Project 5 – Vehicle Detection – Writeup

David Chrzanowski

The goals are:

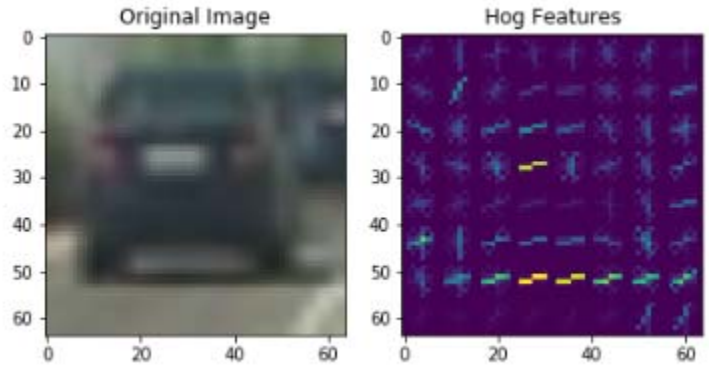
- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a Linear SVM Classifier.
- Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
- Normalize your 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 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

Histogram of Oriented Gradients (HOG):

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

An example of HOG feature extraction and visualization can be found in the first section of the "*train_classifier.ipynb*" jupyter notebook, and is shown below. Here, and anywhere HOG features are extracted, the "*get_hog_features*" function is called. That function can be found in the "*features_scan.py*" in line 86, and is built around *skimage.feature*'s "hog" function.



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

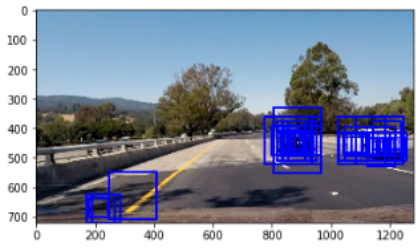
I ended up using the following HOG parameters:

```
color_space = 'HLS'
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = "ALL"
```

I tried each color space by re-training the classifier in that space and then applying it to test images (re-colored into that space). HLS and YCbCr spaces performed the best – with slightly more false positives identified with the YCbCr space – so I went with HLS. The remaining parameters are unchanged from the examples in the lesson. I figured that they worked well enough there, they were probably good enough for the project. Re-training the classifier (and recoding the pipeline) using just the first ([0]) channel in the YCbCr color space could represent a time-saving substitution.

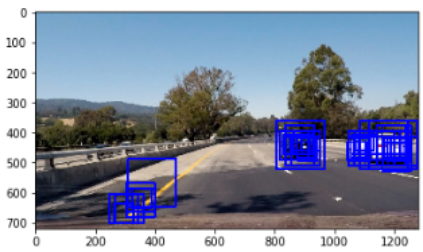
USING HLS:

```
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 6156
29.65 Seconds to train SVC...
Test Accuracy of SVC =  0.9921

<matplotlib.image.AxesImage at 0x1c6e20f0>
```



USING YCbCr:

```
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 6156
21.79 Seconds to train SVC...
Test Accuracy of SVC =  0.9921

<matplotlib.image.AxesImage at 0x46b24f98>
```



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

The linear SVC is trained in second section of the "*train_classifier.ipynb*" jupyter notebook. The glob API is used to import the 2 directories of "not car" images and the 5 directories of "car" images. Features are extracted from the images using the *extract_features* fuction in *features_scan.py* (line 124). This function includes extraction of HOG features (parameters above), as well as spatially-binned color and color-histogram data as features. The lists of features are concatenated (in the order spatial – histogram – HOG), and a *StandardScaler()* object is used to scale the feature vector. A label list is created, and the data is

randomized and divided using the *train_test_split.* This data was then used to train and test the linear Support Vector Classifier.

To prevent having to re-perform these operations the SVC object and StandardScaler object are pickled for later use.

Sliding Window Search:

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?

