Vehicle Detection P5

1. Project Goals

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.

2. Rubric Points

In the following I will describe how I approached the different Rubic points.

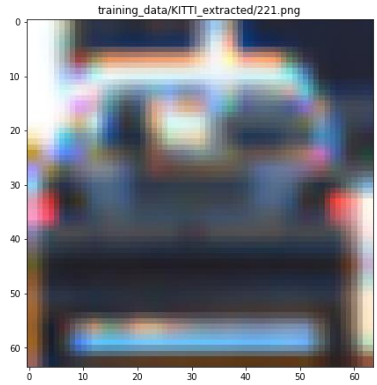
A) **Histogram of Oriented Gradients** (HOG)

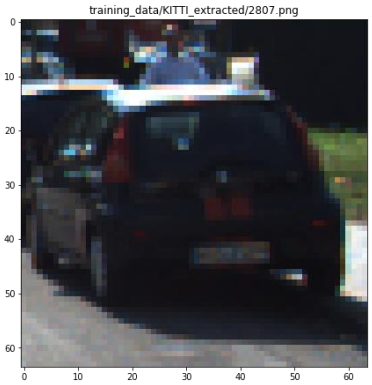
1. Explain how you extracted HOG features from the training images

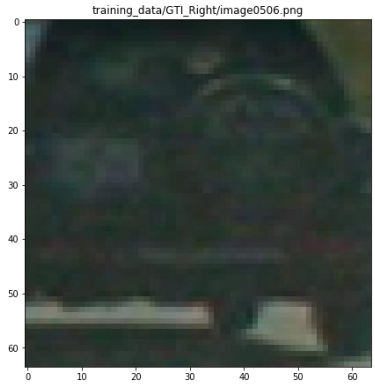
First I created two lists from the given training-sets containing cars and non_cars class objects, see *code cell 2 of the IPython notebook*. The two lists were almost balanced in terms of number cars and non_cars class objects, see report below

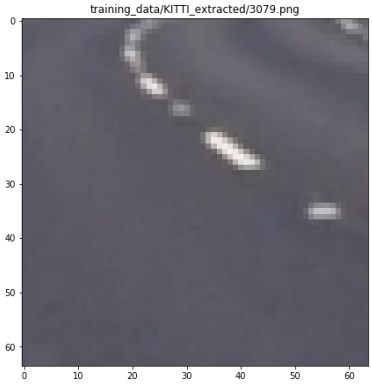
```
NOTE: size of cars features     8792
NOTE: size of non cars featues  8968
```

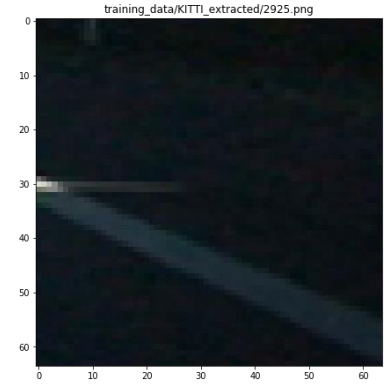
Then I displayed 3 examples of the cars and non_cars class objects, which were selected randomly, see *code cell 3 of the IPython notebook*. Below there are the example images.



