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 (https://review.udacity.com/#!/rubrics/513/view) Points

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

Writeup / README

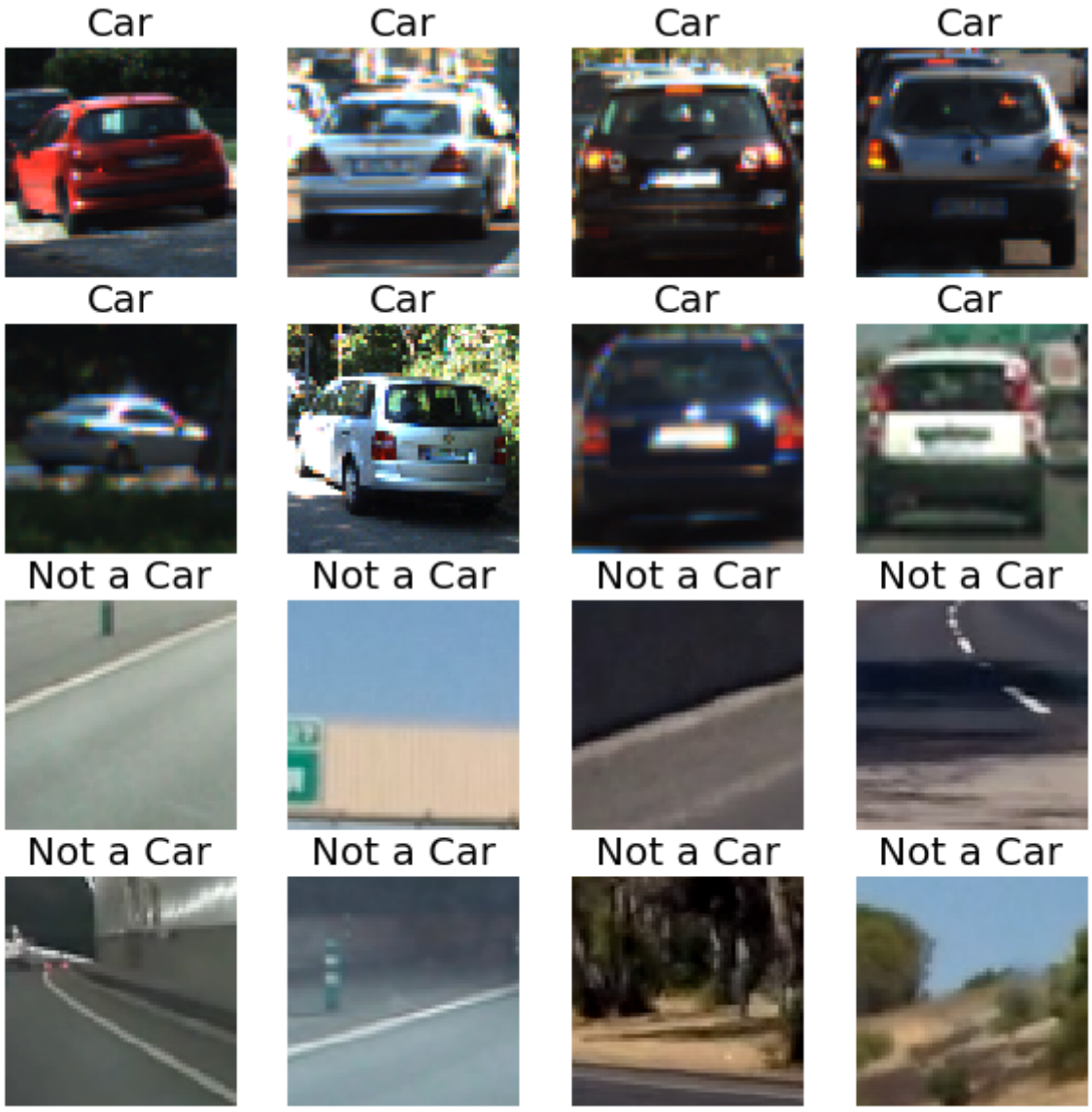
1. This is the current file

Histogram of Oriented Gradients (HOG) Pipeline

1. Extraction of HOG features

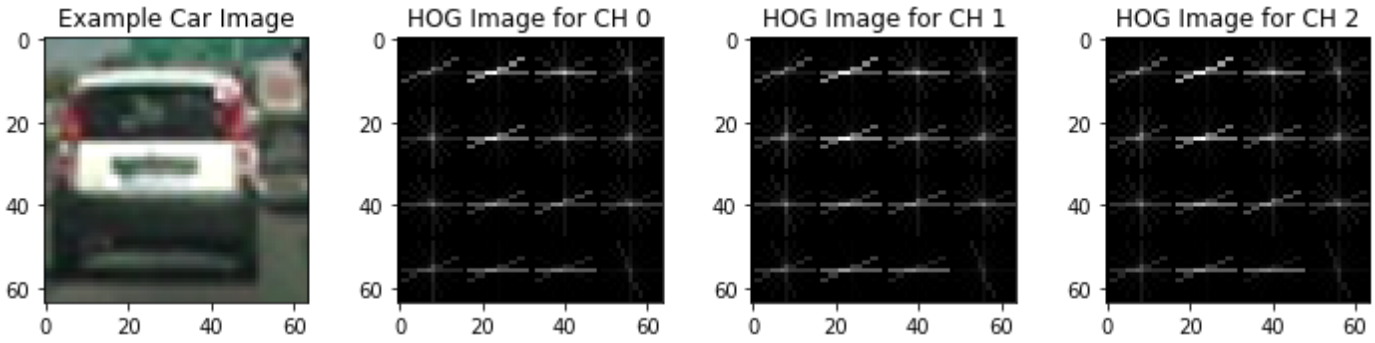
The extraction HOG features from the training images are in the get_hog_features() function in the P5.ipynb notebook. The extract_features() function is used to call get_hog_features().

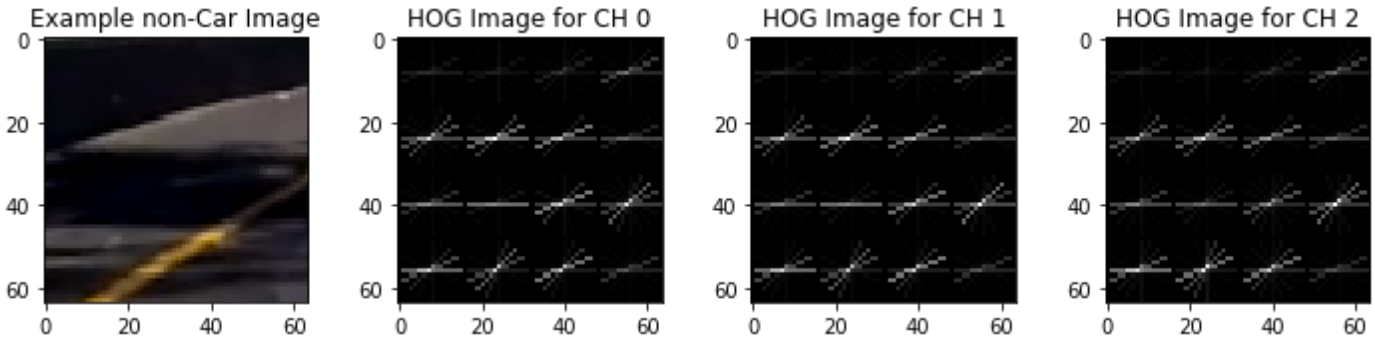
The training data is separated into car and non-car examples. I started by reading in all the `vehicle` and `non-vehicle` images. The vehicle examples were derived from the GTI vehicle image database (http://www.gti.ssr.upm.es/data/Vehicle_database.html) and the KITTI vision benchmark suite (http://www.cvlibs.net/datasets/kitti/). These were packaged and provided by Udacity. There were a total of 8792 vehicle examples and 8968 non-vehicle examples in the dataset, but only 8500 examples of each are actually used for a even test set. Each example is a color image with a resolution of 64x64 pixels. Here are examples of the `vehicle` and `non-vehicle` classes:



I looked into various color spaces and found that YCrCb was good for the extraction of the HOG parameters. This was done in the get_hog_features() function using the skimage.feature.hog() API.

