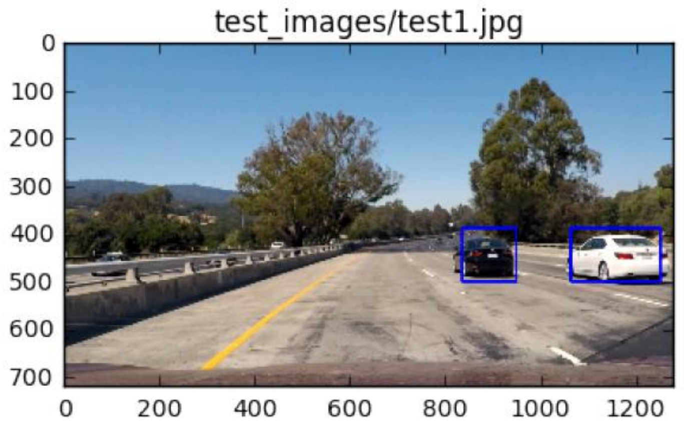
Below are the results for the rest of test images.

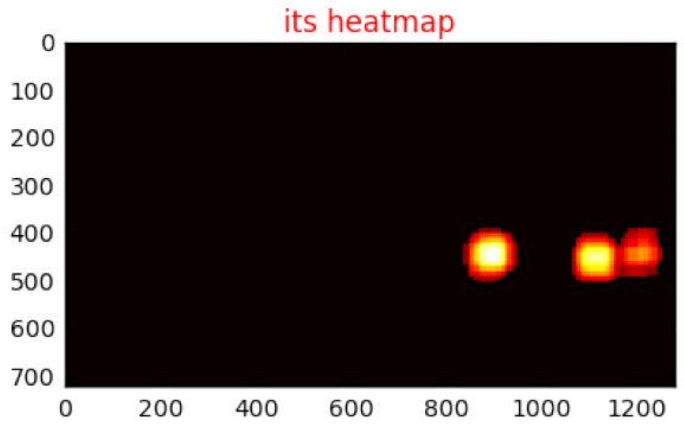


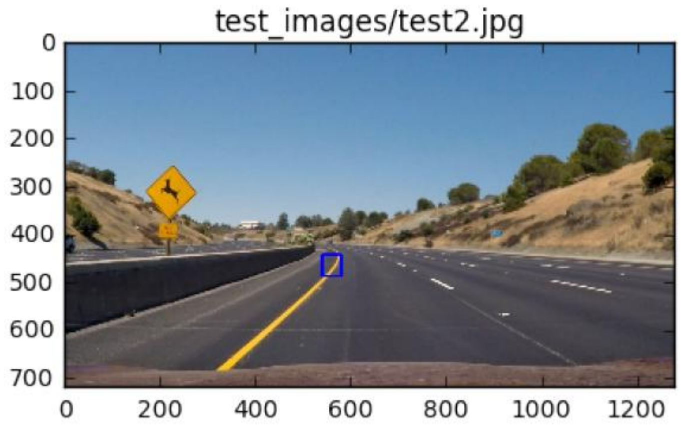


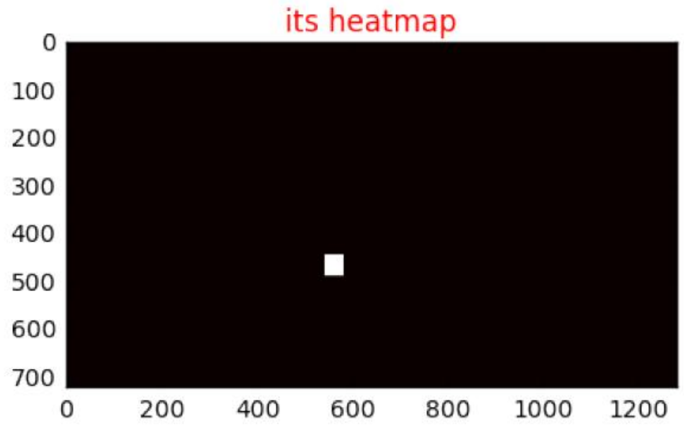


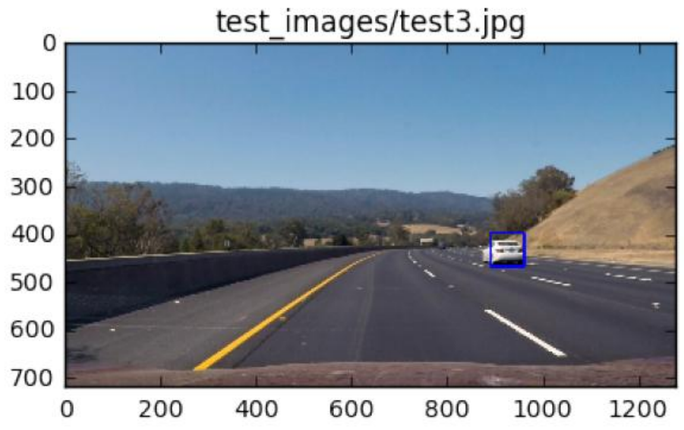


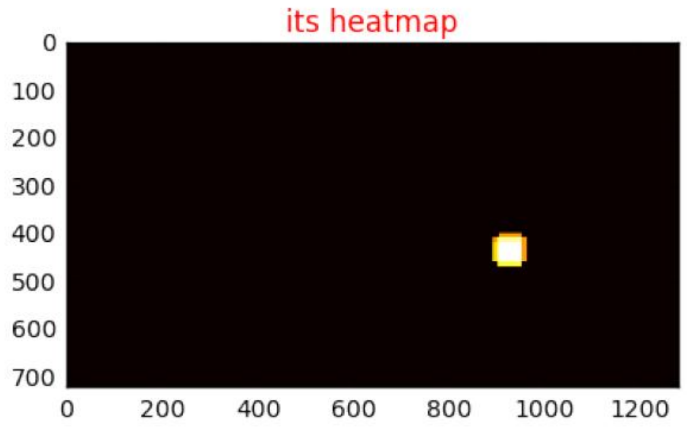

test_images/test1.jpg

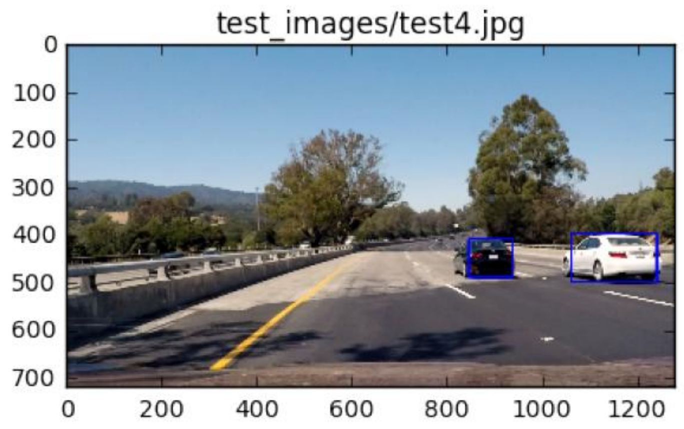

its heatmap

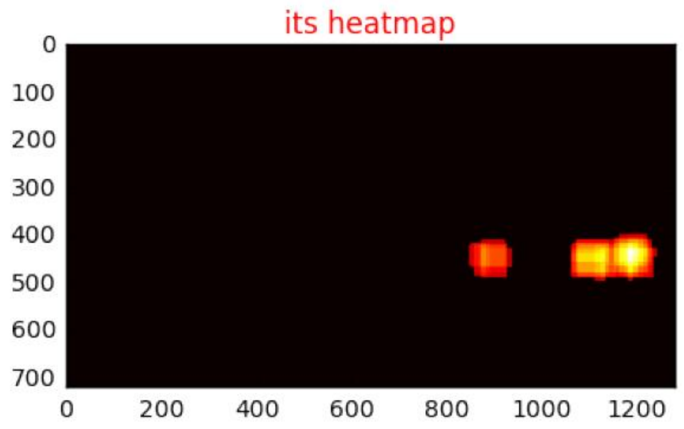

test_images/test2.jpg

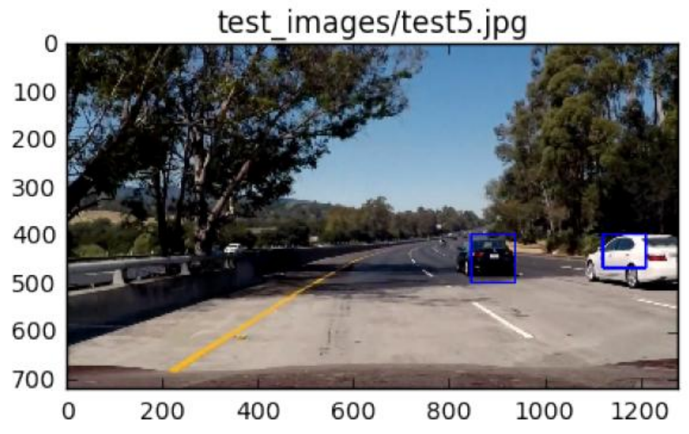

its heatmap

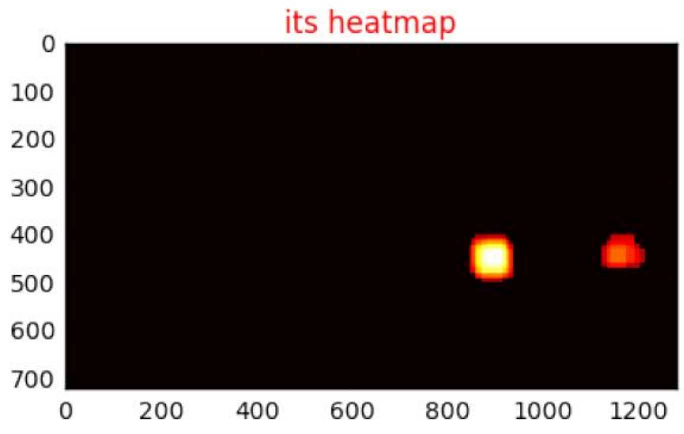

test_images/test3.jpg


its heatmap









The results are pretty good, with only one false positive in test2.jpg.

VI. Video Implementation

In order to buffer several successive frames in a video, I place the detection pipeline into a class and use deque function from "collections" package. It