Vehicle Detection and Tracking

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.

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

The project submission includes the following files:

- Ipython notebook
- Ipython html
- PDF report detailing the rationale behind the current code
- Example Output Images – Test output images folder
- Output Video – Test output video folder

[***Note:*** All important functions are defined in 2nd cell of the iPython notebook]

Rubric 1: Explain how (and identify where in your code) you extracted HOG features from the training images. Explain how you settled on your final choice of HOG parameters.

Rubric 2: 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 investigation was initiated by understanding the data set that was provided as part of the project. The data set consists a bunch of car and non-car images. Here is an example of how the images look:

Total Number of Car Images: **8792**

Total Number of Non-car Images: **8968**

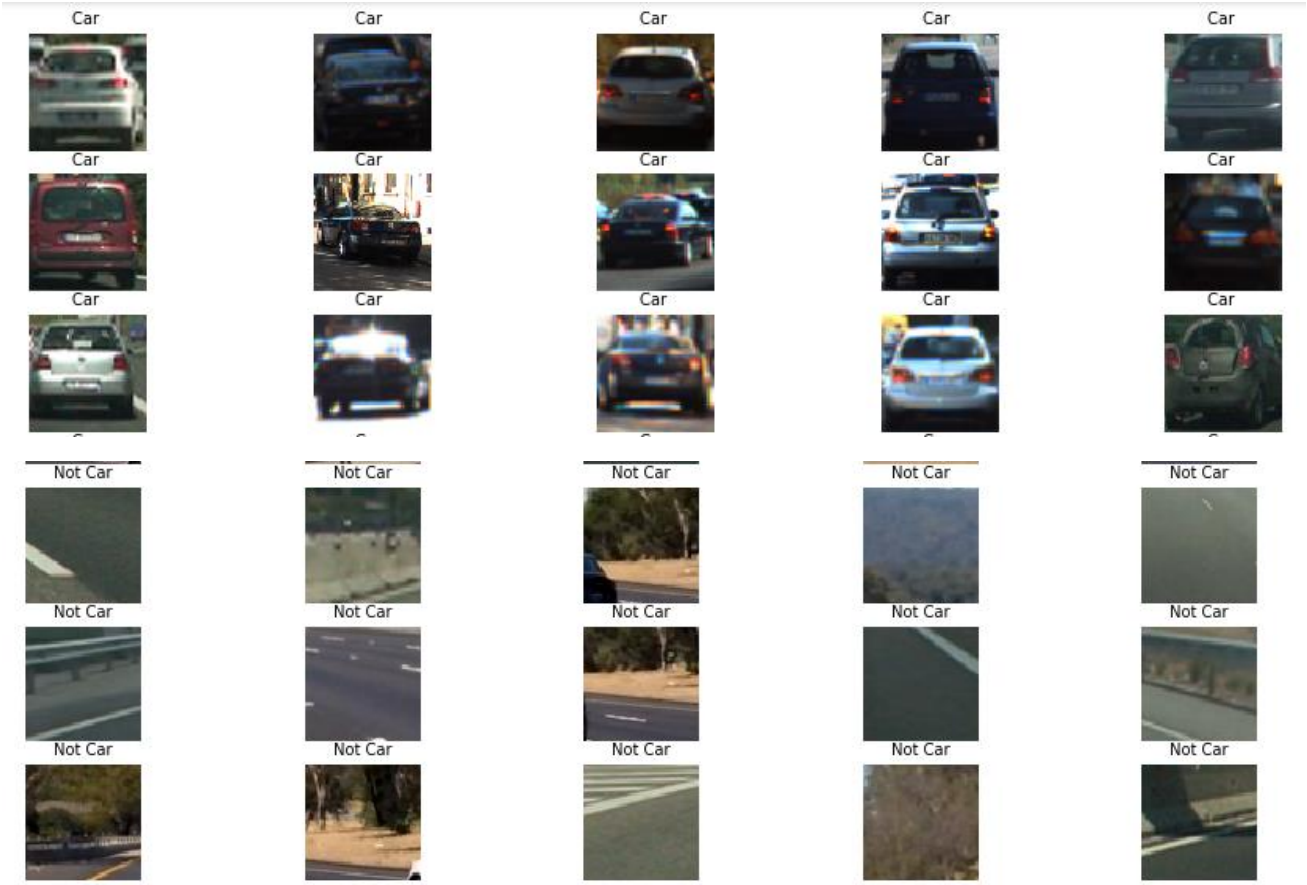

Fig1: Set of Car and Non- Car images from the data set

The above images provide us with the visualization of the dataset. From the numbers above, it is understandable that there are equal number of car and non-car images. The data set can be augmented if needed at a later stage. The attached code, does not involve any augmentation of data.

HOG Features are extracted from an image by running the get_hog_features function defined in 2nd cell of the IPython notebook under the title of "Helper functions". The histogram of gradients (HOG) method divides the overall image into different sub cells and computes a collective/ dominant gradient within each sub cell. Based on the figure below, depending upon the parameters, car and non-car images can be distinguished.

Fig2: Visualization of the HOG Features for Car and Non-Car Images

The function extract_features (in the 2nd cell of the iPython Notebook) returns a list of feature parameters given a list of image paths. This function not only returns HOG features but can combine Spatial Features, Color Histogram Features and HOG Features through an optional switch. The extract_feature function is used in 6th cell of the iPython Notebook on both the car and the non-car images.