Here are some examples of what the hog output for all channels of the YCrCb color space looks like for both car and non-car examples:





2. HOG parameters

The list of all parameters used in the project are located in the 2nd box of the P5.ipynb file. The parameters used to extract the HOG features are listed here:

Parameter	Value
color_space	'YCrCb'
orientations	8
pixels_per_cell	12
cells_per_block	2
transform_sqrt	False

After looking at the HOG output pictures, it was decided to use only channel 0 of YCrCb from the examples. Only 8 directions were selected as well for runtime. This would cut down on the runtime of extracting the features of both training and the testing. The pixels_per_cell parameter was increased to 12 to significantly decrease the runtime of the algorithm. There was a trade off between runtime and accuracy.

3. Test/training Data Split and Classifier

Only HOG features were used to train the classifier. The code to train the classifier is located in the 7th block or under the 'Train the classifier' section of P5.ipynb.

The training data was scaled and then centered using sklearn.preprocessing.StandardScaler. This step was optional as we are only using HOG features. If the histogram of color and/or spatial binning of color features were used, then this step would be absolutely necessary.

The training data was randomized and split into training and test datasets using sklearn.model_selection.train_test_split. Test data was 20% of the entire data set.

Linear support vector classification was done using the sklearn.svm.LinearSVC class. Only the default parameters were used. This classifier was used as it is a fast classifier. The accuracy of the classifier was 93%, which was sufficient for vehicle detection if filtering was used.

Sliding Window Search

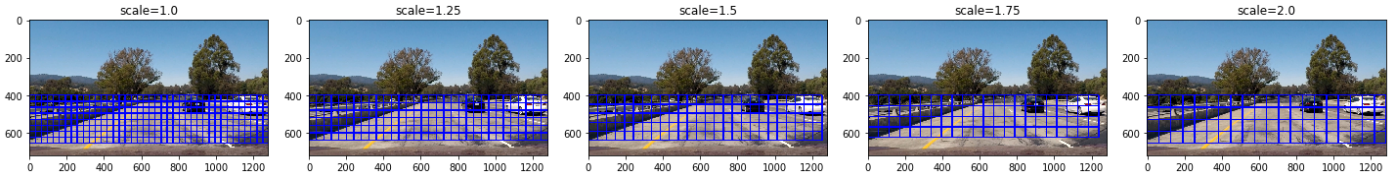
1. Sliding Windows

This is implemented in the find_cars() function in the P5.ipynb notebook.

This is based mostly on the example from the "Hog Sub-sampling Window Search" lecture. The hog features are extracted for the entire picture between the ystart and ystop parameters. The ystart=400 and ystop=656 parameters dictate where in the y-direction to start and stop searching. These were chosen by excluding the sky and horizon, and also the top of the car hood.

