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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Write-up / README that includes all the rubric points and how you addressed each one. You can submit your write-up as markdown or pdf. Here is a template write-up for this project you can use as a guide and a starting point.

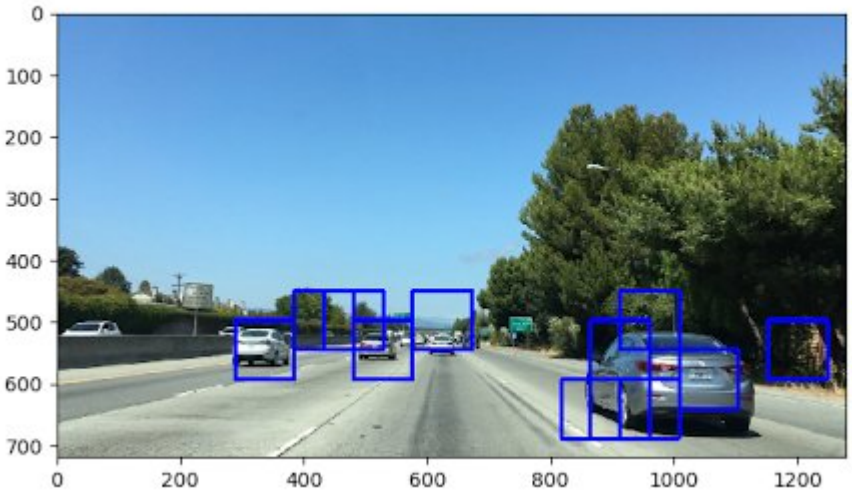
You're reading it!

Histogram of Oriented Gradients (HOG)

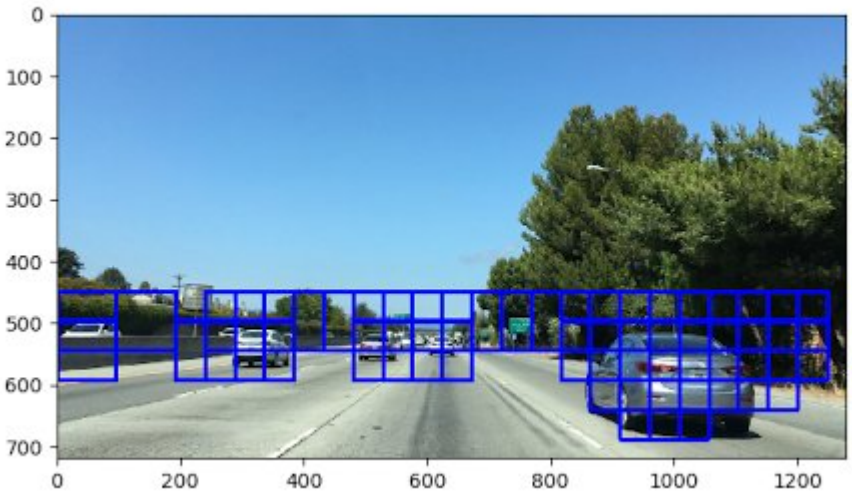
1. Explain how (and identify where in your code) you extracted HOG features from the training images.

The code for this step is contained in the `get_hog_features()` function in the P5.ipynb notebook. I'm extracting separate HOG gradients of the given image's YUV channels. I used all channels since they yielded a richer information, furthermore I was using this color space, because empirical tests showed its superiority in this scenario. Some example pictures of using different color spaces:

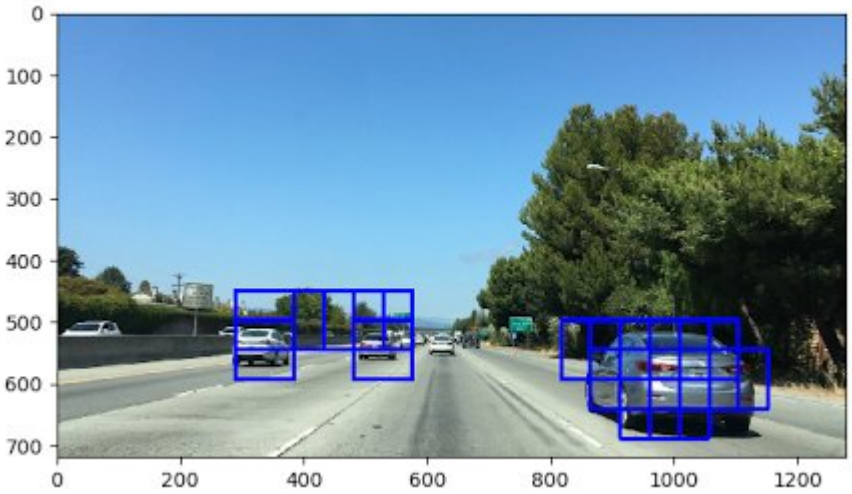
RGB:


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

HLS:

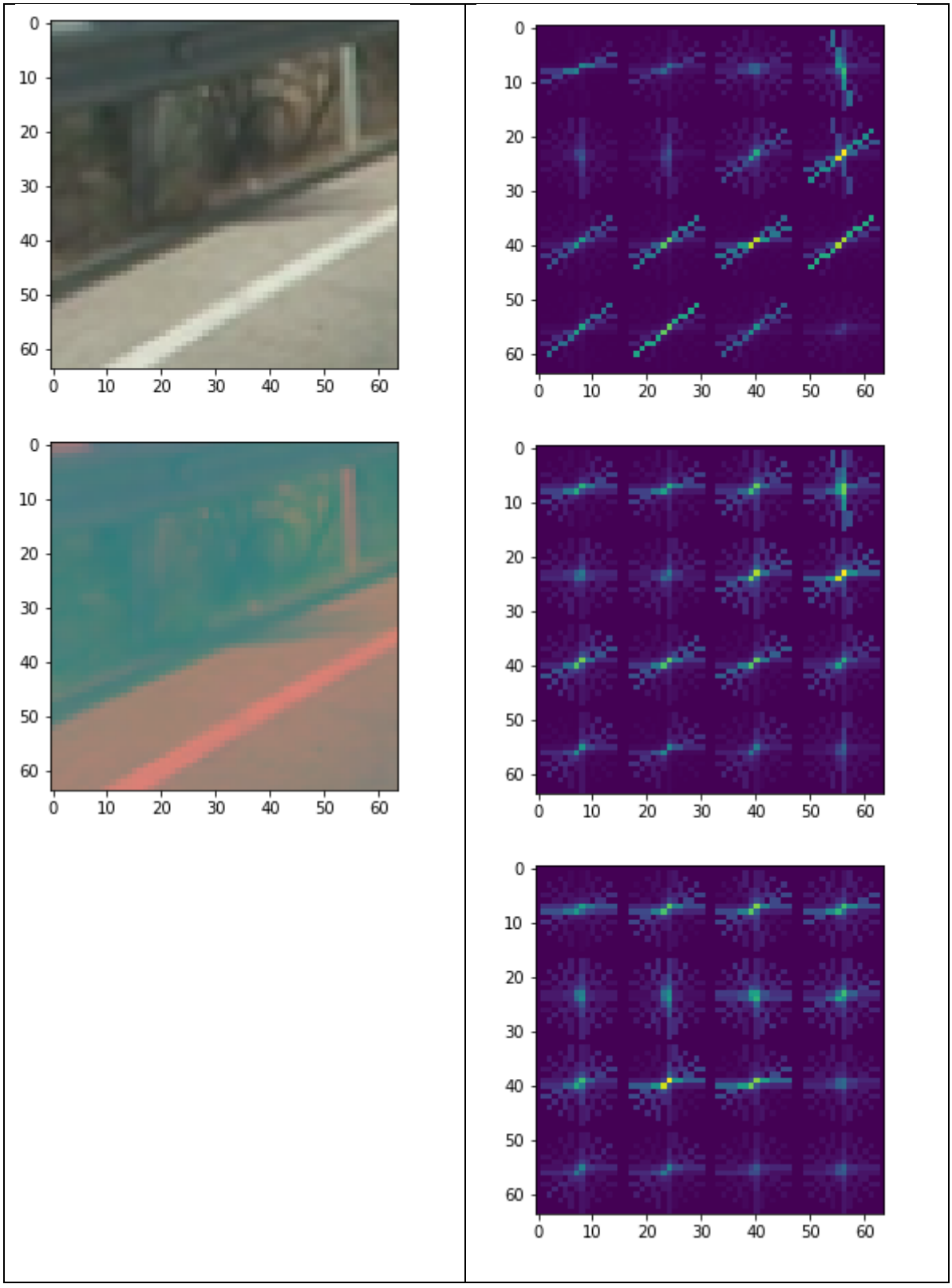

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

YUV:

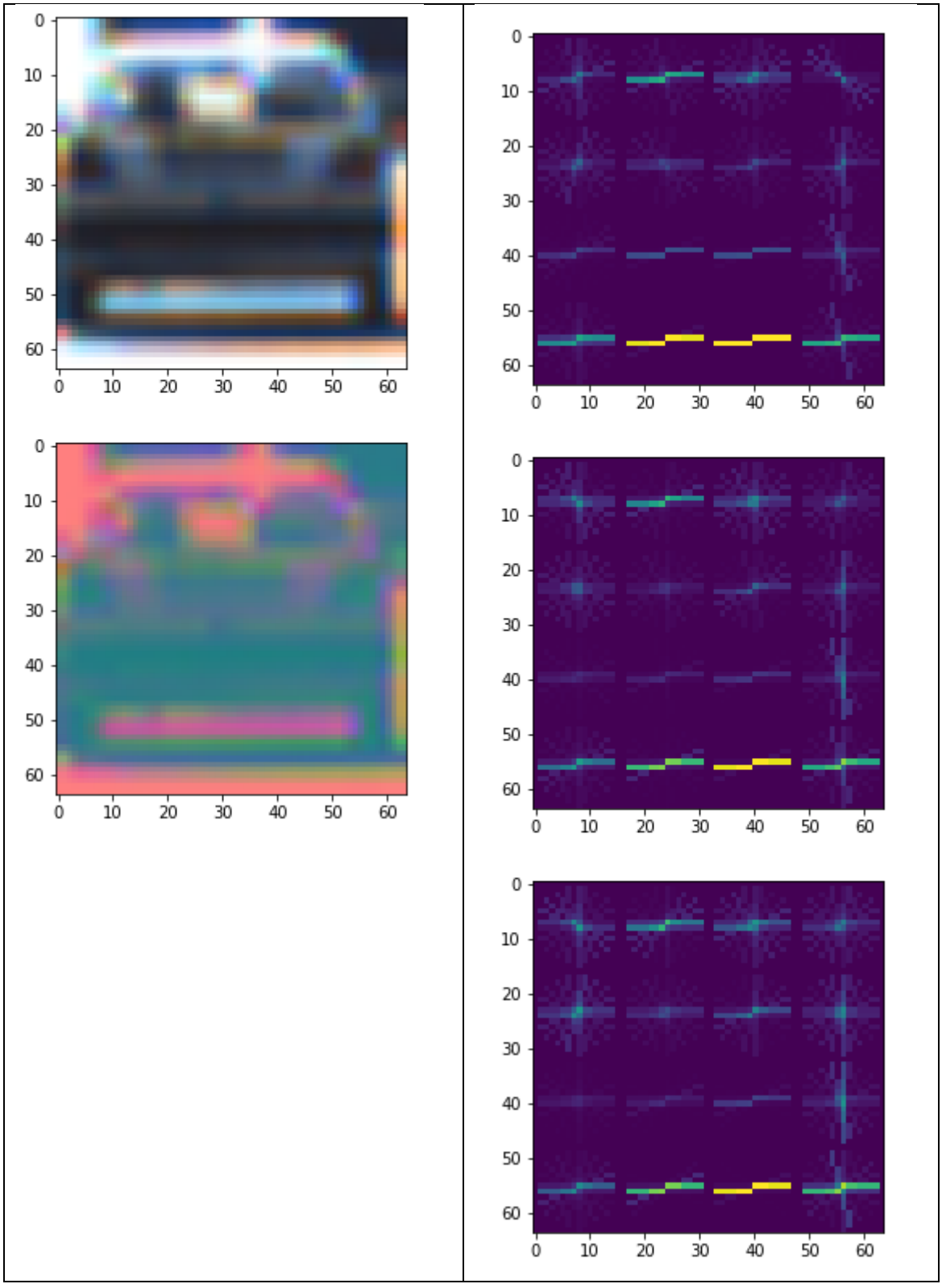

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

When experimenting with color spaces I also experimented with orientations, pixel/cell and cell/block values. I came to the conclusion that 11/16/2 was working best for me. 7 orientations were too little, and 13 put too much load on the cpu/mem. While exploring these possibilities I also checked the resulting HOG feature vector's histogram whether it is distinctive enough in order to decide between cars and non-cars.