Since these are a combination of 3 different features, it is imperative to normalize all the features to a zero mean and similar range of variation. This is highly important to remove dominance from any of the individual feature contributions. This is performed through the StandardScaler function available through sklearn package.

With the features available, a label vector needs to be defined to differentiate car vs non-car classification based on known images. Based on these features and labels, a Support Vector Machine classifier (Linear SVC) was trained to differentiate between car and non-car images. This portion of the code is in 7th cell of the iPython Notebook

Through the switches, a thorough investigation was performed, and the HOG parameters were tuned chosen based on the test accuracy and further testing. Below table indicates the different combinations that were attempted:

Sno.	Features Used	Color Space	Orientation	Pixel/Cell	Cell per Block	HOG Channel	Test Accuracy	Training Time
1	Hog	RGB	8	8	2	0	0.9462	8.5s
2	Hog + Hist	RGB	8	8	2	0	0.9814	7.4s
3	Hog +Spat +Hist	RGB	8	8	2	0	0.9831	8.46s
4	Hog +Spat +Hist	LUV	8	8	2	0	0.989	10.14s
5	Hog +Spat +Hist	LUV	8	8	2	ALL	0.991	8.6s
6	Hog +Spat +Hist	YUV	8	8	2	ALL	0.9932	21.3s
7	Hog +Spat +Hist	YUV	8	8	2	0	0.9859	8.46s
8	Hog +Spat +Hist	YUV	8	16	2	ALL	0.9935	4.26s
9	Hog +Spat +Hist	YCrCb	8	16	2	ALL	0.9924	4.36s
10	Hog +Spat +Hist	YCrcb	8	8	2	ALL	0.9918	8.25s
11	Hog +Spat +Hist	YUV	11	16	2	ALL	0.9942	21.2s
12	Hog +Spat +Hist	LUV	9	8	2	1	0.9879	11.09s
13	Hog +Spat +Hist	YUV	12	16	2	ALL	0.9941	16.5
14	Hog +Spat (32,32) +Hist(80)	YUV	9	8	2	ALL	0.993	23.23
15	**Hog +Spat (32,32) +Hist(80)**	**YUV**	**11**	**16**	**2**	**ALL**	**0.9868**	**15.37s**

All the different combinations had tradeoffs in the form of training time, extraction time (note provided) and test accuracy. The final chosen one was option number 15. The final parameters chosen also depended on the results from the test images in the test images folder.

In the 7th cell of the iPython code, classifier was implemented, and hit & trial method was used to zero in on the chosen parameters. The final accuracy of the classifier built is around ***98.68%***.

Rubric 3: 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?

Rubric 4: Show some examples of test images to demonstrate how your pipeline is working. How did you optimize the performance of your classifier?

