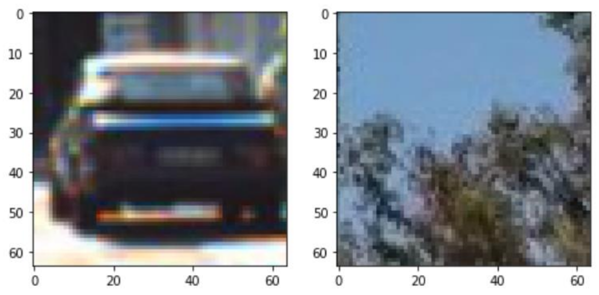
This is the write-up report for project5 Vehicle Detection and Tracking.

Histogram of Oriented Gradients (HOG)

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

Test image with manual box drawing



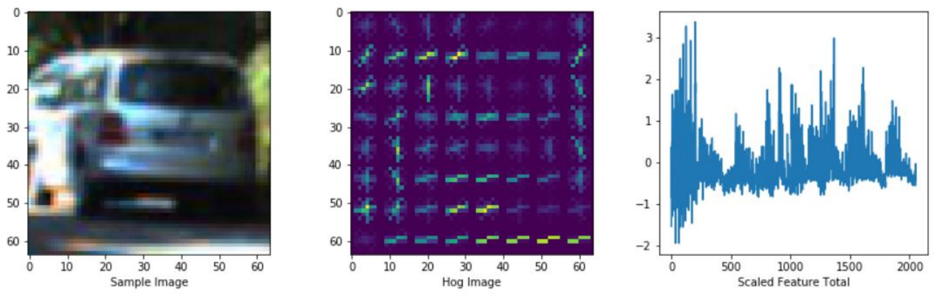
Sample image from car dataset and not-car dataset

Just like in every project, I started off from reading the test/sample images.

First two cells of ipynb notebook are used as import/reading images and some debugging places.

Then, I copied all functions from the lessons to use it as a coding ground, which is on the third cell.

HOG features are extracted from get_hog_features(), and this function is also used in other functions such as extract_features() and single_img_features() as well.

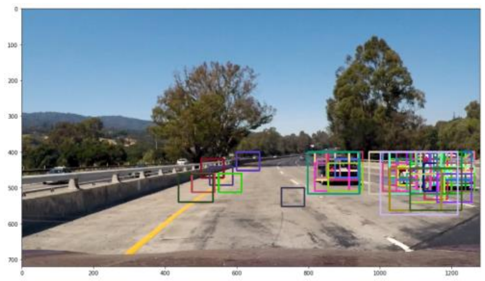


2. Explain how you settled on your final choice of HOG parameters.

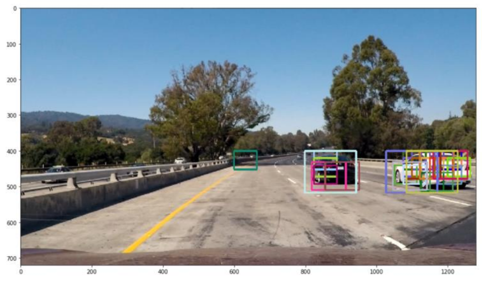
Although I attempted different color space images and feature plots, the difference between the car images and not car images wasn't very distinguishable nor was clear to evaluate numerically.

Therefore I stayed with 3 simple criteria.

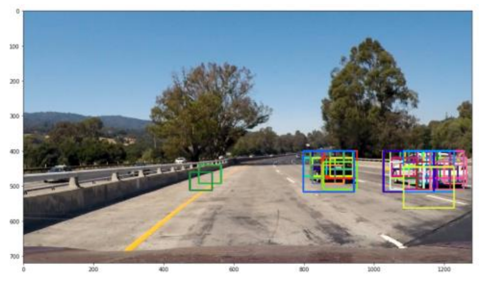
1. Better Test Accuracy of SVC
2. Yield less time consumption
3. Visually better performance on test video



