Vehicle Detection & Tracking Project

Introduction:

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.

Rubrik Points

This write up will address each of the rubrik points.

The code for this project is run from main.py. Each stage of the project is in a separate section that the user can step through.

Note: I had wanted to summarize/convert this to a jupyter notebook at the end of my development, but had time constraints.

Histogram of Oriented Gradients

To determine the best features for use in the vehicle detection classification, a procedure ***explore_feature_extraction_parms()***, lines 178-251 in ***feature_explore.py*** module (& invoked from main) was developed. In this procedure, an exhaustive search of the various parameters was performed using the small data set of “cars” & “notcars” provided in the lessons. The classifier, a SVM, was run on each combination and a score obtained for that combination, and the results printed. To reduce the exploration space/time, the exploration was done in a particular order & the results from the previous stage was selected to continue the exploration.

Although, you can see the full color space commented out now, all the colour spaces, hog_channel_parms, orient_cells, and so on were examined as seen in the comments part.

```
colorspace_parms = ['YUV', 'YCrCb'] # ['RGB', 'HSV' , 'LUV', 'HLS', 'YUV',
'YCrCb']
hog_channel_parms =  [2, 'ALL'] # [0, 1, 2, "ALL"] # Can be 0, 1, 2, or "ALL"
orient_parms = [9, 10] # [9, 8, 7, 10, 11, 12]
pix_per_cell_parms = [8, 16] # [8, 16, 32, 4]
cell_per_block_parms = [2]   # [2, 4, 8, 1]
spatial_size_parms=[(32,32), (16,16)]
hist_bins_parms=[32, 16]
```

From this exploration, the best set of parameters were then manually selected, based on the score values produced during the exploration & with an understanding of the efficiency of later parts of the program. See line 71 in main.py for the final selection.

```
BEST_PARMS=['YCrCb', 9, 8, 2, 'ALL', (16, 16), 32]
```

Once, the best parms were selected, then the same SVM classifier was run again with the large data set that was provided for this project, and the results of that were pickled to **svc_car.p**. The code for doing this, again invoked from ***main.py,*** can be seen in ***run_classifier_and_pickle()*** in ***feature_explore.py*** lines 343-367. This pickled classifier & its feature scaler, & feature selection parms were used later in the project to extract features from images/frames & detect cars in the various figures & videos. Also, prior to fitting the classifier, the features were scaled to zero mean & unit variance through the use of sklearn **StandardScaler().**

It should also be noted that an exploration was developed & performed to also examine the type of classifier & the classifier parameters such as using a non-linear SVM and some of the classifier parms (such a “C”). In the end, the use of a non-linear SVM was rejected as it took a very long time for the classification to run & be fit, and the resulting improvement in score was only marginal at best. The code for this can be seen in ***module feature_explore.py***, lines 371-414 in procedure ***explore_classifier_parms()***.

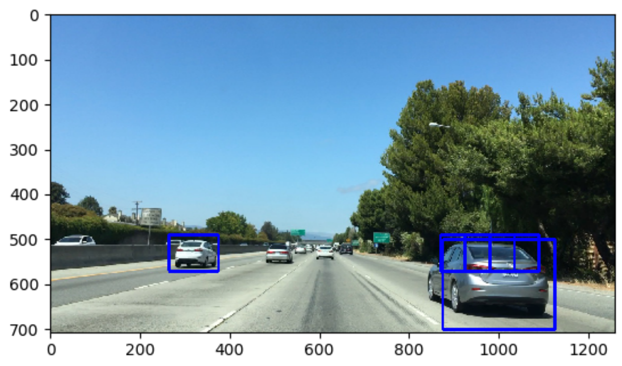
```
Sample Output of the exploration phase:

... snip

 Sorted Summary Results for Parm Exploration
   ['0.9892', 'YCrCb', 9, 8, 2, '2', 32, 32, 16]
   ['0.9892', 'YCrCb', 9, 8, 2, '2', 16, 16, 16]
   ['0.9935', 'YCrCb', 9, 8, 2, '2', 32, 32, 32]
   ['0.9935', 'YCrCb', 9, 8, 2, '2', 16, 16, 32]
   ['0.9957', 'YUV', 9, 8, 2, 'ALL', 32, 32, 32]
   ['0.9957', 'YUV', 9, 8, 2, 'ALL', 32, 32, 16]
   ['0.9957', 'YCrCb', 9, 8, 2, 'ALL', 32, 32, 16]
   ['0.9978', 'YUV', 9, 8, 2, '2', 32, 32, 32]
   ['0.9978', 'YUV', 9, 8, 2, '2', 32, 32, 16]
   ['0.9978', 'YUV', 9, 8, 2, '2', 16, 16, 32]
   ['0.9978', 'YUV', 9, 8, 2, '2', 16, 16, 16]
   ['0.9978', 'YUV', 9, 8, 2, 'ALL', 16, 16, 16]
   ['0.9978', 'YCrCb', 9, 8, 2, 'ALL', 32, 32, 32]
   ['0.9978', 'YCrCb', 9, 8, 2, 'ALL', 16, 16, 16]
   ['1.0', 'YUV', 9, 8, 2, 'ALL', 16, 16, 32]
   ['1.0', 'YCrCb', 9, 8, 2, 'ALL', 16, 16, 32]

 Continuing with parms 'colorspace' & 'hog_channel' set to:  YCrCb ALL

 Exploring parms: 'orient' .

 === Extracting Features & Running Classifier ===

...snip
```