1. *Search and Classify - Sliding Window Technique*

Through the code, both the sliding window techniques were attempted. However, the sliding window with multiscale provided very poor results and was discontinued. The Search and classify method described in the lessons was used in the 8th cell of the iPython notebook.

In this sliding window technique, there were 2 windows that were considered based on image car positions in the images. The image y lookup was narrowed to lower half of the image to reduce effort in looking for cars in the sky/on top of trees.(:)).

```
windows = slide_window(image, x_start_stop=[None, None], y_start_stop=[400, 550
                    xy_window=(128, 128), xy_overlap=(0.5, 0.5))
windows+= slide_window(image, x_start_stop=[440, 950], y_start_stop=[420, 470],
                    xy_window=(48, 48), xy_overlap=(0.75, 0.75))
```

The parameters chosen for ystart and ystop indicate the y range limitation of the sliding window search and x start stop is modified to limit smaller windows to particular x range of the image . The overlap was chosen to be a default of 0.5. However, in accordance with the multi scale window approach to get a reliable detection, the overlap was modified to be 0.75 for a smaller window of 48x48 which is required to capture images that are farther away. These 2 windows were chosen to complement one another to look for appropriate images. However, this attempt was cut short due to the better output of Hog Sub sampling based sliding window technique.

[***Note:*** Images for this sub-section will be available through the exported html]

2. *Hog Subsampling - Sliding window Technique*

Through this code, the subsampling based sliding window technique was implemented. This method is implemented in the 9th cell of the Ipython Notebook.

This technique was implemented through the find_cars function (very well defined in the coursework). This function is implemented in the 2nd cell of the code. This method combines Feature extraction with a sliding window search, but rather than perform feature extraction on each window individually which can be time consuming, the features are extracted for the entire image (or a selected portion of it) and the full-image features are subsampled according to the size of the window and fed to the classifier.

The current implementation performs the classifier prediction on the Spatial+ Hist + HOG features for each window region and returns a list of boxes corresponding to the windows that generated a positive ("car") prediction.

Fig3: Find Car detection of test3 image

Before proceeding to the results/finetuned version of the find_cars function, it is imperative to explain the process through which the current results were obtained.

The find_car function takes in 4 parameters that are tunable:

1. ***Ystart & Ystop*** - Ystart and Ystop are used to define the range of Y values through which a sliding window will be implemented and positve detection shall be notified.
2. ***Scale*** -Scale represents the ability to scale the 64x64 window to look for bigger or smaller images. This is how multi-scale window technique is achieved in this method
3. ***Cells Per Step*** -Cells Per step control the level of overlap as the sliding window moves across the Image. For eg: The window is 64x64 is already divided into 8x8 cells and when the cells_per_step is 2- This defines the overlap percentage to be 75 % ((2/8) is 25 percent movement).

The general idea while tuning is to differentiate between cars that are closer and cars that are farther. Two specific rules were used to tune and select these parameters.

1. Lower scaling is better for smaller images and hence can be used to determine cars that are farther away – This determines the ystart and y stop location. Also,The image is more concentrated when its farther away and hence the overlap percentage is tuned to be higher.
2. Higher Scaling values can be utilized for larger /closer images- This also determines the appropriate ystart and ystop. Since the image is bigger and closer, the overlap percentage is tuned to be lesser.

Based on these 2 rules and a lot of attempts (:)), The implementation creates a pipeline based on the following code:

```
# Different Scaling with Different ystart and ystop
rectangles = find_cars(test_img, 400, 480, 1.0, svc, X_scaler, orient, pix_per_c
rects.append(rectangles)

rectangles = find_cars(test_img, 400, 520, 1.5, svc, X_scaler, orient, pix_per_c
rects.append(rectangles)

rectangles = find_cars(test_img, 400, 530, 2.0, svc, X_scaler, orient, pix_per_c
rects.append(rectangles)

rectangles = find_cars(test_img, 450, 620, 3.0, svc, X_scaler, orient, pix_per_c
rects.append(rectangles)
```