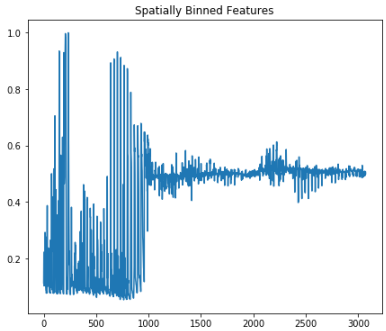


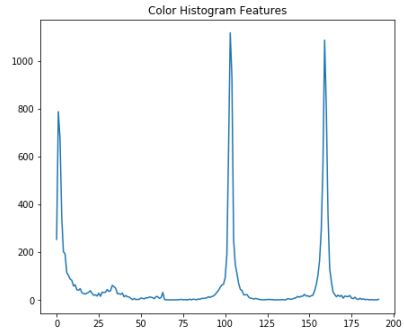


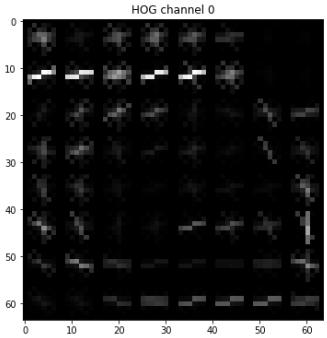


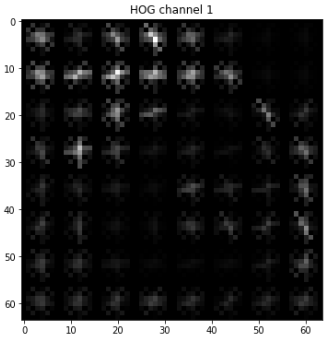


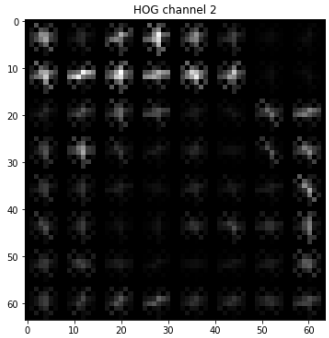
Then I ran spatial-, histogram color- and HOG feature extractions using different color spaces the on the car and non_car class objects. I used the function `extract_features()` which uses the `skimage.hog()` and the functions `bin_spatial()` and `color_his()`, `see` *code cell 4 of the IPython notebook*.
Here I explored different parameter-sets like color spaces, spatial size, bin histograms and orientations, pixels_per_cell, and cells_per_block for HOG, see *code cells 5, 6 of the IPython notebook*.
The figures below present feature extraction results of the cars and non_cars object class examples using the parameter set: color space = YCrCb, spatial size = (32, 32), bin histograms = 64, orientations = 9, pixels_per_cell=(8, 8) and cells_per_block=(2, 2), see *code cell 7 of the IPython notebook*.