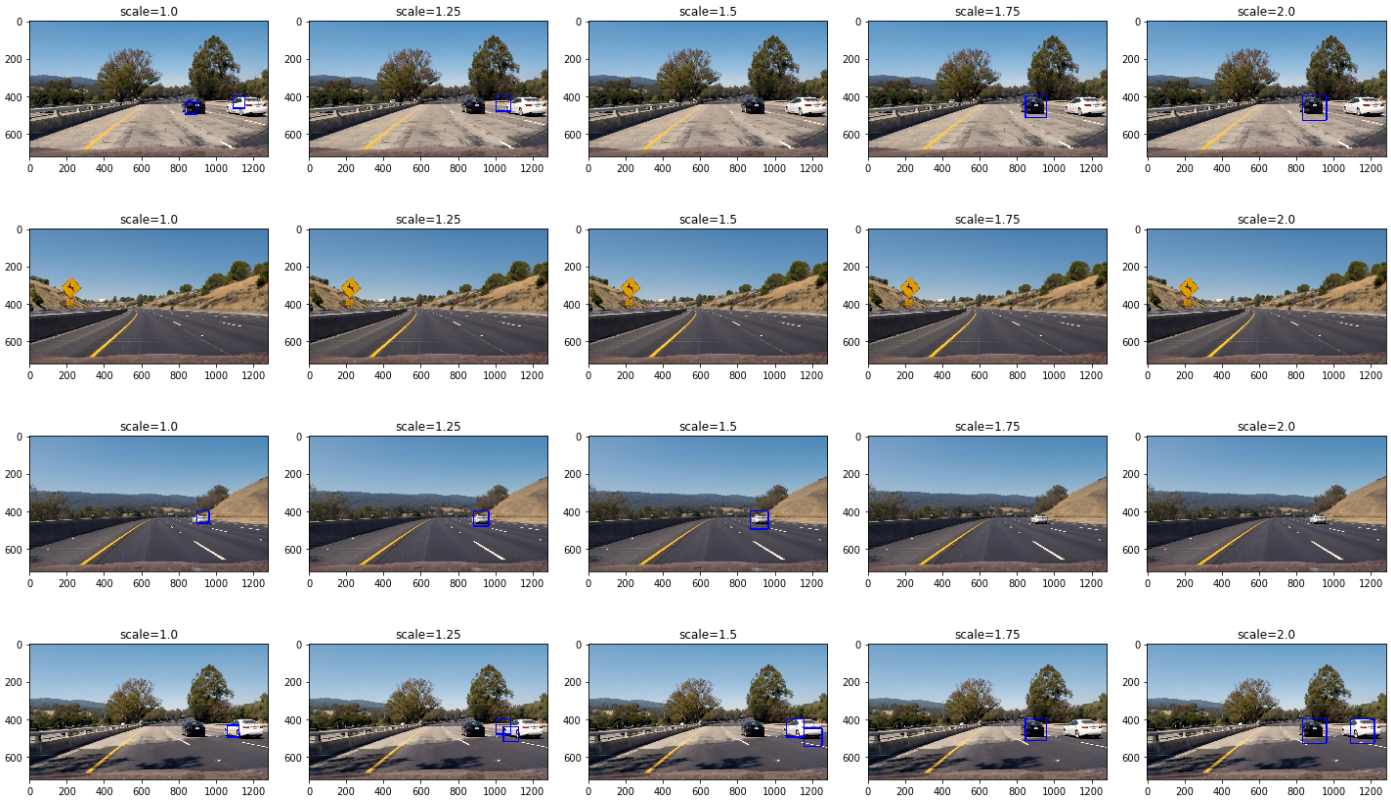
The image is then scaled according to a scale parameter. Then the window is then broken into sub-sampled windows . The same pixels_per_cell parameter used for the example feature extraction were used to sample the smaller windows, which results in 12x12 pixels per cell. When using scale=1.0, there are 107x22 blocks and thus 51x9 steps based on 2 cells per step. This small block is then fed into the linear SVC to produce a prediction on whether is contains a car or not. There are a total of 51x9 steps, which means that 459 predictions are made per video frame.

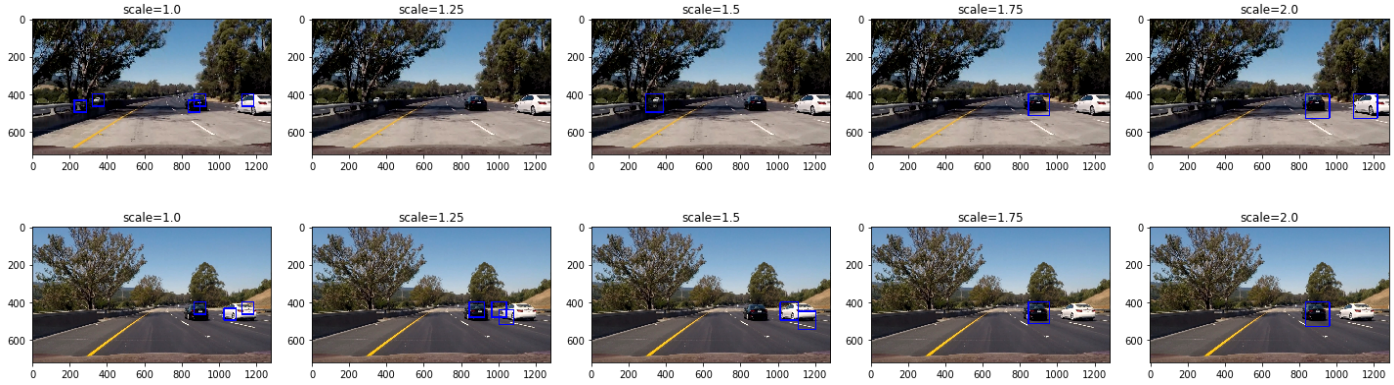
The search space is shown below (when using scale=1.0):