Fig4: Finding Car called multiple times to confirm better detection

Another important aspect while tuning is the presence of false positives or no detections that can limit how much the above-mentioned parameters can be tuned. The multiple find_car function output can be observed through the overlapping windows in the following image:

Fig5: Several Overlapping Detections on the test6 img

Because a true positive is typically accompanied by several positive detections, while false positives are typically accompanied by only one or two detections, a combined heatmap and threshold is used to differentiate the two. The add_heat function increments the pixel value (referred to as "heat") of an all-black image the size of the original image at the location of each detection rectangle. Areas encompassed by more overlapping rectangles are assigned higher levels of heat. (10th cell of iPython Notebook)

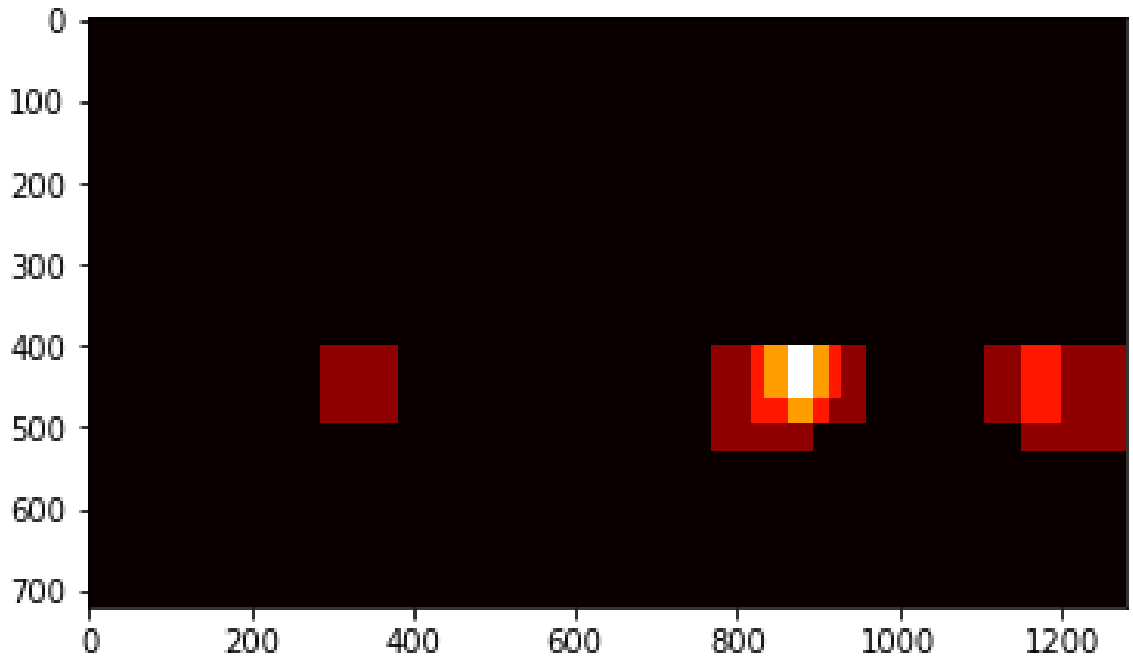

Fig6: Heat Map of the Fig5 based on the overlapping windows

A threshold operation is performed on the heatmap (with a threshold of 1) , setting all pixels that don't exceed the threshold to zero. (11th cell of iPython Notebook)