takes the advantage that an object more likely to appear in the same location in adjacent frames in a video. Heatmap is a greedy way of merging nearby bboxes into a larger one. Below are the codes.

```
class Car_Detection(object):
    def __init__(self,frame):
        self.detections_cars = deque(maxlen=10)
    def process(self,frame):
        cars_frame,_ = detect_frame(frame)
        self.detections_cars.append(cars_frame)
        cars,_ = car_cluster(np.concatenate(
                            np.array(self.detections_cars)),frame,2)
        for c in cars:
            cv2.rectangle(frame, (c[0],c[1]),(c[2],c[3]),(0,0,255),4)
        return frame
```

The result video is uploaded to github.

VII. Discussion

I see two major issues in the result video. The first one is that the detection pipeline behaves less expected in *vehicles cluster*. The second is that the IoU's can be low in certain time frames and the bbox does not cover the whole vehicles. These two issues are partially due to the bias in the training dataset where the majority of the images are from the rear view. So there is a need to expand the dataset and make the dataset more comprehensive.

Another nuisance is that the detection pipeline can be slow, hardly making it real time. This slowness is due to very large number of sliding windows and the large HOG feature sizes. To speed it up, we can use cascade approach. For example, tires/wheels are excellent feature in the first stage of cascade to eliminate many false positives.

One more downside, in my point of view, is that this plain vanilla SVM detection pipeline does not scale well into multiple objects, such as traffic signs and pedestrians.