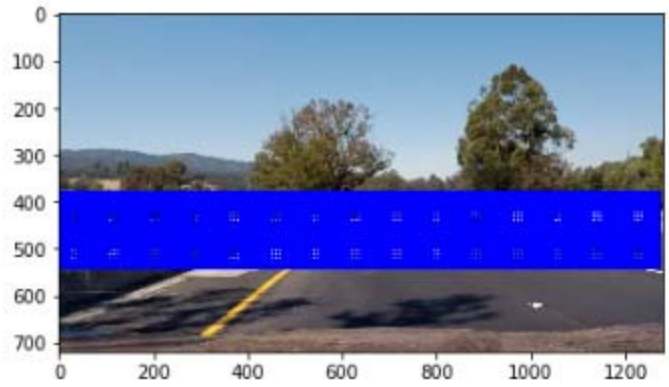
The code that performs the "sliding window" search can be found in the *find_cars* function in *features_scan.py* (line 7). Time is saved in this function by calculating the hog features for the entire search area at once, and then traversing a map of calculated values with the search windows. The windows traverse the full width of the frame, and a vertical span defined by the *ystart* and *ystop* variables fed to the function. The size of the window is defined by the *scale* variable, which is essentially multiplied by 64 to get the number of pixels along one side of the square box. The amount of overlap is determined by the *cells_per_step* variable.

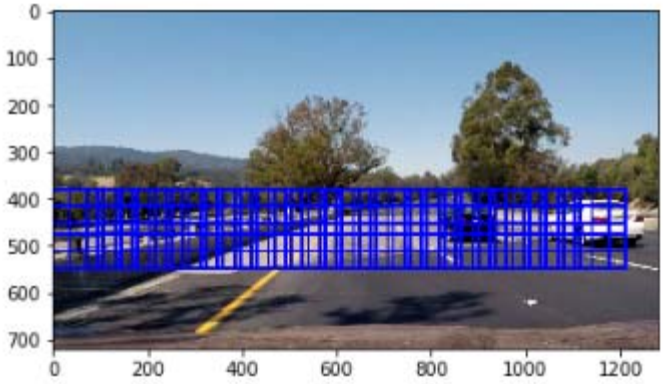
In the pipeline implementation (*processing_pipeline.ipynb)*, the *find_cars* function is called 3 times, each with a different search area, window size, and overlap – see below.

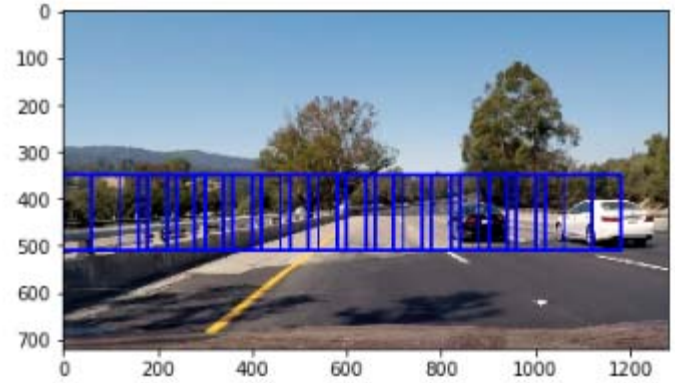
First Pass	Second Pass	Third Pass
ystart = 380 ystop = 550 scale = 1.0 (64x64) cells_per_step = 1 (~88% overlap)	ystart = 380 ystop = 600 scale = 1.5 (96x96) cells_per_step = 3 (~64% overlap)	ystart = 380 ystop = 600 scale = 2.5 (160x160) cells_per_step = 3 (~64% overlap)

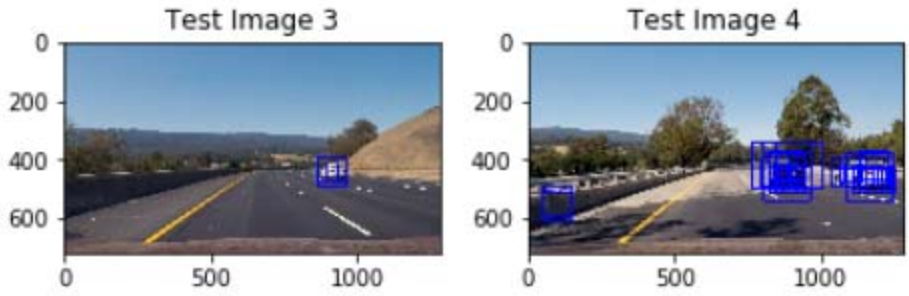
Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

Below is an example of all the windows searched by the pipeline by pass (see table above), as well as detections for 2 example images.