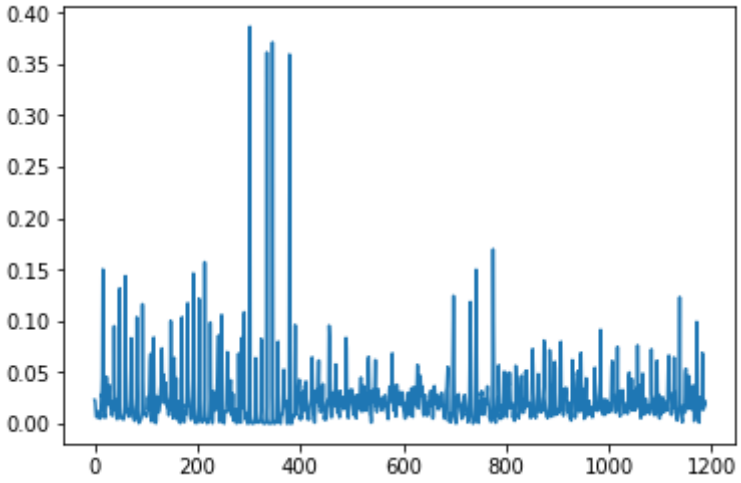
RGB image and YUV image of non-car
…and its YUV component's corresponding HOG gradients:



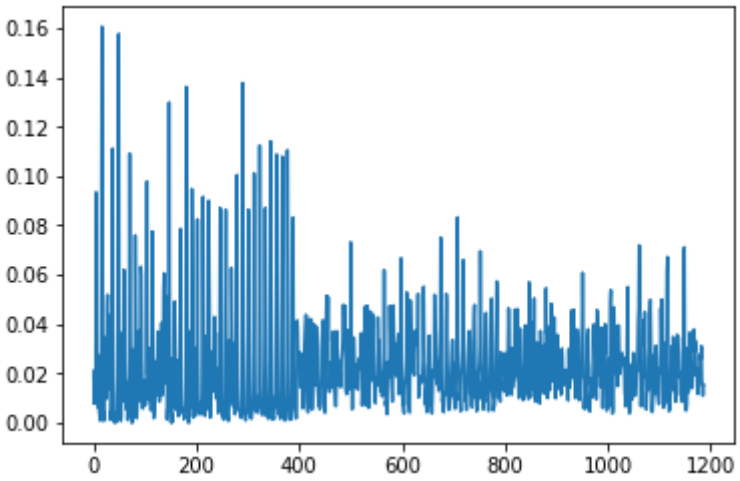
The same for a car image:



Sample feature vector histogram for a car



and for a non-car:



2. Explain how you settled on your final choice of HOG parameters.
Actually, as mentioned before, I was experimenting with them. I choose the that parameter set, that yielded the best accuracy (`from sklearn.metrics import accuracy_score`).
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 is done in section "Training/loading SVM" in the python notebook. I chose `sklearn.svm.SVC` using rbf with C=10. The reason for this was that I tested several options using `GridSearchCV` and this performed best.

```
gamma=auto
{'C': 10, 'kernel': 'rbf'}                          |   99.97% / 96.34%
{'C': 7, 'kernel': 'rbf'}                           |   99.94% / 96.28%
{'C': 13, 'kernel': 'rbf'}                          |   100.00% / 96.28%
{'C': 4, 'kernel': 'rbf'}                           |   99.76% / 96.22%
{'C': 16, 'kernel': 'rbf'}                          |   100.00% / 96.22%
{'C': 1, 'kernel': 'rbf'}                           |   99.17% / 95.93%
```

```
{'C': 1, 'kernel': 'linear'}                    |   100.00% / 95.69%
{'C': 4, 'kernel': 'linear'}                    |   100.00% / 95.69%
{'C': 7, 'kernel': 'linear'}                    |   100.00% / 95.69%
{'C': 10, 'kernel': 'linear'}                   |   100.00% / 95.69%
{'C': 13, 'kernel': 'linear'}                   |   100.00% / 95.69%
{'C': 16, 'kernel': 'linear'}                   |   100.00% / 95.69%
{'C': 16, 'kernel': 'poly'}                     |   99.50% / 95.16%
{'C': 13, 'kernel': 'poly'}                     |   99.38% / 94.92%
{'C': 10, 'kernel': 'poly'}                     |   99.17% / 94.86%
{'C': 7, 'kernel': 'poly'}                      |   98.73% / 94.75%
{'C': 1, 'kernel': 'poly'}                      |   97.61% / 94.16%
{'C': 4, 'kernel': 'poly'}                      |   98.08% / 93.98%
{'C': 1, 'kernel': 'sigmoid'}                   |   91.68% / 90.50%
{'C': 4, 'kernel': 'sigmoid'}                   |   87.87% / 86.19%
{'C': 7, 'kernel': 'sigmoid'}                   |   86.69% / 84.71%
```

As of the HOG gradients: I did not use spatial transformation neither color histogram. They did not seem to add much value to the SVM, and just made the handling bulkier and slowed down processing the images. I tried generating larger feature vectors, but it did not help much.