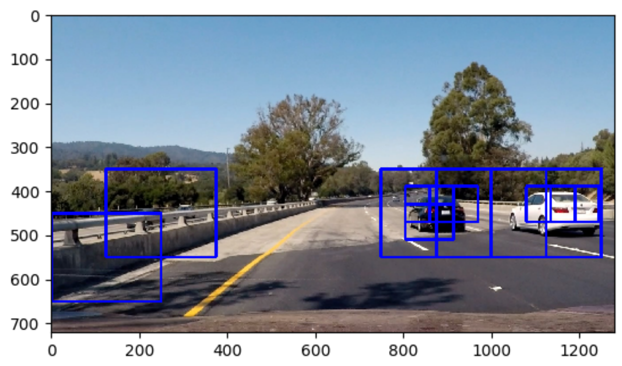
Sliding Window Search To Detect Vehicles:

Two methods of sliding windows were implemented as shown in the lesson, the traditional but inefficient sliding window & the more efficient hog_subsampling where feature extraction is only done once per frame and then subsampled.

The first, "sliding window" algorithm was implemented in module ***sliding_window.py*** & explored by invoking from main.py, lines 372-448. Two sizes of windows were explored. This technique was not used in the pipeline to detect cars. The results of this is shown below:

```
SLIDING WINDOW (2 window sizes) Examples with FP's:
```

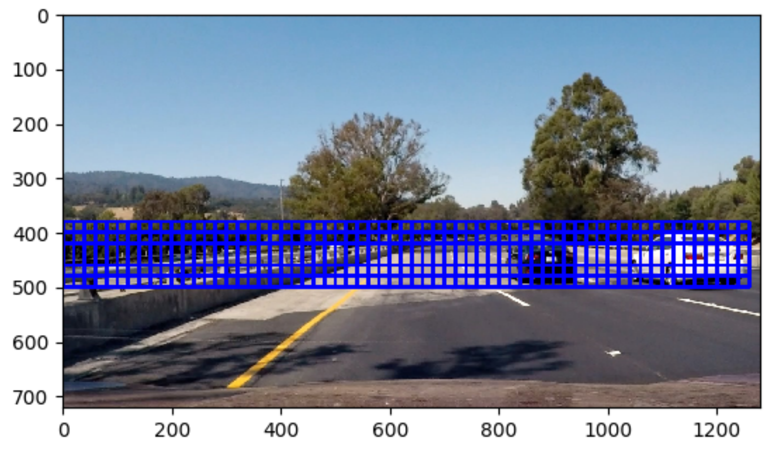




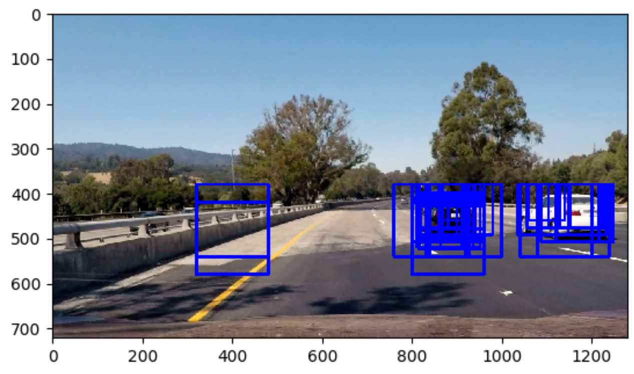
The second, hog_subsampling algorithm was implemented in ***hog_subsample.py*** lines 34-135 in procedure **find_cars()** and invoked/explored from main.py lines 453-552. Five sizes of windows were used, and this was also used later in the pipeline to detect cars.

```
find_cars_search_area = [[ 380, 660, 2.5],  # 0
                        [ 380, 656, 2.0],  # 1
                        [ 380, 656, 1.5],  # 2
                        [ 380, 550, 1.75], # 3
                        [ 380, 530, 1.25]] # 4
```

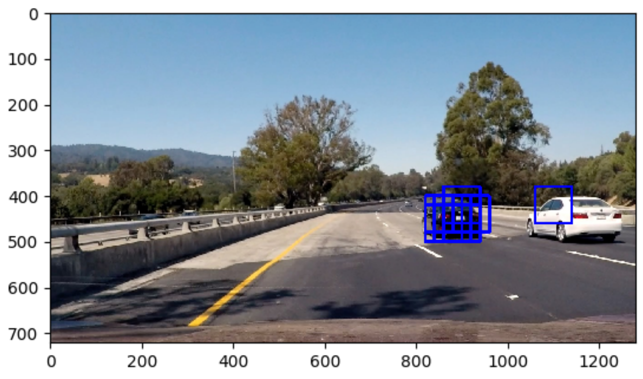
HOG SUB_SAMPLING SEARCH AREA Example:



HOG SUB_SAMPLING One scale (1.5) Example



HOG SUB_SAMPLING MULTIPLE SCALED WINDOWS SEARCHING RESULTS Example



Removal of Duplicates & False Positives:

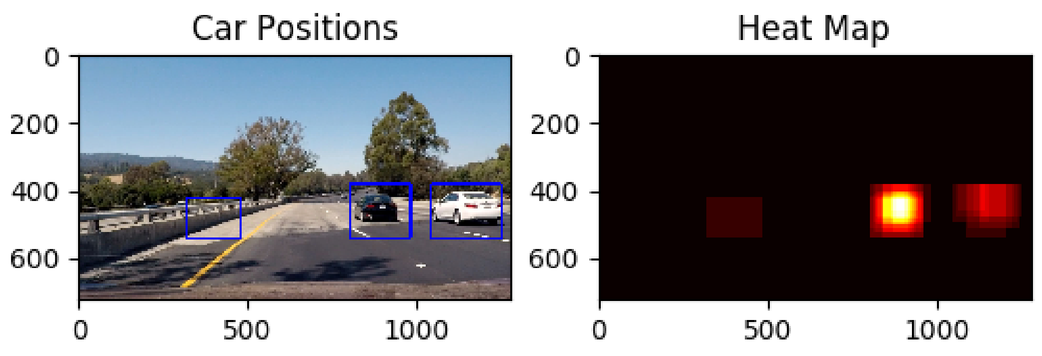
Two techniques to remove duplicate car detections & false positives were explored: the use of "heatmaps" and the use of the classifier's "decision function".

Heatmaps:

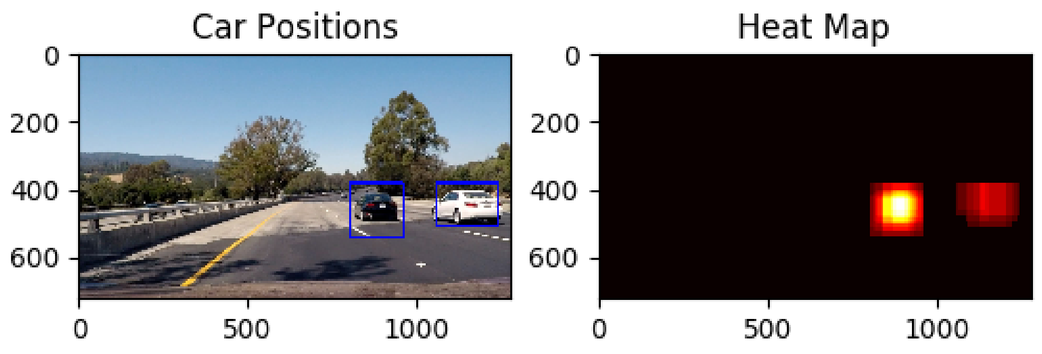
The first technique, using "heatmaps" , was explored in ***main.py*** lines 561-621, and was found to be quite effective at removing duplicates and false positives on individual test images. From this, a false positive threshold was established to be used that yielded the best results.

```
# Apply threshold to help remove false positives
FP_THRESH = 2 # 1
```

USE OF HEATMAPS & THRESHOLDING of "1" Example:



USE OF HEATMAPS & THRESHOLDING of "2" Example:



This "heatmap" technique was also integrated later in the pipeline to work over several frames. A heatmap was implemented there that spanned a number of frames (NUM_FRAMES_TO_TRACK=6), that resulted in true vehicle detections to get "hot" based on true & same detections over several video frames, while insuring that transient & false detections remained “cool”. The threshold number was set to be a factor of the individual threshold & the number of frames (FP_FRAME_THRESH = FP_THRESH * NUM_FRAMES_TO_TRACK) See lines 267-300 in procedure ***carDetectPipeline()*** in ***main.py***.

Decision Function/Confidence Score:

The second technique, use of a “decision function “was also explored and used during the pipeline development phase. When the classifier was predicting the identification of a car, the classifier’s decision function was invoked to determine that prediction’s confidence score - see lines 110 & 125 in the ***find_cars()*** procedure in **hog_subsample.py**. This confidence value was then stored along with the possible car detection locations. Later, a threhold value was set to 0.1 (CONF_THRESH = 0.1) to reduce the number of false predictions in the pipeline, lines 225-232, by calling procedure ***apply_threshold_of_confidence()*** , implemented in module ***draw_bboxes.py*** lines 63-71, from line 227 in the ***carDetectPipeline()*** in ***main.py.*** The threshold value of 0.1 was obtained experimentally.

However, this technique of using the confidence factor did not have as a dramatic effect as the heatmap technique. A better method of using the confidence values over several frames could have been explored, along with a better approach to determining an idela value & dynamically thresholding it than just using a hard-coded value.