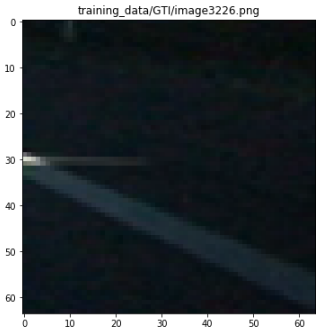


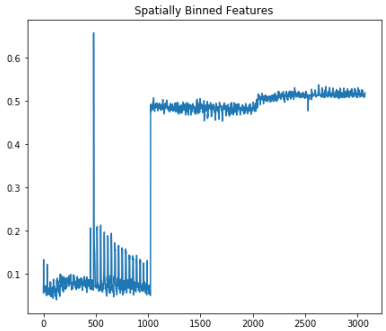


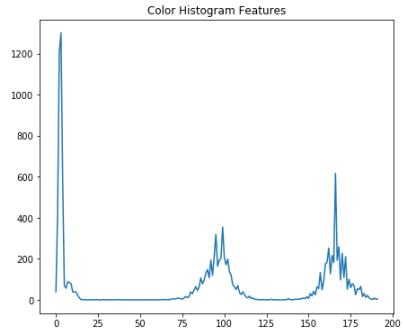


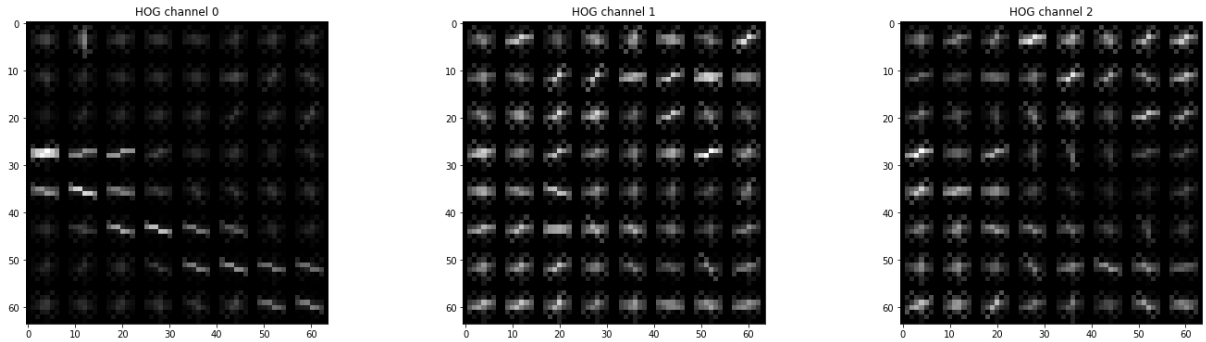












2. Explain how you settled on your final choice of HOG parameters

I tried various combinations of parameters for spatial, bin histogram, HOG and color spaces based on the experience from the quiz-lessons and the visual results of the related feature extractions.
Finally, I did choose the parameter-set based on the test accuracy of the classifier. My classifier did perform best = test data-set accuracy of 99.27% using the parameter-set listed below, see *code cell 5 of the IPython notebook*:

```
colorspace = 'YCrCb' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = 'ALL' # Can be 0, 1, 2, or "ALL"
hist_bins=64
spatial_size=(32, 32)
```

Note: I could not explore the color map LUV because the HOG extraction generated on error message.

3. Describe how you trained a classifier using your selected HOG features

After I extracted the features from the car and non_car class objects, I joined both feature lists and normalized it as presented below, see *code cell 8 of the IPython notebook:*

```
# normalize training data
X = np.vstack((car_features, non_car_features)).astype(np.float64)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
scaled_X = X_scaler.transform(X)

# labels
y = np.hstack((np.ones(len(car_features)), np.zeros(len(non_car_features))))
```

Then I splitted the data-set into 80% training and 20% test data-set by using `train_test_split` from `sklearn.model_selection`, see *code cell 9 of the IPython notebook:*

```
from sklearn.model_selection import train_test_split

# Split up data into randomized training and test sets
rand_state = np.random.randint(0, 100)
X_train, X_test, y_train, y_test = train_test_split(scaled_X, y,
test_size=0.2, random_state=rand_state)
```

After this I trained the model/classifier using a linear SVM model trade-offing spatial, bin histogram, hog and color spaces as described before. Using the final parameter-set (previous section) I achieved 99.24% accuracy

on the test data-set, see *code cell 10 of the IPython notebook*

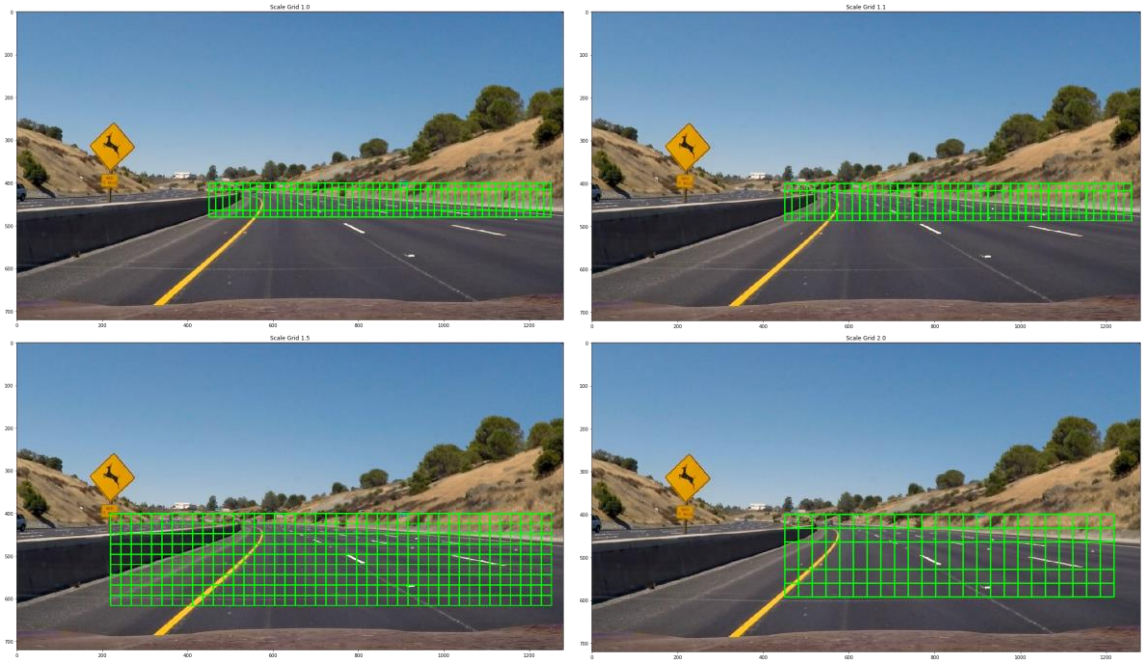
```
NOTE: 43.35 seconds to train SVC
NOTE: training accuracy of SVC =  1.0
NOTE: test     accuracy of SVC =  0.9927

NOTE: SVC sample predicts:  10
NOTE: predition hitrate
      predicted: [ 1.  1.  0.  1.  1.  1.  1.  0.  0.  0.]
      labels:    [ 1.  1.  0.  1.  1.  1.  1.  0.  0.  0.]
NOTE: 0.02062 seconds to predict for 10 labels with SVC
```