For feeding data to the classifier I divided the feature and label sets into two: test (20%) and training (80%) data. I using SkLearn's `train_test_split()` function with randomization enabled.

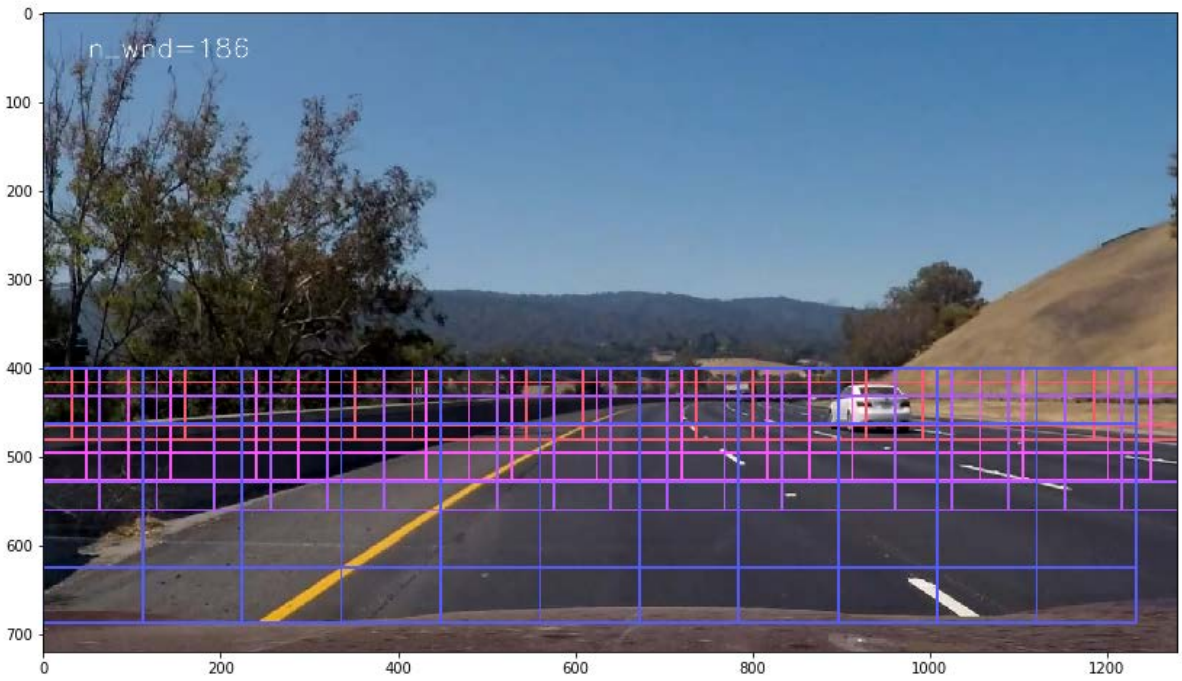
It is worth mentioning here that this might not be the best solution, since many sample images are taken from a video and this results in the test set being "too good" – there is a high chance that very alike images are present in the test set somewhat biasing the accuracy results.

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?

I covered four overlapping regions with different sizes of windows. They can be seen on the picture below. I thought it was important to use bigger windows for the nearer objects meanwhile smaller ones for the further ones. Altogether I scan every image using 186 windows. Overlapping sideways is always 50%.

