
VEHICLE DETECTION PROJECT

TABLE OF CONTENTS

0. Objective and Goals	1
1. Histogram of Oriented Gradients: Feature Extraction	2
2. Color Histograms & Spatial Binning	4
3. Support Vector Machine Classifier Training	6
4. Detection (Sliding Window Technique) & Prediction	8
5. Eliminating False Positives	16
6. Pipeline	22
7. Discussion	23
References	24

OBJECTIVE

This project requires the developer to develop a pipeline to identify vehicles from car-mounted camera feeds of highway driving.

GOALS

Project goals were the following:

1. To extract features from a Histogram of Oriented Gradients (HOG) on a labelled training set of images; train a Linear SVM classifier
2. Optionally, to apply a color transform and append binned color features, as well as histograms of color, to the HOG feature vector.
3. To normalize features, randomize a selection for training and testing for steps 1, 2
4. Implement a sliding-window technique and use the trained classifier to search for vehicles in images.
5. To develop a pipeline on a video stream (starting with the test_video.mp4 and later implement on the full project_video.mp4) and create a heat map of recurring detections frame by frame to reject outliers and follow detected vehicles.
6. To estimate a bounding box for vehicles detected.

Note: All of the code developed is in the file p5-final.py, and references from here onward will refer to it thus, [1].

1. HISTOGRAM OF ORIENTED GRADIENTS (HOG):

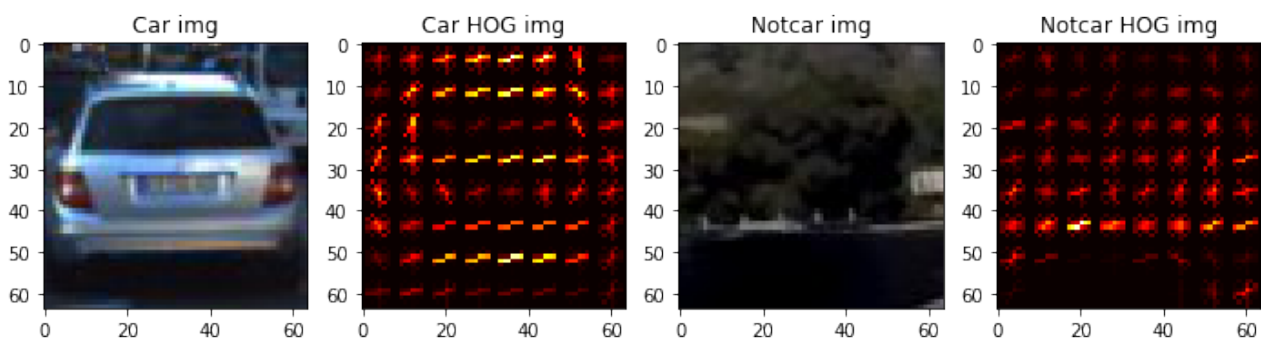
FEATURE EXTRACTION

HOG is useful as a feature descriptor that simplifies a digital image, small or large, by extracting useful information while discarding the rest. It is useful as input to classifiers such as a support vector machine (SVM). The process yields a one-dimensional array (called a feature vector) with values calculated thus:

1. Calculate the gradient along the X / Y directions to identify edges while discarding the remaining information. See Open CV's Sobel() function.
2. Calculate the magnitude and direction of the two gradients

```
abs_sobelx= $\sqrt{(sobelx)^2}$   
abs_sobely= $\sqrt{(sobely)^2}$   
abs_sobelxy= $\sqrt{(sobelx)^2+(sobely)^2}$  – Magnitude  
Angle  $\theta$  =  $sobely / sobelx$  – Direction
```

This yields an image with a magnitude and a direction of that gradient, arranged approximately from 0-180°. Running these operations on the Udacity-provided image dataset (datasets of images of cars and of “not cars”) extract the following information from such images. An example of each type of image is shown next:



It is worth noting that there is a discernible outline to these features corresponding to the images. These images were extracted using the `single_img_features()` function in [1], line 220.

```
car_features, car_hog_image = single_img_features(car_image,  
    color_space=color_space, spatial_size=spatial_size, hist_bins=hist_bins,  
    orient=orient, pix_per_cell=pix_per_cell, cell_per_block=cell_per_block,  
    hog_channel=hog_channel, spatial_feat=spatial_feat, hist_feat=hist_feat,  
    hog_feat=hog_feat, vis=True)
```

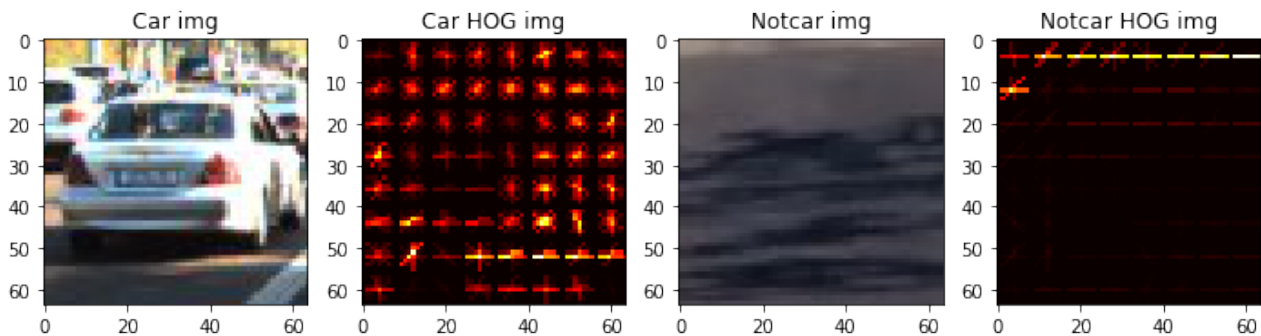
```
notcar_features, notcar_hog_image = single_img_features(notcar_image,  
    color_space=color_space, spatial_size=spatial_size, hist_bins=hist_bins,  
    orient=orient, pix_per_cell=pix_per_cell, cell_per_block=cell_per_block,  
    hog_channel=hog_channel, spatial_feat=spatial_feat, hist_feat=hist_feat,  
    hog_feat=hog_feat, vis=True)
```

Various parameters (such as the ones shown below) were used in several combinations

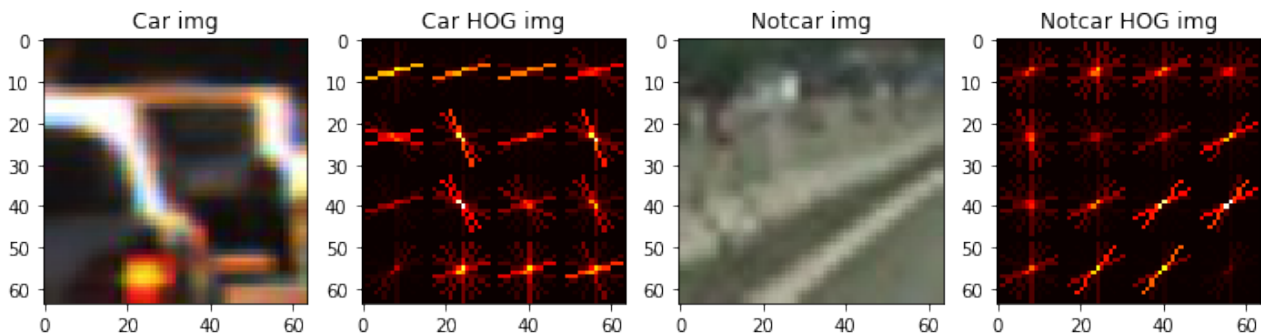
to get a better idea of the HOG results.

```
#Set up parameters for the functions to be called
color_space = 'RGB' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9 # HOG orientations
pix_per_cell = 8 # HOG pixels per cell
cell_per_block = 2 # HOG cells per block
hog_channel = "ALL" # Can be 0, 1, 2, or "ALL"
spatial_size = (32, 32) # Spatial binning dimensions
hist_bins = 16 # Number of histogram bins
spatial_feat = True # Spatial features on or off
hist_feat = True # Histogram features on or off
hog_feat = True # HOG features on or off
```