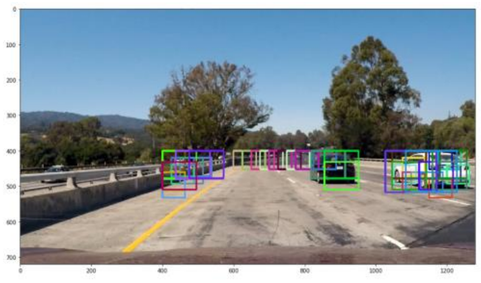
RGB



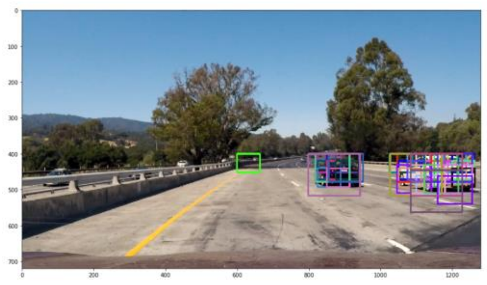
HSV



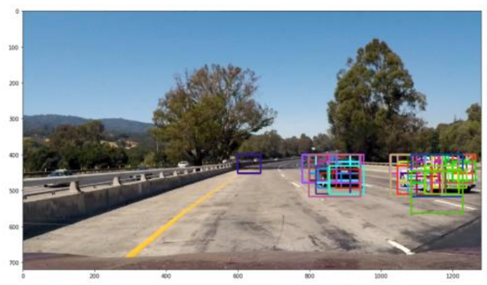
LUV



HLS



YUV

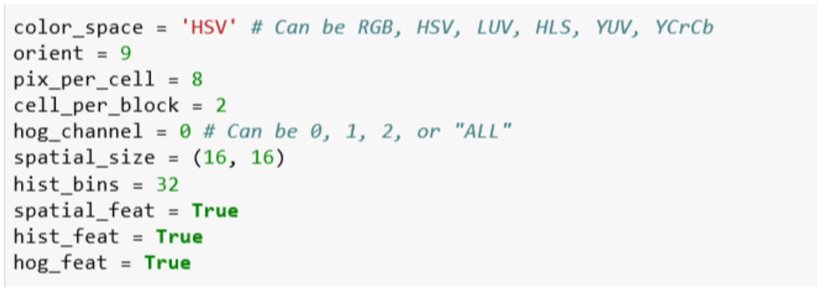


YCrCb

As you can see above comparison of different color spaces, HSV, YUV and YCrCb are the ones that yield relatively less false positives. After comparison on several video outputs, I decided to use HSV.

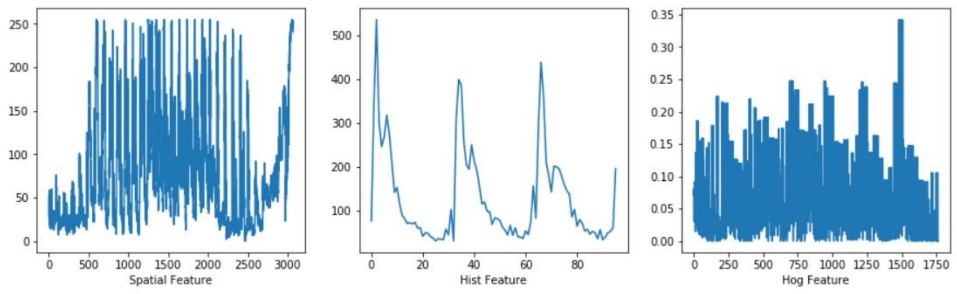
3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

In fourth cell where titled as "Train SVC" is the code cell where trained a classifier using datasets, using sklearn SVC, Support Vector Classification. For training, following parameter setting was used.

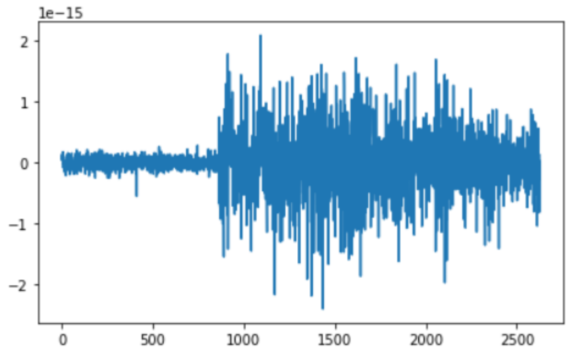


```
color_space = 'HSV' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = 0 # Can be 0, 1, 2, or "ALL"
spatial_size = (16, 16)
hist_bins = 32
spatial_feat = True
hist_feat = True
hog_feat = True
```

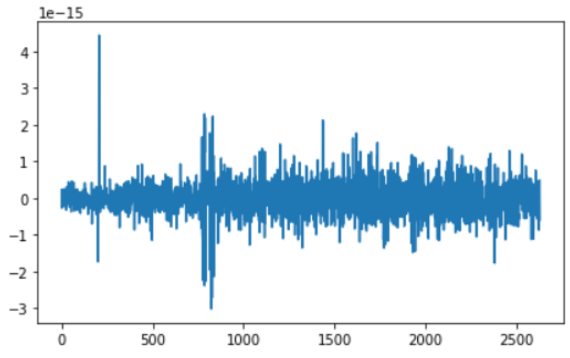
The feature set used for training contain all three types of features – bin spatial feature, color histogram feature in HSV and HOG feature. This feature set is scaled standardized to prevent one type of feature dictate the classification result.