B) Sliding Window

1.Describe how you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

I used the sub-sampling sliding window approach as described in the lesson 35 for the sliding window concept. I added to the basic implementation debug functions and ROI for x coordinates, see function `find_cars()` of *code cell 11 of the IPython notebook*. The final number of searches and scales were implemented in the function `search_multiple()` of *code cell 12 of the IPython notebook*. After a lot of evaluations analysing the test images and project video, starting from 2 searches, I decided to use 4 searches using an overlap of 2 cells per step in order to avoid missing detections or split detection bounding boxes (the number of search windows could be definitely optimized further). Below there is the configuration of the search windows and the related grid screen shot at one of the test images:

1. Search 1: scale 1.0 (64x64), ROI: x = 450-1280 & y = 400-500
2. Search 2: scale 1.1 (70x70), ROI: x = 450-1280 & y = 400-510
3. Search 3: scale 1.5 (96x96), ROI: x = 400-1280 & y = 400-500, x = 220-1280 & y = 500-650
4. Search 4: scale 2.0 (128x128), ROI: x = 450-1280 & y = 400-650



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

The figure below presents the results of the vehicle searching function `search_multiple()` applied on the test images, see *code cell 12 of the IPython notebook*. The classifier classifies cars and non_cars objects well.

I tried to optimized the performance of the classifier by using as less searches as possible and by using ROIs to search inside areas of interest only (I think the performance could be optimized further, see point Discussion).

The detection pipeline uses the `search_multiple()` in order to receive the drawing boxes for the identified vehicles. The heat map, thresholding and labeling approach of lesson 37 were implemented inside the pipeline in order to filter out **false positives** and create a **bounding box**es for overlapping boxes, see *code cell 14 of the IPython notebook* for heat map, threshold & labeling functions and see *code cell 17 of the IPython notebook* for the standalone detection pipeline.

Note: The final detection pipeline used for the video, see *code cell 16 of the IPython notebook*, uses a **circular buffer** for the recorded drawing boxes. The rationale for the **circular buffer** is to integrate the heat map over a number of samples/images.

The figures below present the detection pipeline results plus the heat map after thresholding using the standalone detection pipeline, see *code cell 18 of the IPython notebook.*

C) Video

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video

The Github repository of the submission contains the final video output, named `project_video_output_final.mp4`. Additionally there is also a video output for the challenge, named `project_video_final_challenge.mp4`.

2. Describe how you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes

As stated in the previous section I implemented a **heat map, thresholding** and **labelling** approach of lesson 35 inside the detection pipeline for false positives and for combining overlapping bounding boxes using a **circular buffer** (in order to integrate the heat map over a number of samples/images). Additionally, I also filtered out false positives by **ROIs** (opposite road).

In the following I will explain how the heat map, thresholding, labeling and circular buffer were used inside the detection pipeline, see *code cell 16 of the IPython notebook*.

Circular Buffer, *code cell 15 of the IPython notebook*

- The circular buffer `hit_list` saves the drawing boxes for a number of images.

```
################################################################################
#
# Circular buffer for detection pipeline
#
################################################################################

import collections

CBUF = 10
hit_list    = collections.deque(maxlen=CBUF)
```

Detection PipeLine, function `detectPipeLine_()` of *code cell 16 of the IPython notebook*