Car Detection Pipeline:

A pipeline function ***carDetectPipeline()*** lines 165-305 in module ***main.py*** was created to perform all the processing on individual frames to search for cars in that frame (using hog_subsampling technique), to detect & remove duplicate car detections & false positives, and to draw bounding boxes on the final cars remaining in the frame. If test images were being processed by this pipline then, the resulting images were optionally written to disk for use with submitting this project.

A procedure ***test_carDetectPipeline()***, lines 90-95 in main.py was invoked to test the pipline with the project’s set of 6 test images, and the results of each phase for each test image were written to disk. These can be seen in the project file submissions under output_images/output_test_images. Examples for test1 are provided in this document.

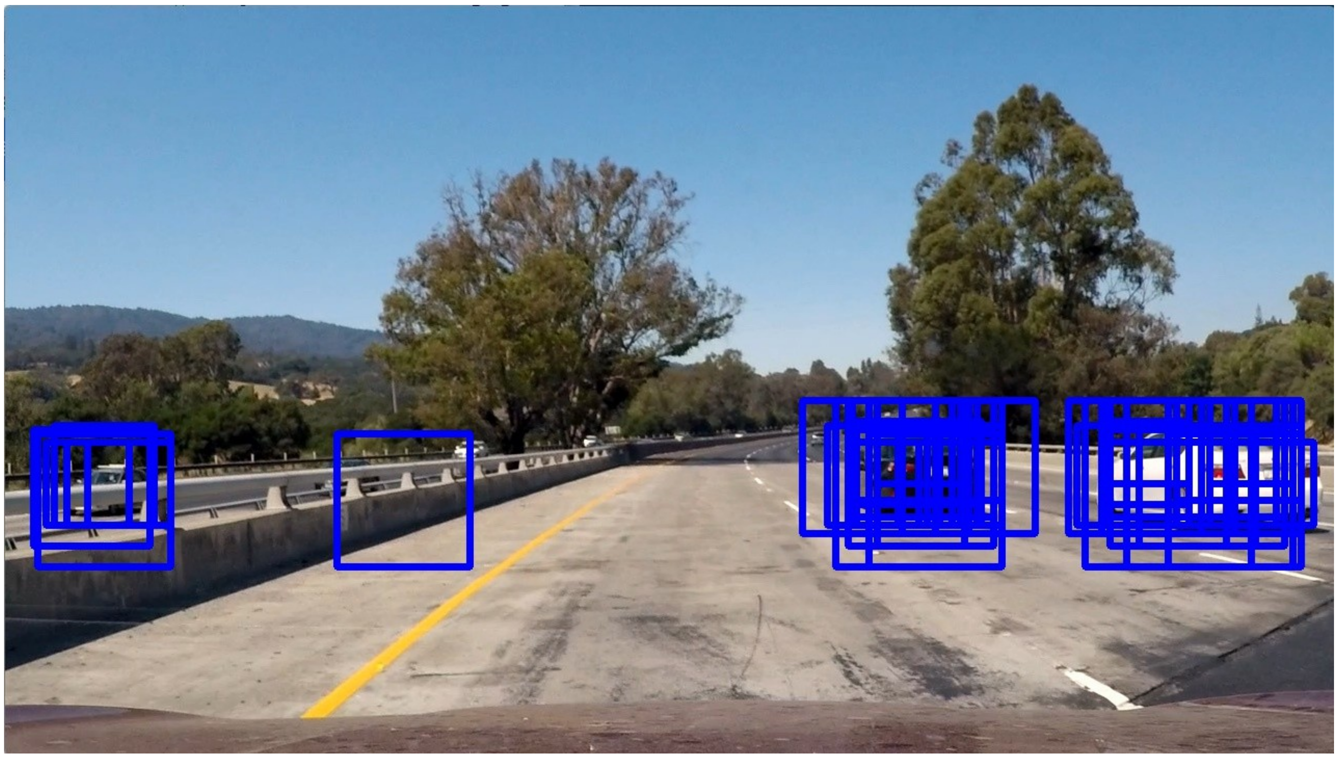
Also, the prototyping of the pipeline was initially explored in lines 630-638 of **main.py** with one test image.

The following steps can be seen in this car detection pipline, ***carDetectPipeline()*** lines 165-305 in module ***main.py:***

