Vehicle Detection – Project 5
Sean Robinson

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.

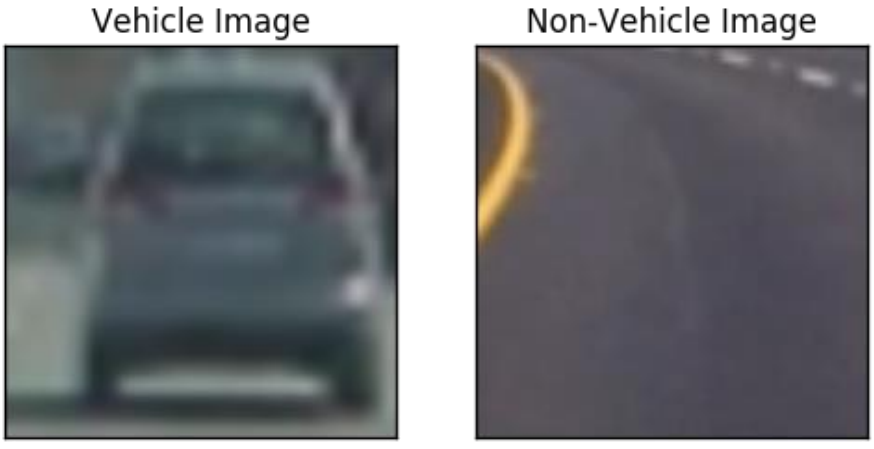
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Histogram of Oriented Gradients (HOG)

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

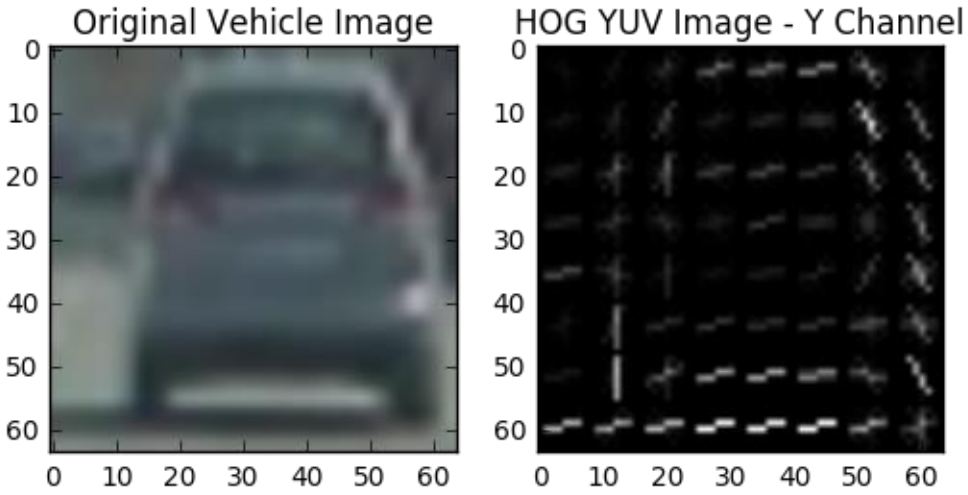
The code for this step is contained in the second code cell of the IPython notebook (vehicle_training_model.ipynb) including lines 14-126 from vehicle_training.py.

I started by reading in all the vehicle and non-vehicle images using glob. Here is an example of one of each of the vehicle and non-vehicle classes:



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

I experimented with RGB, HSV, HSL, and YUV color spaces as well as numerous HOG parameters. The final combination that I chose resulted in the best accuracy (98.56%) on my test set. Here is an example of the original vehicle image along with the corresponding HOG feature image in YUV color space (Y channel only).



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).

A LinearSVC was used as the classifier. The SVC training process can be seen in code cells 3, 4, and 5 of vehicle_training_model.ipynb. The features are extracted and concatenated using defined functions in vehicle_training.py. The images must first be scaled to 0-255 before being passed into the feature extractor, since the training data consists of PNG files.

The features include HOG features, spatial features, and color histograms. The classifier is set up as a pipeline that includes a scaler as seen in code cell 5 in the Ipython notebook.

The model was then stored in the *models* folder as "clf_9_10_3_0.pkl" and obtained a test accuracy of 98.56%. The training/test data was randomized and training set was comprised of 80% of the total data with the remaining 20% being used as the test data set.

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?