- For each image the `search_multiple()` function passes the recorded drawing boxes to the circular buffer.
- If the circular buffer is full (2) then a "blank" heat map is created (2.0). After this, the heat map is "filled" by integrating over all saved drawing boxes of the circular buffer (2.1). Then a threshold is applied on the resulting heat map (2.3) in order to remove **false positives**. Then the `label()` function of `scipy.ndimage.measurements` is called to create combined bounded boxes (2.4-2.5). See code below:

```
def detectPipeLine_ (threshold, image, svc, X_scaler, orient, pix_per_cell, cell_per_block, spatial_size,
                     hist_bins) :

    global hit_list

    # 1. search for objects
    box_list = search_multiple (image,svc,X_scaler,orient,pix_per_cell,cell_per_block,spatial_size,
                                hist_bins)
    # add box_list to circular buffer
    hit_list.append(box_list)

    # 2. if buffer full run heatmap & labeling
    if (len(hit_list)==CBUF) :
        # 2.0 heat map
        heat = np.zeros_like(image[:,:,0]).astype(np.float)
        # 2.1 Add heat to each box in box list
        for box in hit_list :
                heat = add_heat(heat,box)
        # 2.3 Apply threshold to help remove false positives
        heat = apply_threshold(heat,threshold)
        # 2.4 Visualize the heatmap when displaying
        heatmap = np.clip(heat, 0, 255)
        # 2.5 Find final boxes from heatmap using label function
        labels = label(heatmap)
        draw_img = draw_labeled_bboxes(np.copy(image), labels)

        return draw_img, heatmap, labels[1]

    else :
        return None, None, 0
```

The figures below demonstrate how the detection pipeline works based on 2 images recorded from the project video. The image sequence goes from right to left and up to down:

1.Image to process, 2. Identified vehicles (i.e. overlapping boxes, false positives), 3. Heat map without thresholding 4. Heat map after thresholding, 5. Label for combined drawing boxes, 6. Detection result