For example, HSV color space (channel S, or 1) with 32 orientations produced the below HOG features.



Another example: LUV color space (channel L, or 0) with 16 pixels per cell and 9 orientations produced the below HOG features.

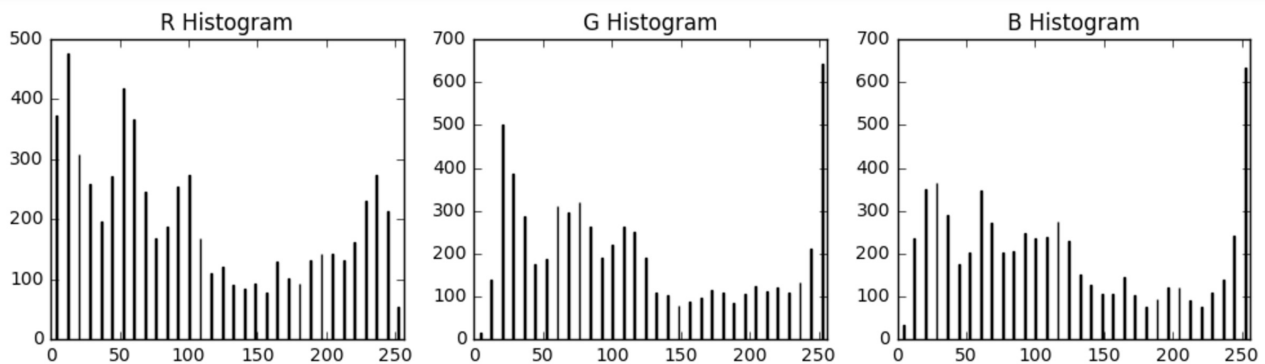


After many manual combinations of the parameters, the final set was fixed as follows: {RGB color space, ALL channels, 9 orientations, 8 pixels per cell, 2 cells per block}.

Extraction of HOG features form the feature vectors to be used for classifier training.

2. COLOR HISTOGRAMS AND SPATIAL BINNING AS FEATURE DESCRIPTORS

Histograms of colors (the constituent channels, for e.g., Red, Green and Blue in RGB) can also act as features. As they affect performance of the classifier, they have been included in the pipeline.



The spatial binning feature, which is simply upsizing or downsizing an image from the image in the dataset, each of which is 64 x 64 pixels (RGB), did not yield a consistently good result during program execution, and hence was dropped. For example, setting the binning size to (32,32) would downsize by a factor of $\frac{1}{2}$ to 32 x 32 pixels.



So, color histograms only are included as features in this pipeline along with HOG. Example images above are shown from the Udacity course material.