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



Through exhaustive trial and error, it became clear that when using a blanket threshold across the full image (discussed later in “filter” section) more overlap was required for the smaller windows. This is because as cars move farther away and get smaller, it is easy for their detections to get thrown out, as there is less of them to find and they may get a small number of detections.

Video Implementation:

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.)

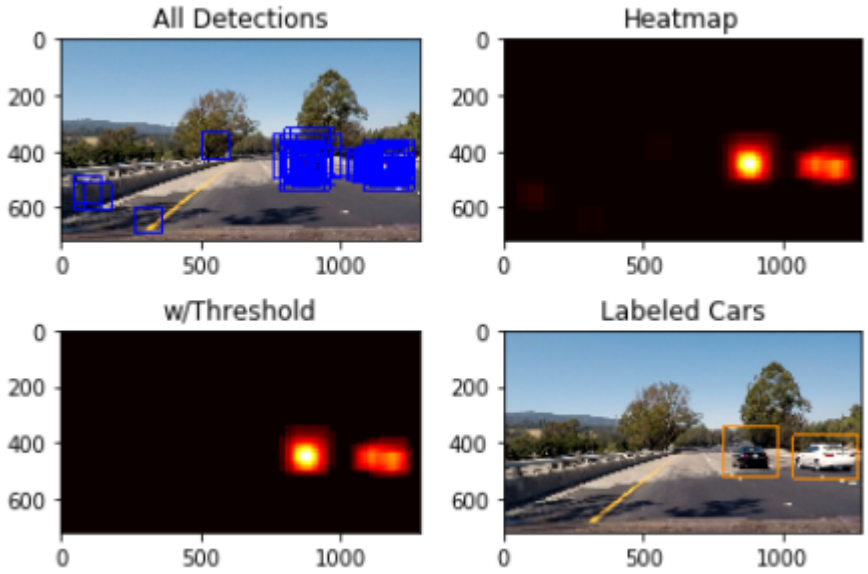
See the *project_video_output.mp4* file in this directory.

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 filtering process is achieved by several calls of the *add_heat* function (*id_label.py*), the *apply_threshold* (*id_label.py*) function, and *cipy.ndimage.measurements’ “label”* function. *draw_labeled_bboxes* illustrates found cars. This can be found in the second section of *processing_pipeline.ipynb.*

The implementation creates a heatmap, where on a blank 1-channel copy of the image every pixel within a positively-identified window gets a value of 1 added to it. Naturally there are many overlapping positive identifications, leading to some cells with values of 5-10+ if they were identified in that many windows. For a single frame – any cells with values of 2 or fewer are thrown out to avoid false positives. The values are then summed across a sliding record of the last 10 frames, and another threshold (of 7) is then applied to smooth the ultimate bounding box and further avoid false positives. The *label* function and *draw_labeled_bboxes* is applied to the 10-frame sum, and *draw_labeled_bboxes*

(id_label.py) uses the labeled boxes to draw the box(es) that appear(s) in the video.



Discussion:

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?

False positives proved a significant challenge in my implementation. Some of them were coming from the parts of the video in which oncoming traffic becomes visible. The only way I could find to eliminate them was by applying a zero-out mask in the *apply_threshold* function to the left side of the video (regardless of heatmap value). I justify this by saying that with knowledge of the lane curvature (project 4!) and lane position, one could dynamically adjust the search area for surrounding traffic. As implemented, my pipeline would fail given a lane change, as well as any significant change in the grade of the road (as my vertical search area is narrow).

My pipeline could be made more robust by:

- Improving the training of my SVC (which I believe is a little overfit)
- Making the search area a function of the car's position and road shape/grade, rather than statically hard-coded as I have implemented here

- Further optimizing the window search (as currently the time to render the video is significantly short of real-time detection)