Image 1032: Example for false positive & overlapping drawing boxes

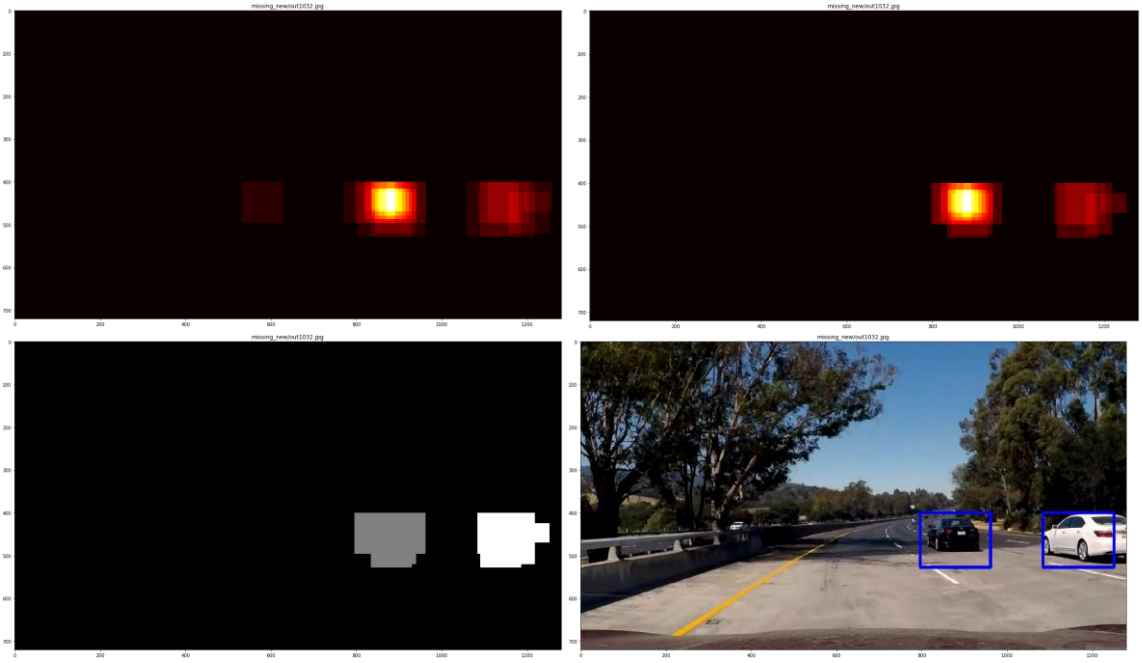
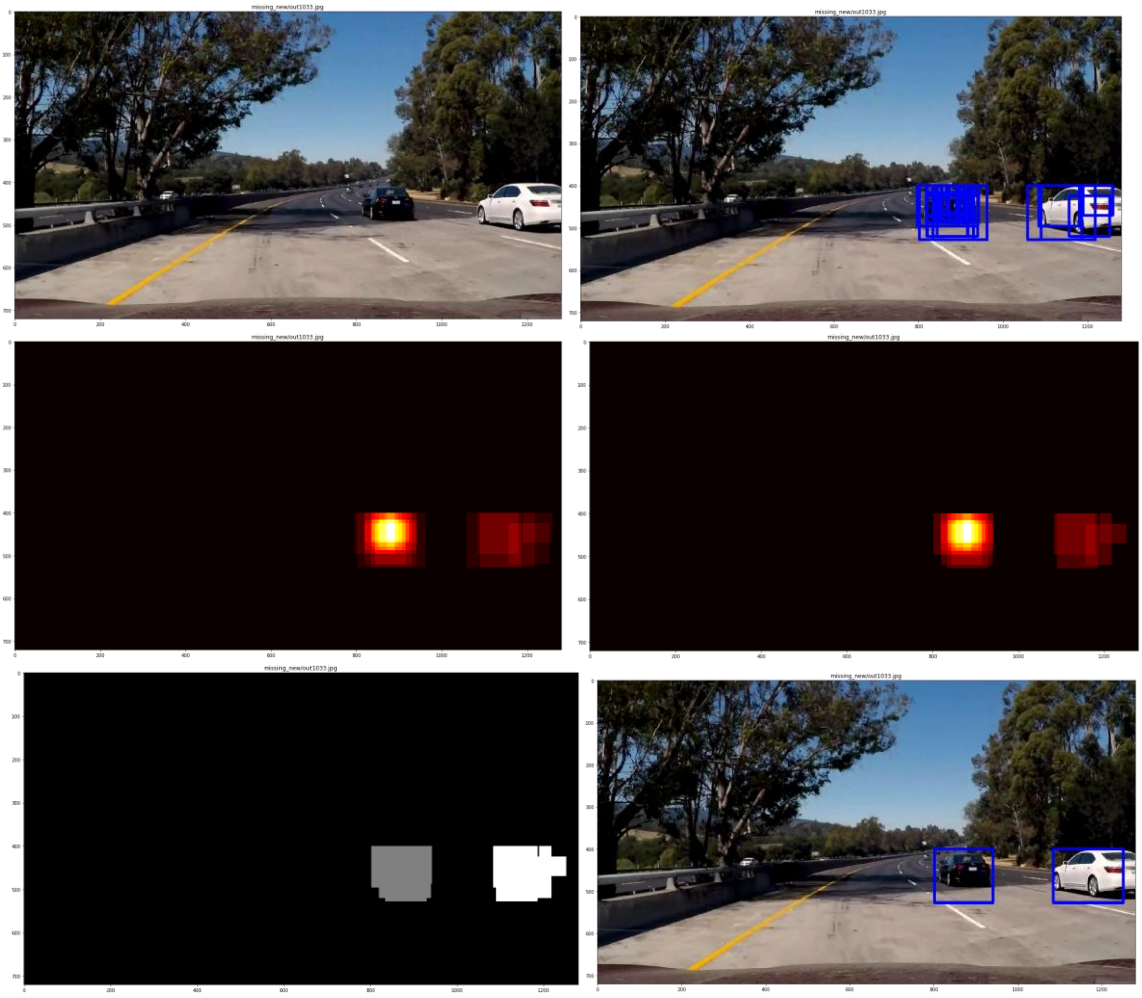


Image 1033: Example for overlapping drawing boxes



D) Discussion

I observed a lot of issues when selecting the number of search windows. I started with 2 search windows and end up with 4 in order to get a robust implementation. There have been the following issues:

- Vehicle detections with one drawing box (= 1 hit) have been removed by the heat map and thresholding process. Therefore, I tried to identify vehicles by multiple hits/drawing boxes.
- I also observed that one vehicle has been identified as two vehicles by having separate drawing boxes in the front and back after the labeling process. I worked around this by defining tighter the overlaps for different search windows.
- Defining ROI was sometimes quite challenging since It was influencing the detection process i.e. loosing detection information.

I think my search window implementation can be improved versus number of searches and therefore, in terms of overall execution performance.
My implementation is limited in terms of execution performance. Also there could be detection issues at the boundaries of the search grid and if cars are overlapping (see project video)