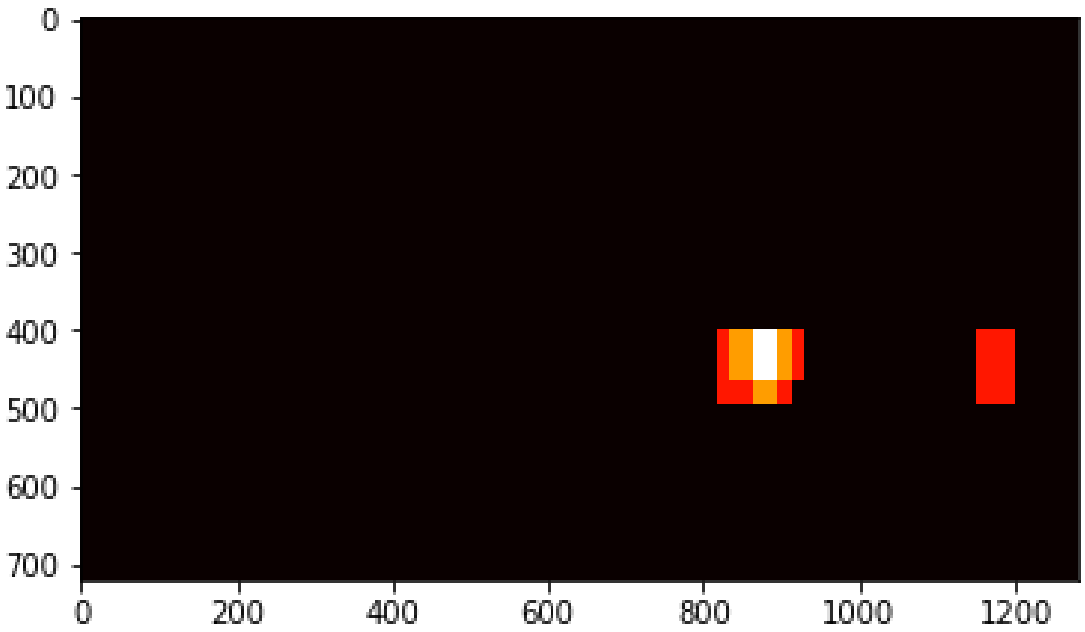

Fig7: Heat Map after threshold application

The Scipy label function is used to collect all the spatially adjacent areas of the ehatmap and assigs each of them a specific label. (12th cell of iPython Notebook)

```
2 cars found
```

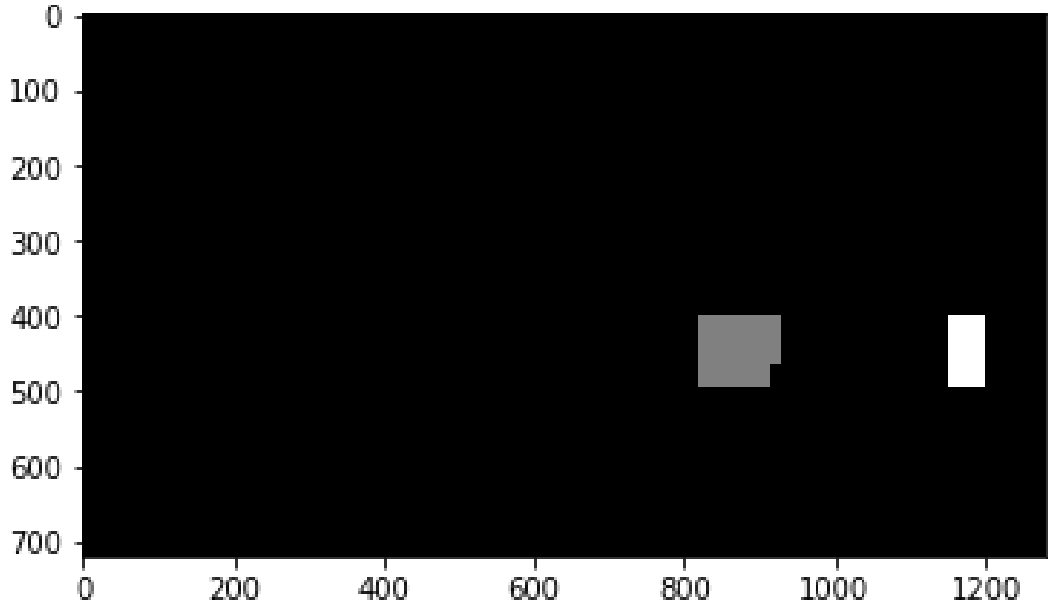

Fig8: Label function through scipy

The final detection area is set on the original image based on the labels(13th cell for example image):

Fig9: Detected Labels on top of Original image

Several iterations of this was performed with different feature vectors. It was initially implemented to include only hog features and then furthered by adding hist and spat features. This helped in ruling out false positives and further strengthened confirmed detections.