These plots, however, didn't seem nothing but a series of noise to me, so I attempted an extra analysis on the all feature sets. Following is the simple average of standardized feature sets between all car dataset and all non-car dataset.



Feature Avg of cars



Feature Avg of non-cars

Following is the code used for the analysis

```python
import numpy as np
def ft_avg(features):
    vector = np.zeros_like(features[0])
    for ft in features:
        vector += np.array(ft)

    avgs = [x/len(features) for x in vector]
    return avgs
car_ft_avg = ft_avg(car_scaled)
notcar_ft_avg = ft_avg(notcar_scaled)
```

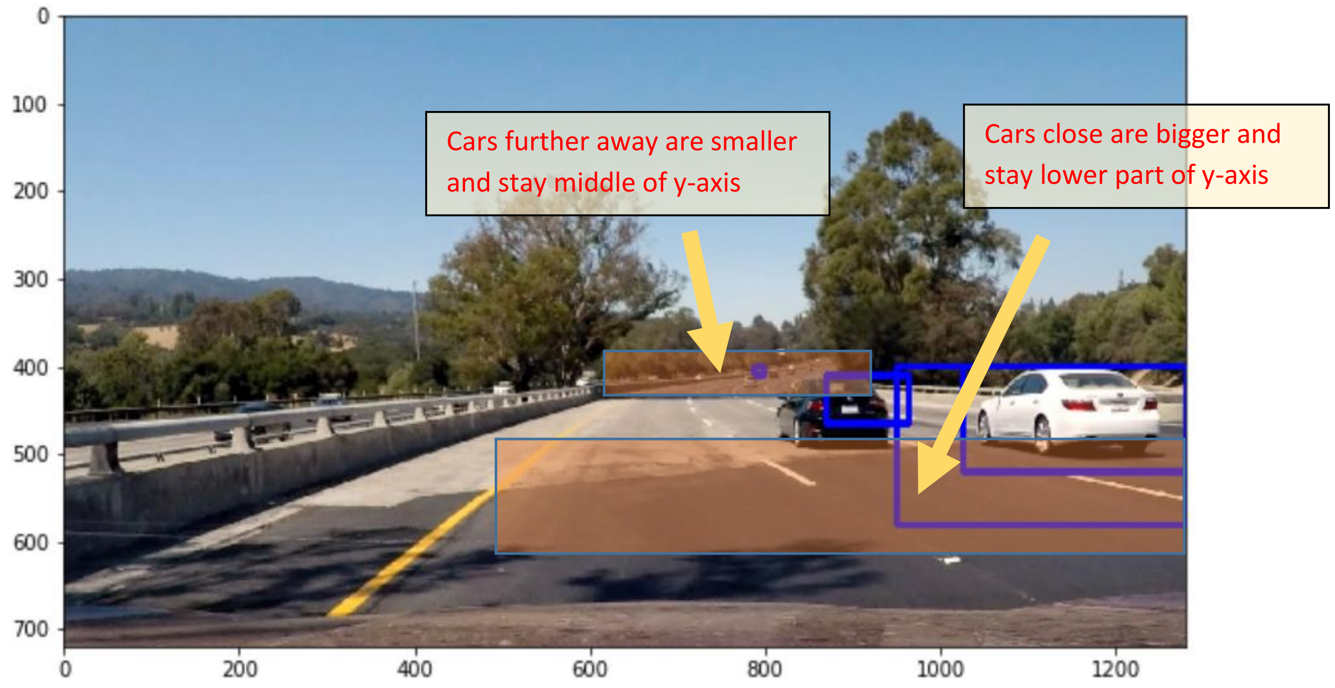
It's still very difficult to find the pattern out of this plot, but one noticeable thing is the variance of first part (likely the bin spatial feature) of cars is much smaller than non-cars.

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?

There are two simple things I did.