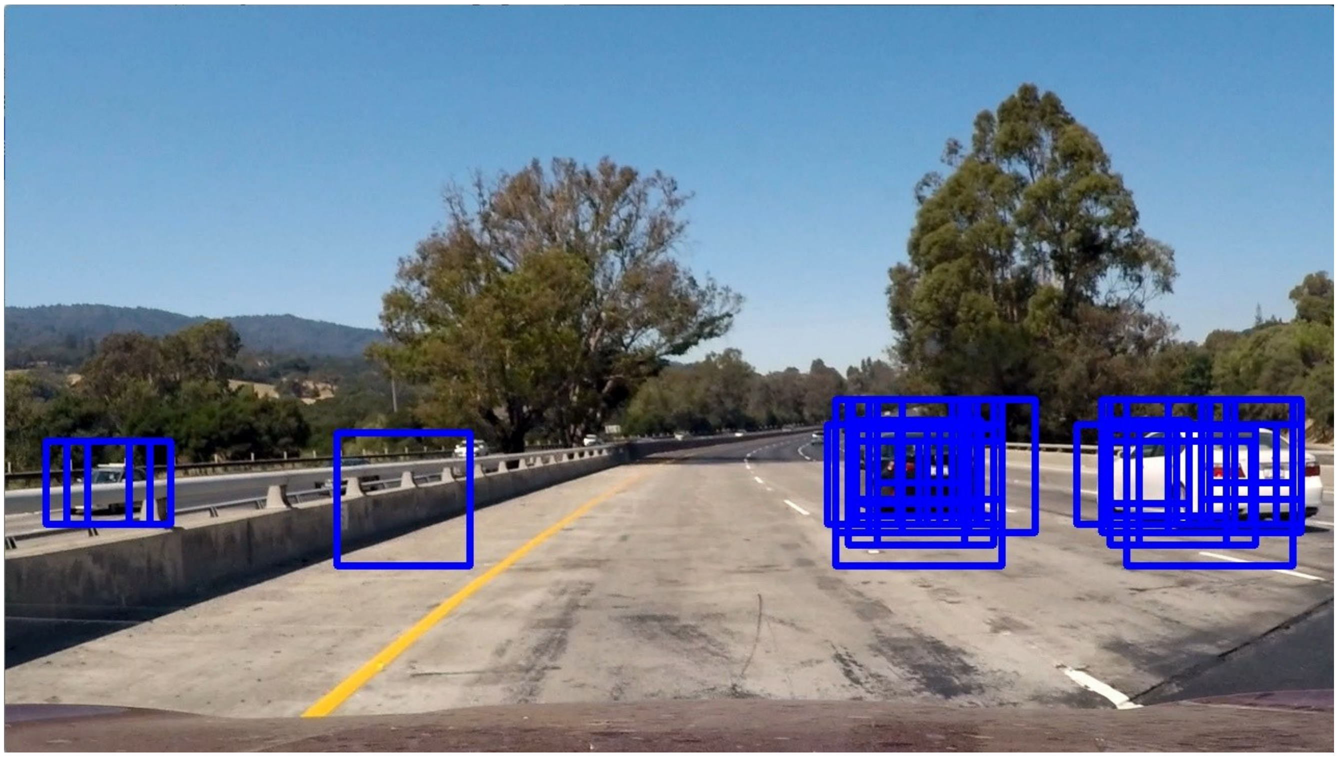
Step 1: Load the trained/fitted classifier function, parameters, and scaling function from a pickle file that had been saved earlier after the classifier had been trained/fitted with the large data set provided with this project. See lines 168-187 in ***main.py*** in ***carDetectPipeline()***.

Step 2: Search for & identfy a list of potential cars, and the confidence values, in the frame/image using 5 different window sizes & the hog_subsampling technique that has already been explained earlier in this write-up. See lines 189-222 in ***main.py*** in ***carDetectPipeline().***

See below for example of output from processing test1.jpg as per the pipline Step 2. Detection of all possible cars.