A MORE PROPER APPROACH

A more “scientific” approach was adopted to ferret out the best set of parameters. Time constraints, however, ruled against finishing that line of inquiry.

```
def generate_feature_sets():
    #Setup for best parameter set search to feed the SVM training
    color_space_range = ['RGB', 'HSV', 'LUV', 'HLS', 'YUV', 'YCrCb'] # 6 color spaces
    orient_space = list(range(6,10)) # range is 6,7,8,9
    pix_per_cell_range = list(range(6,9)) # range is 6,7,8
    hog_channel_range = ['0', '1', '2', 'ALL']
    hist_bin_range = [16, 32]

    #Establish the total # of combinations to sift through
    iter_count=len(color_space_range)*len(orient_space)*len(pix_per_cell_range)*len(hog_channel_range)*len(hist_bin_range)
    featureset_list = [dict() for x in range(iter_count)] #Define a list or an array of dictionaries to store the feature set run to gen car_features etc.

    import pickle
    import time

    iter_count = 0 #Reset counter to count again
    for color_space in color_space_range:
        t1 = time.time()
        print ('Generating featureset ', iter_count, 'at:', t1)
        for orient in orient_space:
            for pix_per_cell in pix_per_cell_range:
                for hog_channel in hog_channel_range:
                    for hist_bins in hist_bin_range:
                        print ('Color_Space:', color_space, ' Orientation:', orient, ' Pix /cell:',
                              pix_per_cell, ' HOG Chnl #:', hog_channel, ' # of Hist Bins:', hist_bins)
                        featureset_list[iter_count] = [{'Color Space': color_space, 'Orientation':
                              orient, 'Pix_per_cell': pix_per_cell, 'HOG_Channel #': hog_channel,
                              'Num_of_Hist_Bins': hist_bins}]
                        car_features = extract_features(cars, color_space=color_space,
                              spatial_size=spatial_size, hist_bins=hist_bins,
                              orient=orient, pix_per_cell=pix_per_cell, cell_per_block=cell_per_block,
                              hog_channel=hog_channel, spatial_feat=spatial_feat, hist_feat=hist_feat,
                              hog_feat=hog_feat)

                        notcar_features = extract_features(notcars, color_space=color_space,
                              spatial_size=spatial_size, hist_bins=hist_bins,
                              orient=orient, pix_per_cell=pix_per_cell, cell_per_block=cell_per_block,
                              hog_channel=hog_channel, spatial_feat=spatial_feat,
                              hist_feat=hist_feat, hog_feat=hog_feat)
                        fn = 'car_noncar_features'+str(iter_count)+'.p'

                        with open(fn, 'wb') as picklefileh:
                            pickle.dump((featureset_list[iter_count], car_features, notcar_features),
                                    picklefileh, protocol=pickle.HIGHEST_PROTOCOL)
                        print ('Finished in ', time.time()-t1, 'sec.')
                        iter_count+=1

    print (featureset_list[0], '\n', featureset_list[101] )
    print ('Total # of iterations for feature preparation: ', iter_count)
```

The above developed functional code (not part of [1]) generated 576 feature sets all stored into pickled dictionaries (~50 GB), awaiting a plan to continue this investigation in short order! (Datasets can be shared, if needed on Google Drive).

The plan was to either do my own GridSearchCV type of function or randomly sample across the featuresets. Then, the best set would be the one that offered the highest validation accuracy during training of the SVM classifier with its own GridSearchCV().

As mentioned above, further investigation was stopped pending availability of more powerful hardware and time!

3. SUPPORT VECTOR MACHINE (SVM) CLASSIFIER TRAINING

A support vector machine learning model is a versatile one for linear and nonlinear classification of complex but small to medium sized datasets, used here to classify car images from noncar images.

This classifier too comes with its bag of parameters, whose best combination for a given dataset, fortunately, can be found by employing the very useful function GridSearchCV () found in the sklearn library.

The parameters that can be tuned are:

Value of C, gamma and choice of kernel, whether RBF or Linear.

```
'''GRID SEARCH OPTION WAS DONE ONCE (This code is not part of [1])

parameters = {'kernel':('linear', 'rbf'), 'C':[0.1, 1, 10], 'gamma':[0.01, 0.1, 1]}
svr = svm.LinearSVC()
t=time.time()
grid_search = GridSearchCV(svr, parameters)
t2 = time.time()
print(round(t2-t, 2), 'Seconds to GridSearch SVC...')
grid_search.fit(X_train, y_train)
t3 = time.time()
print(round(t3-t2, 2), 'Seconds to GridSearch SVC...')
print ('\nBEST PARAMS', grid_search.best_params_)
'''
```

BEST PARAMETERS: {Kernel: Linear, C=0.1, Gamma: 0.01} (Took 557 secs.)