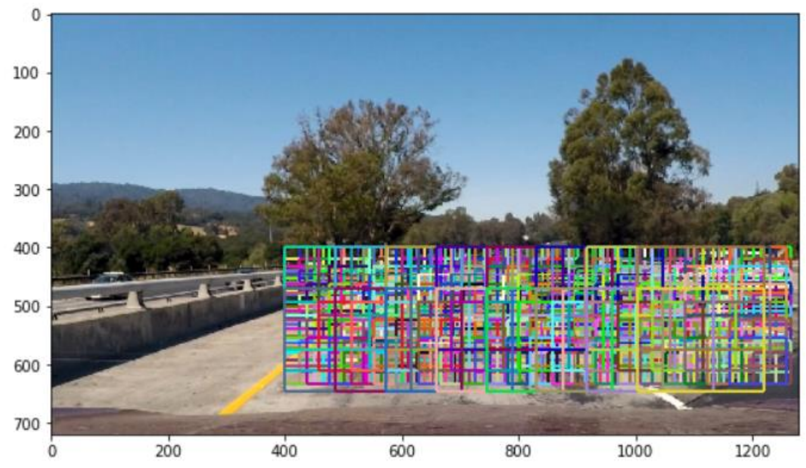
1. Limiting the search area for sliding window
2. Loop 5 different window size


Cars further away are smaller and stay middle of y-axis
Cars close are bigger and stay lower part of y-axis
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

More specifically, I limited search area to be x_start_stop=[400, 1280] and y_start_stop=[400, 650], so that sliding window doesn't have to search sky, other side of road (across the yellow lane at 400), and hood of the vehicle. Then modified slide_window() with a wrapper of for i in range(5): loop and modified window size that will increase window size by 50% on every iteration of the loop.

```
    xy_window = list(xy_window)
    xy_window[0] = np.int(1.5*np.asarray(xy_window[0]))
    xy_window[1] = np.int(1.5*np.asarray(xy_window[1]))
    xy_window = tuple(xy_window)
```

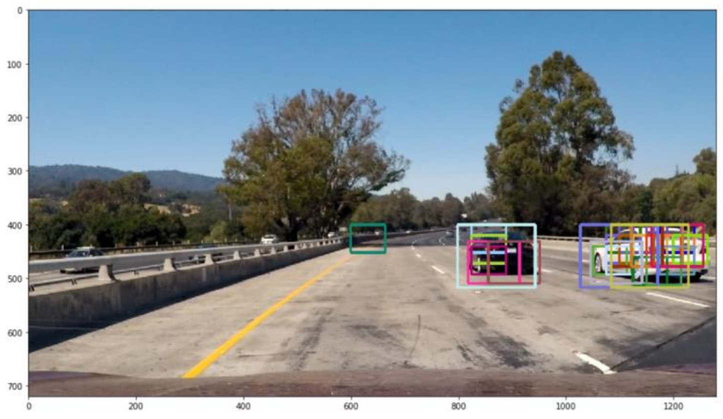
So, the sliding window search with all windows would be like following.



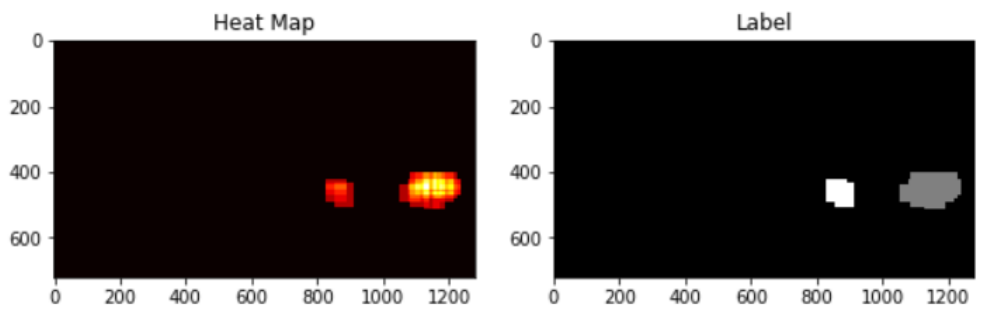
Of course, for loop with 5 iterations is time consuming, but it's a trade-off that also helps to detect false positives by thickening the heat map.

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?