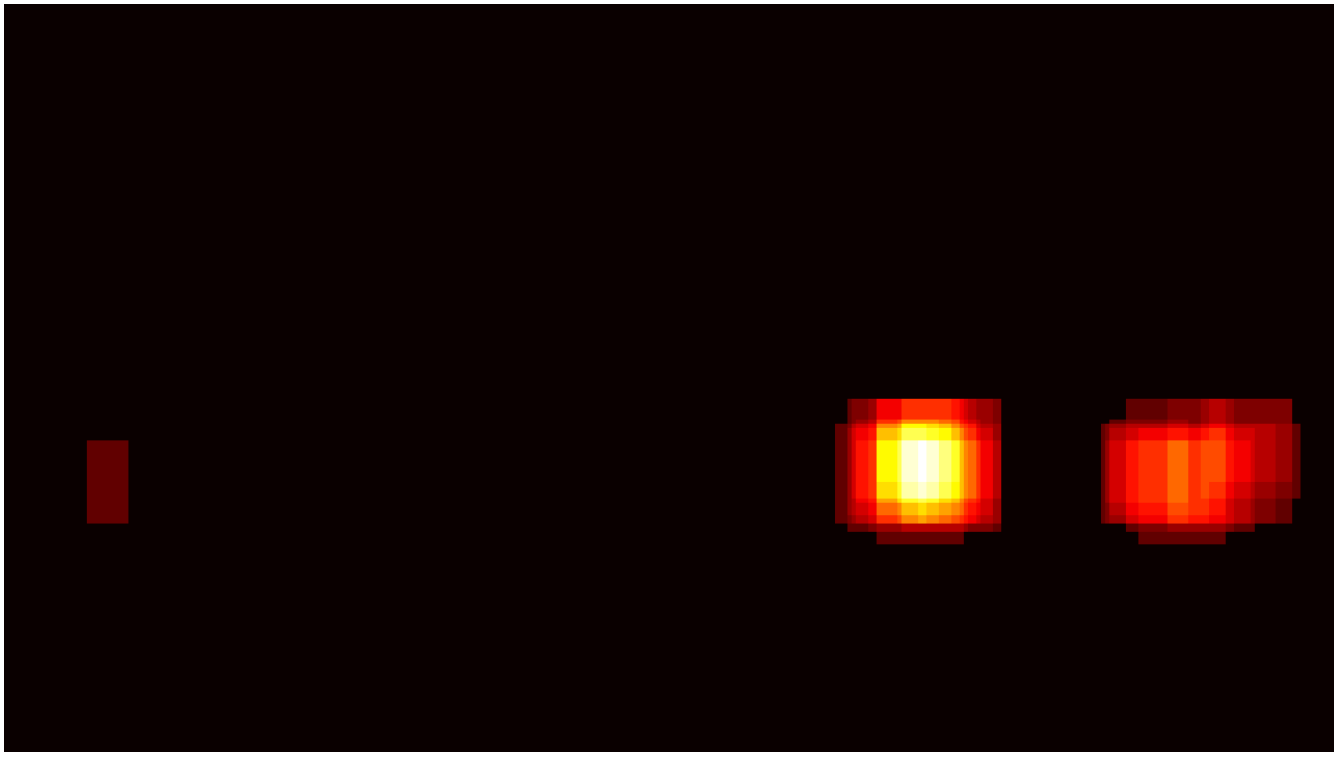


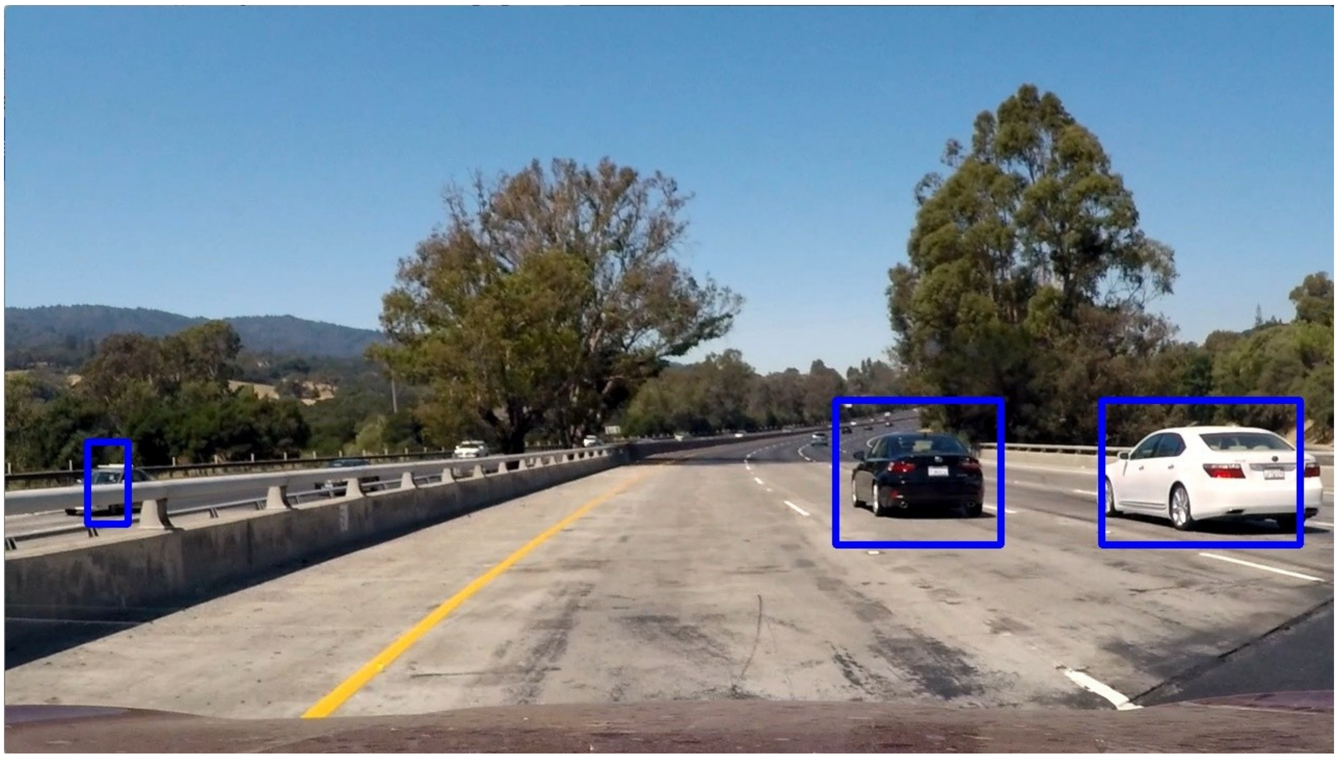
Step 3: Perform thresholding on possible car detections using the confidence value of each possible car detection & a hard-coded thresholding value. See lines 225-232 in ***main.py*** in ***carDetectPipeline().*** An example of the output of this step can be seen below. Only a few duplicates & fp's are removed.