GridSearchCV (), so employed to find the best parameter combination for the image datasets, yielded C=0.1 with a Linear kernel.

Next, the features data was split 80/20% for training and testing, and normalized and shuffled.

The SVM classifier was trained using the above parameters. The code is as shown below.

```
# Using a linear SVC (Code starts at line 640 in [1])
svc = LinearSVC(C=0.1)

# Check the training time for the SVC
t=time.time()
svc.fit(X_train, y_train)
t2 = time.time()

print(round(t2-t, 2), 'Seconds to train SVC...')

# Check the score of the SVC
print('Test Accuracy of SVC = ', round(svc.score(X_test, y_test), 4))
```

With the previously noted feature parameter combination (RGB, ALL channels etc.), the classifier was tried on different subsets of data. Based on the future classifier performance of the SVM, the test dataset that was finalized was the one with approx. 1200 images, with an accuracy of ~97%. See screenshots below.

Number of car images: 5966
Number of non-car images: 8968
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 8412
74.0 Seconds to train SVC...
Test Accuracy of SVC = 0.9977

Due to some reported (unconfirmed by the student and beyond the scope of this project) aberration in "intercept_scaling" in LinearSVC(), the standard SVC() function was used to train the model (the training time, 74 sec, is a lot more than SVC()'s training time, shown below).

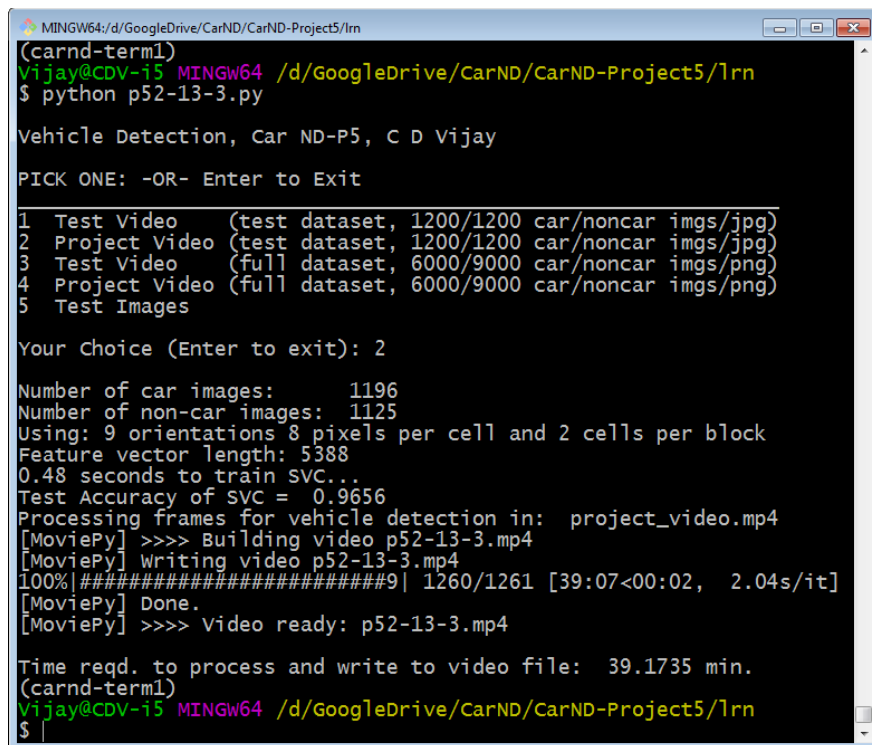
Number of car images: 5966
Number of non-car images: 8968
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 8412
22.31 Seconds to train SVC...
Test Accuracy of SVC = 0.9977

Due to nary a difference between the two in classifier accuracy, the latter one was chosen for this project due to its faster classification time.