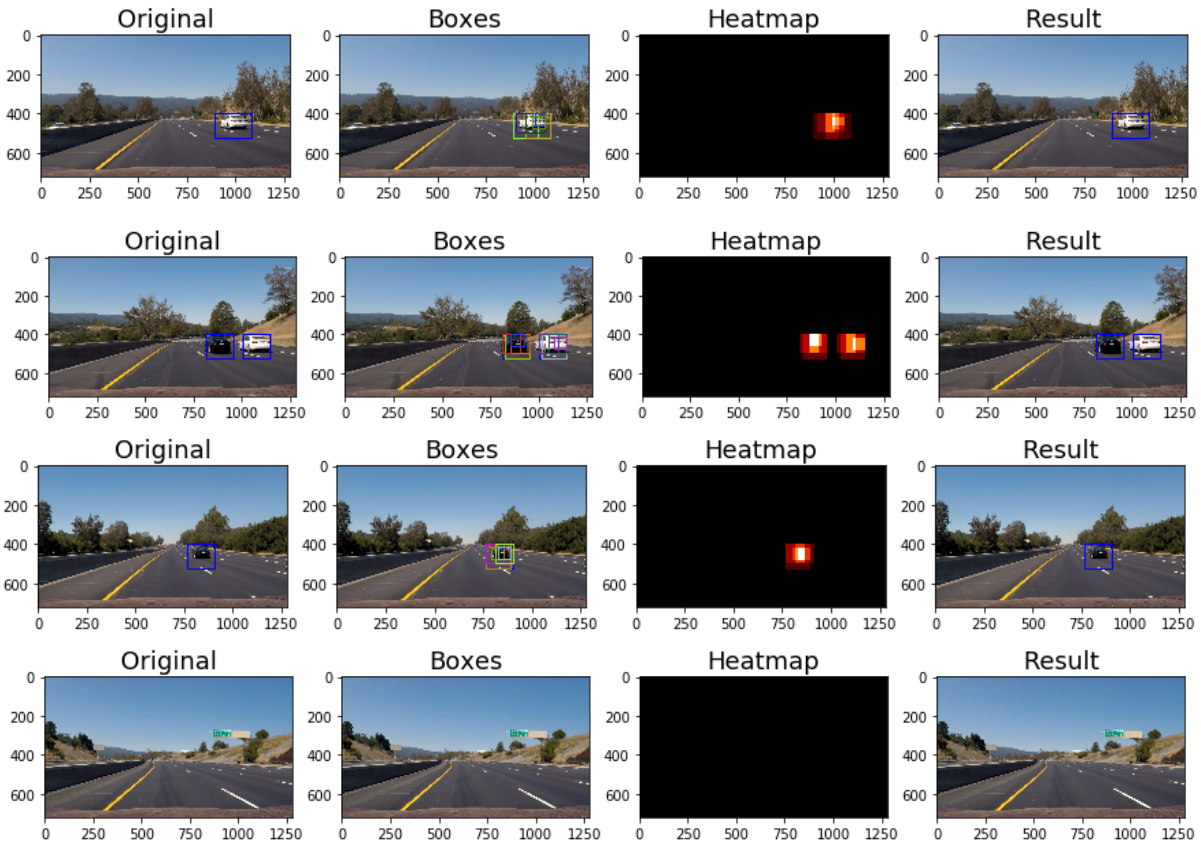
The function is implemented in "Creating window templates" cell. Function name is `slide_window()`.



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 classifier uses limited feature vectors in order to improve speed and reduce memory use. Furthermore, HOG-subsampling is an optimization itself: images can to be processed as a whole, no need for submitting the individual patches one by one.

Here are four snapshots of the pipeline showing the original picture, the matched windows, the heat-map, and the corresponding labeled bounding boxes:



For creating the final bounding box first I cumulated all the matching search windows into a heat-map, then I applied a threshold on the result in order to drop out weak candidates, then I used `scipy.ndimage.measurements.label()` to create a final bounding box for the car (with blue).

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's a link to my video result

The final video can be downloaded from my GitHub repo.

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.

Filtering consists of two parts: the already mentioned heatmap thresholding, and a python class, which "memorizes" previous boxes. I implemented this by introducing a `collections. deque` and store newly established bounding-boxes on a FIFO basis. In order to follow the car smoothly and account for any missing boxes in the meantime last registered boxes contribute to the heatmap. Then this aggregate heatmap can be thresholded with a greater value so that nothing gets "burnt-out".

Implementation can be seen in P5.ipynb where `class BoxMover` is defined.

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?

One of the most challenging problems were obtaining the optimal parameter set for SVM. I still think it could be improved, but the time is a hard constraint :)

Furthermore, the implementation could still be improved in terms of processing speed, but again, there was not time for making it real-time.

According to my assumption the pipeline is likely to fail by hilly (up/down) conditions since the whole image is not processed (namely the upper part).

What I could think of that the underlying learning "database" could be filtered and extended so that matching becomes more robust. E.g. trucks and other type of objects won't be recognized.