Step 4: Store frame state information, ie) the possible car detections & corresponding confidence values for the frame/image, for use in later frames, to be able to remove duplicates & false positives if processing video. See lines 233-236 in ***main.py*** in ***carDetectPipeline().*** It was decided to store frame information for 6 frames. However, some exploration could be done to determine if better results could be obtained with use of different number of frames.

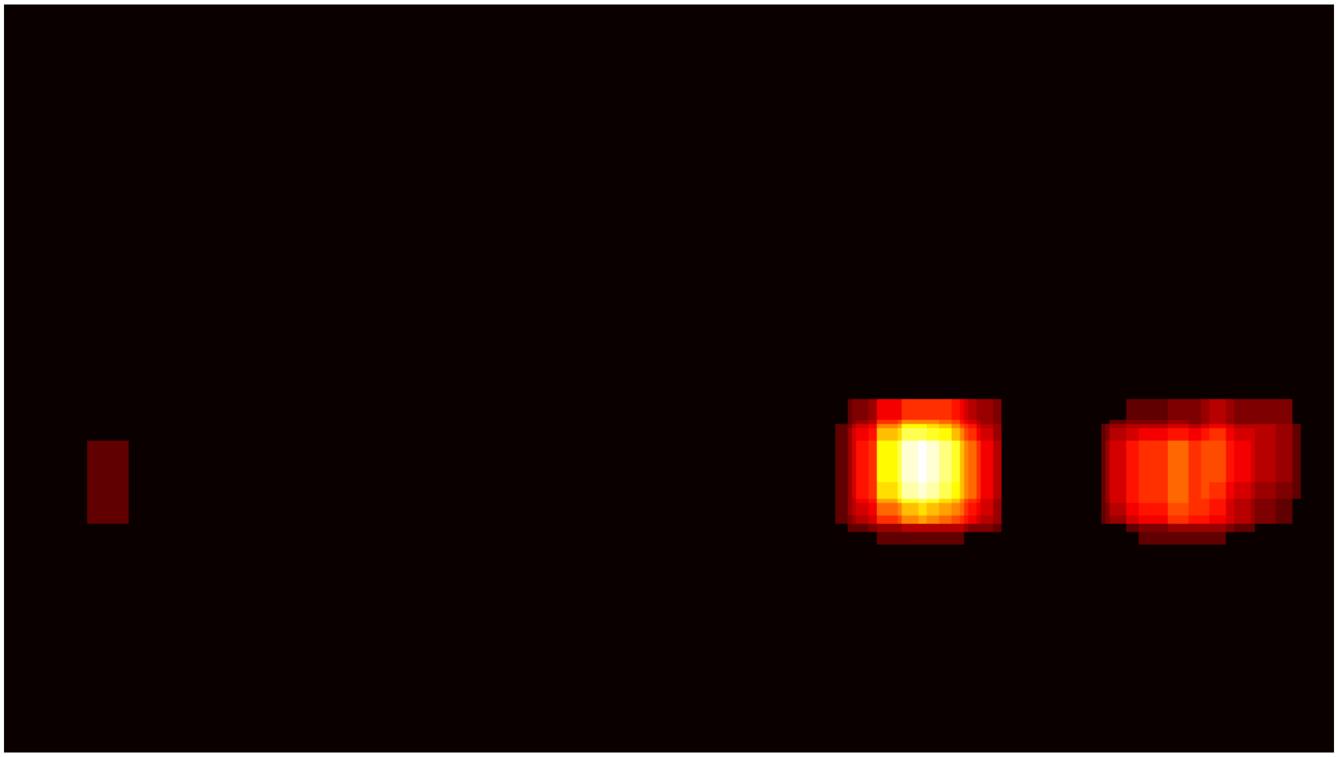
Step 5: Remove duplicates & false positives for use with test images, using heatmap technique as explained earlier. See lines 238-266 in ***main.py*** in ***carDetectPipeline().*** An example of the heatmap that is produced from one image/frame is shown below (for test1.jpg) amd the subsequent test1 image after applying the heatmap.