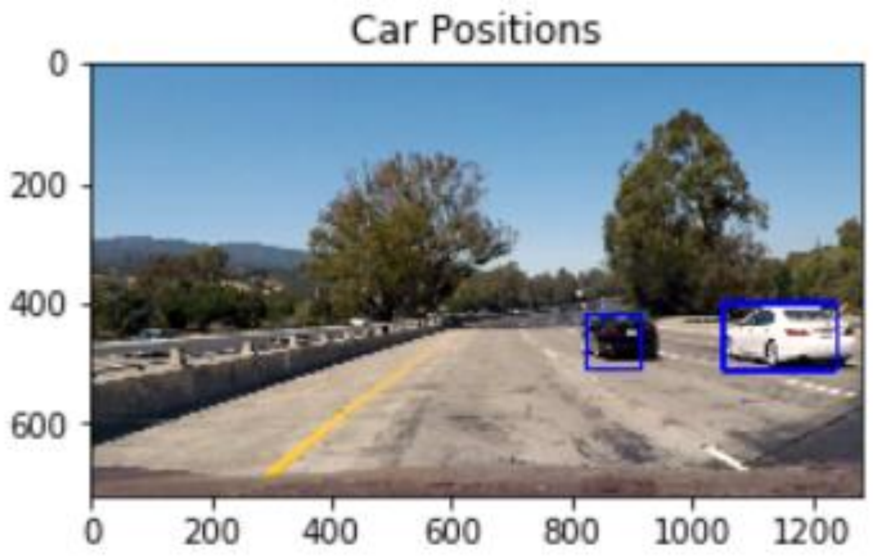
The difference between different color spaces are provided earlier. Using the HSV color space and feature extraction, I get following result with multiple boxes.



Then, heat map is used to find the spot where hot windows are overlapping.



The final result is like following.



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.)

Link to my final video (https://youtu.be/YENXHBwQG7s, using YOLO – see next for detail)

Link to my video (https://youtu.be/NFChMwSigIY using lesson methods)

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.

Although a series of above methods work on test images on up to certain point, boxes are very wiggly on videos. Classification with fixed size box over everywhere in each frame isn't great method fundamentally.

Thus, I decided to try something not taught in the lessons but popular object detection method I found on one of YouTube channel I'm subscribing – known as YOLO.

Basically, YOLO divide an image into bounding regions and detects an object using the relationship between neighboring regions.

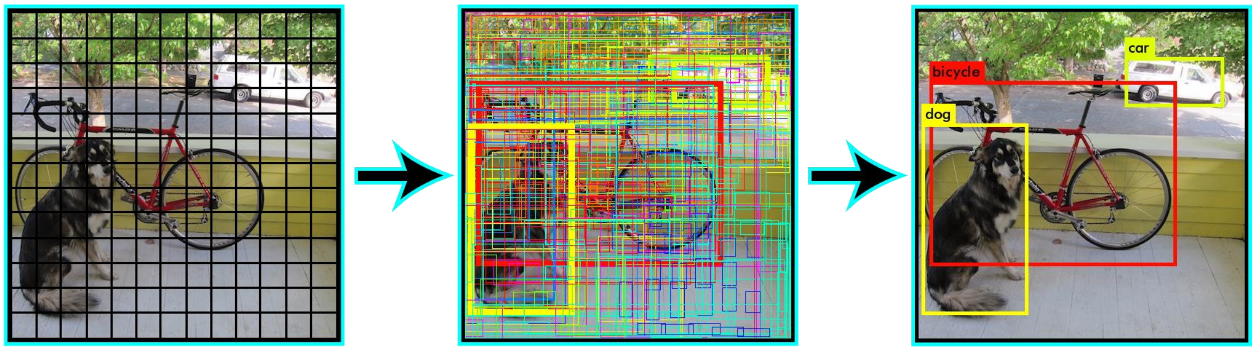


Image explaining YOLO from darknet webpage. See https://pjreddie.com/darknet/yolo/ for detail.

There are several YOLO versions and codes out there. One I used is from following github repo, simply because this repo can output video easily.

https://github.com/AlexeyAB/darknet

To use YOLO, run following in terminal

```
git clone https://github.com/AlexeyAB/darknet
cd darknet
make
```

Also, pre-trained weight could be obtained with

```
wget https://pjreddie.com/media/files/yolo.weights
```

The code is slightly different between different OS. One I used is Ubuntu, so I used following. (make sure the video is in darknet dir)

```
./darknet detector demo data/coco.data yolo.cfg yolo.weights project_video.mp4 -i 0 -out_filename out.mp4
```

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?

Although YOLO method works much better than our lesson approach with SVC, HOG and Heatmap, there was one moment where it read the vehicle's hood as a vehicle. This issue could be solved if I could adopt something similar to what I did to limit the search area for sliding window. I couldn't find where to modify in src code yet to implement this.

Of course, there is a room to improve existing lesson approach, but I'm out of idea for now. Perhaps mixing Project 4 code for advanced road lane with this could help, but that'd take a couple additional weeks of work.

Also, there are even several other options I'm aware of such as SSD or U-Net as discussed in a couple of forum discussions. These options are the ones I was looking into, but couldn't implement yet.

https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/
https://chatbotslife.com/small-u-net-for-vehicle-detection-9eec216f9fd6