2. Evaluation of the the sliding windows.

I searched using 2 scales (1.50, 2.0) using YCrCb with only the first channel for HOG features. The scale parameters used were chosen by the evaluation on the single images shown below. Since multiple scales are used, the pipeline parameters were chosen to optimize for runtime. Smaller scales seem to cause more false positives, so larger scales were used. Here are some examples of the test images with various scale values:

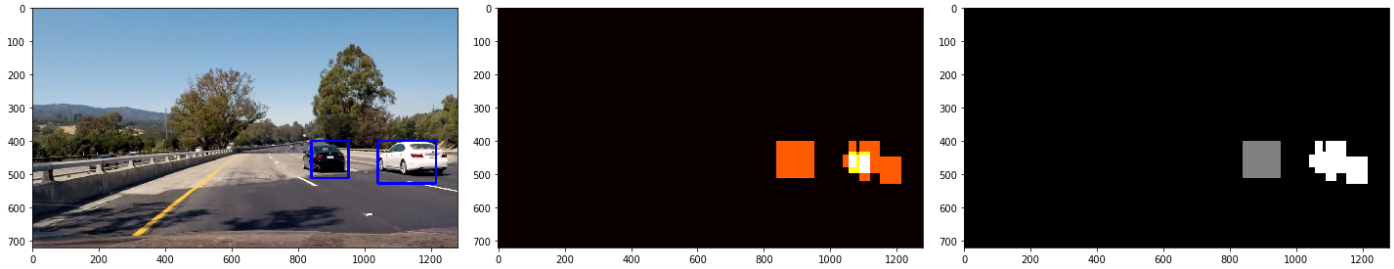




3. Heatmaps

The sliding window approach produces some false positive car detections and multiple detections of the same car. A heatmap approach was used to fix these problems. The sliding windows were run for multiple scales and then combined to create a heatmap. This code is in the heatmaps() function in P5.ipynb.

The detected boxes in the frame for each scale are fed into the function. The heatmap increases by 1 for each detection of that pixel. A configurable threshold was used to filter out low levels of detections. The function `scipy.ndimage.measurements.label()`, was then used to detect the clusters of detected pixels and the number of clusters corresponding to the number of cars. Bounding boxes were used to surround the detected pixels and draw for the output image. The boxes, heatmap, and label output is shown here:



Video Implementation

1. Here's a link to the video with the vehicle detection run on the project video:

Here's a link to my video result (./output.mp4) or you can find it on Youtube (https://youtu.be/BQ6lInzKsjc)

Here's a debug video with a moving heatmap along side the vehicle detection: Youtube (https://youtu.be/-fuJVcWdqN4)

2. Filtering

The same heatmap approach from above was used to filter out false positives and multiple detections between multiple frames. A heatmap using 25 frames was used and a threshold parameter of 15 was chosen. These parameters were chosen based observations of the algorithm on the test video.

Discussion

1. Problems

- Runtime - It takes around 9 mins to process a 50 second video, which isn't real time.
- Multiple cars are usually detected as one car when they are close together.
- Cars that don't look like the training set. There might be cars that are similar in color to the ground or environment.
- Lighting conditions at night. In this case, the cars will not look at all similar to the training set.
- The bounding box could be tighter around the vehicle - would require a set of parameters geared towards smaller pixel/cell sizes and scales.

2. Possible enhancements:

- Search a subset of the windows based on expected next position of the vehicle. This would speed up the pipeline slightly. Keeping track of the positions of the various vehicles would enable tracking of multiple vehicles the are together or stacked from our perspective.
- Use a neural network like YOLO: Real-Time Object Detection (https://pjreddie.com/darknet/yolo/) or SSD: Single Shot MultiBox Detector (https://arxiv.org/abs/1512.02325). These neural networks can run at speed using GPUs.