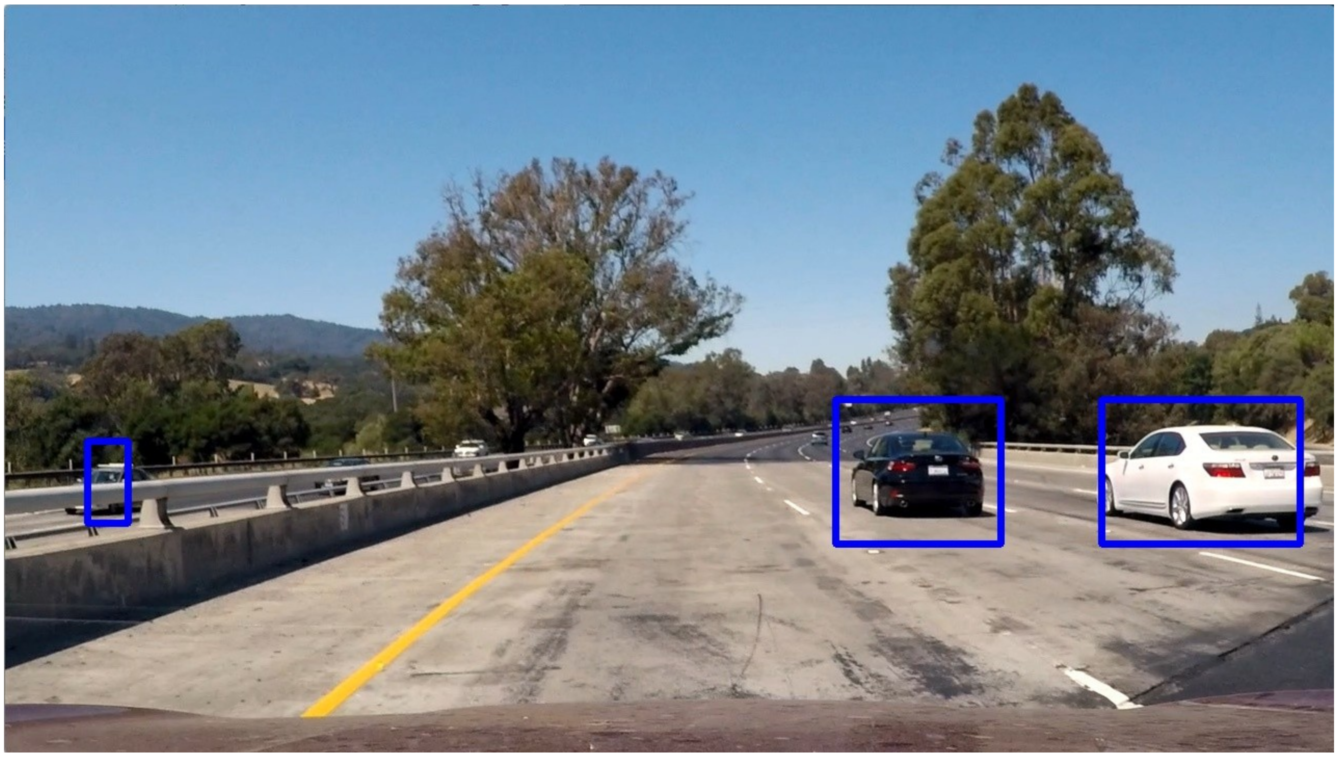




Step 6: Remove duplicates & false positives for use with video, using heatmap technique over multiple frames, as explained earlier. See lines 267-302 in ***main.py*** in ***carDetectPipeline().***

The example of this is shown in the next two figures, integrated heatmap sums & resulting image processed over several frames. However, since the example is just for one test image, it shows the same result of step 5, in this case when working with a simgle test image. For the results of this step one has to view the video. But this was done so as to see that the same code would work with both static images & frames.





Pipeline (Video):

The ***carDetectPipeline()*** was also used to process both the shorter test video and the longer project videos. The code for this video producing portion can be seen in lines 646-663 in ***main.py***. The output is provided under the directory ***output_videos***, in files ***test_video_from_single_pipeline.mp4*** & ***project_video.mp4*** accordingly. In the video outputs, one can see that cars are successfully being detected and tracked throughout the videos. **Please review and evaluate these videos.**

Combining Project 4 & 5: Not for evaluation

As well, an attempt was made to combine both the car detection pipeline & the advanced lane finding pipeline (see lines 671- 694 in main.py). The results can be seen in *test_video_from_combined_pipline.mp4* & *project_video_from_conbined_pipline.mp4*. However, although results were obtained that showed the lane lines & the cars being detected, the quality of car detection was not as good as just the use of the single pipeline. This is only being provided to show that this was attempted but should not be used as the final results. Unfortunately, due to time contraints, no attempt was made to investigate and debug this difference.

Discussion:

This section provides a discussion of problems/issues faced in the implementation of the project.

Jitter & Size of Boxes Surrounding Detected Cars:

As can be seen in the final video, the bounding boxes surrounding the detected cars jitters & varies in size. This should be smoothed in a better fashion.

Also, as some cars were passed and the previously detected cars left the field of view, it can be observed that the bounding boxes leave the cars before the car is entirely out of view. This could be corrected to widen the field of view for looking for cars but was not done due to time constraints. Also, in a real car, one is also interested in the cars behind ones self so this would also apply to be able to detect all cars within a “crashing” vicinity of the automated vehicle ie, on all sides in front and back.

Detection of Vehicles on Opposite Lanes:

As can be seen in the video, there are occasional bounding boxes in the divided highway in the opposing lane. While cars are being detected properly there, it seems that this would be distracting and irrelevant to the automated vehicle. Better technique of only focusing on detecting cars in appropriate locations could be done.

Lack of Error Detection & Handling:

Again due to time contraints for this project, there really was no error analysis or checking that was done, and no software error handling provided. Functions could fail for many reasons and the result would be the program crashing. This would not be acceptable for production level code.