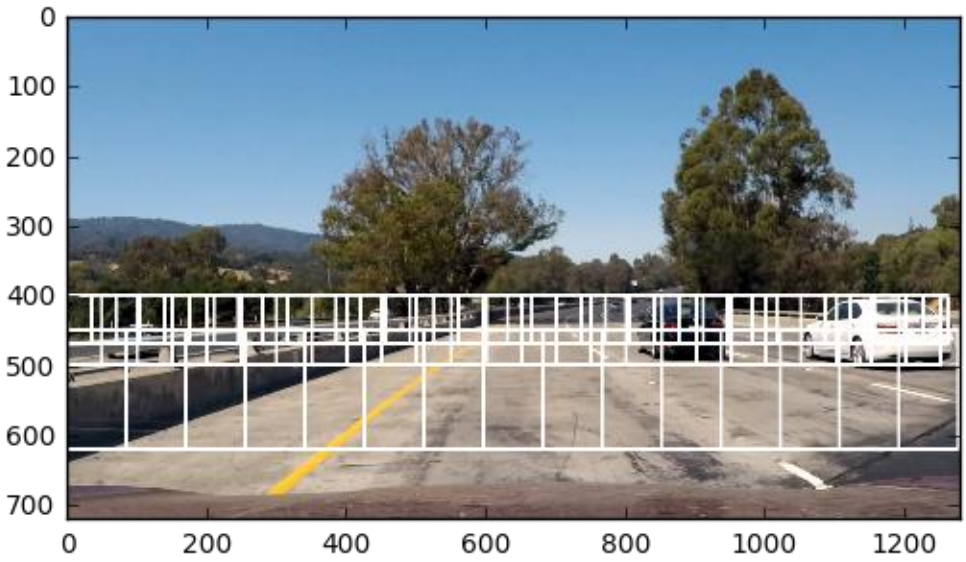
I used the function “window_search” in lines 134-140 of vehicle_detection.py to generate the list of windows to search. The input to the function specifies both the x and y window size, the window overlap percentage in x/y, and the y-axis range of the input image that the window is to be applied to. I tried various scales and eventually selected the following window parameters based on the video processing results:

Window 1: 70 pixels x 70 pixels, y-range= 400-500

Window 2: 100 pixels x 100 pixels, y-range= 400-500

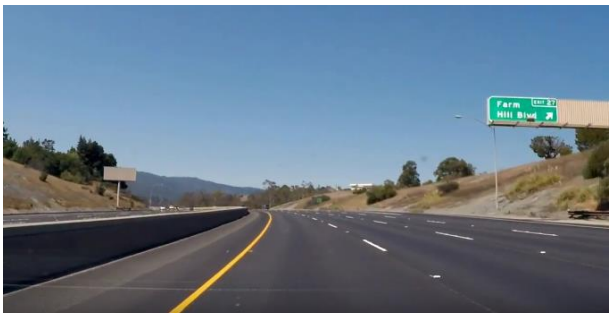
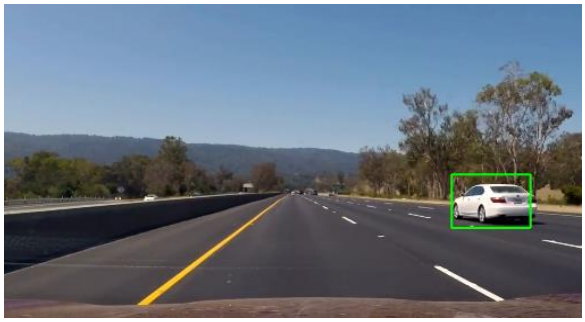
Window 3: 170 pixels x 170 pixels, y-range= 450-620

Here is an example image with the three sliding window scales overlaid on the image:



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?

I searched for vehicles using the YUV (Y-channel) color space with the three window scales. These windows were used to create a heatmap which I then applied a threshold to remove false positives. This combination seemed to provide decent results.
Here are some example images:

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: https://youtu.be/8H_dbkQYKco

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.

Heatmap and Overlapping

I created a heatmap by starting with all the windows marked as a vehicle by the classifier. I then reduced the false positives by using the classifier's decision function. By setting a threshold on the decision function in “search_windows” (lines 125 and 126 in vehicle_detection.py), many false positives were excluded from the video.

The heatmap was then summed over 35 frames and any pixels below a threshold of 10 were ignored. This helped to remove false positives that didn't show up consistently from frame to frame.

Heatmap Tracking

I also used a vehicle tracking feature to try to better distinguish between true/false positives from the heatmap detection. This is performed by the VehicleTracker class in vehicle_tracking.py. After the clusters are extracted from the heatmap, the dimensions are passed to a Kalman Filter in kalman_filters.py.

The heatmap tracking was implemented to identify possible candidates for tracking. These are implemented in lines 178-208 of vehicle_tracking.py.

Discussion

This pipeline works reasonably well for the project video and it didn't detect any false positives. One possible way to improve this pipeline would be to completely mask off areas where we are sure there are not going to be any vehicles detected, such as the sky.

The pipeline also seemed to have trouble when vehicles would overlap one another. The pipeline would think two separate vehicles were one large vehicle until they were spatially separated enough. This could possibly be improved by a better vehicle tracking implementation that would store the number of vehicles detected at any moment. Knowing that the number of vehicles detected will remain the same over at least a few seconds could possibly improve the detection of overlapping vehicles.

The current pipeline runs at a speed of about 8 frames per second on my laptop. Using a convolutional neural network to process the images might be much faster than a SVM classifier like the SVCLinear classifier used here. By making the pipeline faster this would enable close to real-time vehicle detection and possibly allow more refined detection techniques to be implemented.