Pipeline for the image processing - 14th cell of the ipython notebook
Test images processed - 15th cell of the ipython notebook

Fig10: All test images except test3 are successful (with same values of heat threshold)

Note: Test 3 image fails due to lower heat levels – Still detected by finding cars but threshold rules out that detection since lower than threshold. Choosing any threshold lower than 1 (at around 0.8) confirms the detection

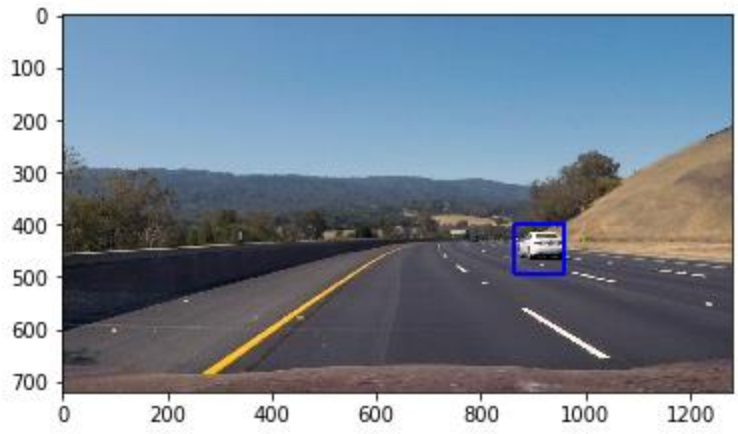

Fig11: Test3 successful with 0.8 heat Threshold

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 if you are identifying the vehicles most of the time with minimal false positives.)

Please find the processed video in the 'test_video_output' folder with advanced pipeline as well as normal pipeline. Also available by executing 16,17,18th cell of the iPython Notebook.

Describe how (and identify where in your code) you implemented filter for false positives and some method for combining overlapping bounding boxes.

The code for processing frames of video is contained in the cell titled "Improvements to Object Detection pipeline". It is contained within the 19th and 20th cell of the iPython Notebook.

The code is identical to the code for processing a single frame of image described above, except for storing the detections (returned by find_cars) from the previous 12 frames of video using the old_rects parameter from a class called veh_detect. Rather than performing the heatmap/threshold/label steps for the current frame's detections, the detections for the past 12 frames are combined and added to the heatmap and the threshold for the heatmap is set to 1 + len(old_rects)//2. This value/math was found by hit and trial by selecting several constant thresholds and finding them to be insufficient.

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?

After a personal struggle with advanced lane lines, this was one of the relatively less coding intensive projects. There was a lot of time spent on fine tuning and experimentation using different features and parameters. However, the platform provided by the coursework was pivotal to visualization and completion of this project.

The pipeline was created using the linear SVC classifier based on a relatively small data set of car and non-car images. These are some of the shortcomings and future improvements that can be implemented.

Shortcomings:

1. The current implemented classifier is not fine-tuned and can be tuned further to achieve higher level of performance.
2. Lesser data can confuse the classifier causing it to provide misclassifications or false positives – Can be mitigated by either augmenting data set with rotated/flipped images or having more data
3. Real time processing requires understanding of appropriate tradeoffs between high accuracy detection and time taken for the detection
4. Multiple environmental conditions such as snow or lighting can confuse the classifier since a white car and snow might have similar hog features.
5. The ability to detect oncoming cars is relatively tough.
6. Cars on the side of the images can be relatively tough to detect with current implementation – Can be tuned to look for more windows; Feed partial images of cars to train classifier

Further improvements:

1. Different combination of the Hog parameters and classifier – GridSearchCV to fine tune the classifier and choose an rbf kernel for the SVC; Potentially other classifiers such as decision tree,etc.
2. Tagging of individual vehicles once they are in the frame and indirectly predicting their motion in subsequent frames
3. Dynamic Heat Thresholds for vehicles that are farther away- Can make this classification of farther cars based on vehicle speed to simulate look ahead distance. Still need to be careful about noises and false positives if using such an approach
4. Using a convolutional neural network, to avoid the sliding window search altogether