With some other changes as noted below (using SVC()):

Number of car images: 5966
Number of non-car images: 8968
Using: 6 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 6696
11.03 Seconds to train SVC...
Test Accuracy of SVC = 0.9987

Example Run



```
(carnd-term1)
Vijay@CDV-i5 MINGW64 /d/GoogleDrive/CarND/CarND-Project5/1rn
$ python p52-13-3.py

Vehicle Detection, Car ND-P5, C D Vijay

PICK ONE: -OR- Enter to Exit

1 Test Video (test dataset, 1200/1200 car/noncar imgs/jpg)
2 Project Video (test dataset, 1200/1200 car/noncar imgs/jpg)
3 Test Video (full dataset, 6000/9000 car/noncar imgs/png)
4 Project Video (full dataset, 6000/9000 car/noncar imgs/png)
5 Test Images

Your Choice (Enter to exit): 2

Number of car images: 1196
Number of non-car images: 1125
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 5388
0.48 seconds to train SVC...
Test Accuracy of SVC = 0.9656
Processing frames for vehicle detection in: project_video.mp4
[MoviePy] >>>> Building video p52-13-3.mp4
[MoviePy] Writing video p52-13-3.mp4
100%|#####| 1260/1261 [39:07<00:02, 2.04s/it]
[MoviePy] Done.
[MoviePy] >>>> Video ready: p52-13-3.mp4

Time reqd. to process and write to video file: 39.1735 min.
(carnd-term1)
Vijay@CDV-i5 MINGW64 /d/GoogleDrive/CarND/CarND-Project5/1rn
$
```

4. DETECTION (SLIDING WINDOW TECHNIQUE) & PREDICTION BY SVM CLASSIFIER

The technique is to divide the “to-be-searched” image into fine or coarse windows as the detections show. The bigger image (shown next) is divided into say (m x m) chunks starting from a given point, which has been chosen to be (0,400) in pixels, about the vertical midpoint so that the extraneous information (skyline, trees in the foreground etc.) can be filtered out.

The above process yields a specific number of windows in our view zone (from 0,400) downwards. Now, the tasks are to:

1. Scale this window down to the original dataset image size (in this case, m x m down to 64x64 pix).
2. Extract the feature vector for this smaller (or sub)image (HOG, color histogram and spatial as selected), and have the trained SVM classifier prove its mettle on it (ie, predict).
3. If a car is detected, then note this window (add to a list of “hot’ windows). The windows so detected are now considered to have car objects in them (or not, ergo, called a false positive).

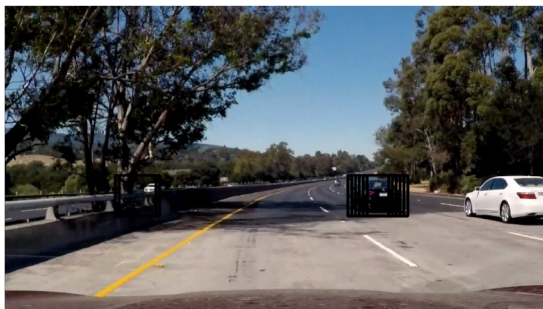
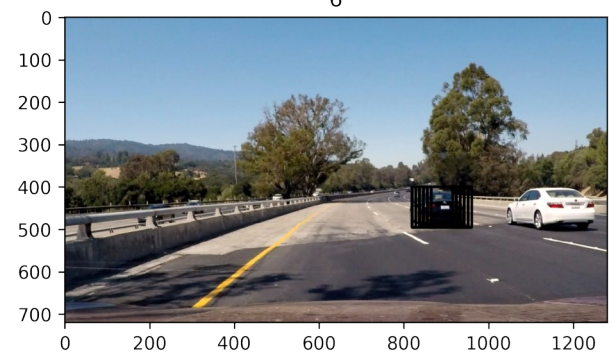
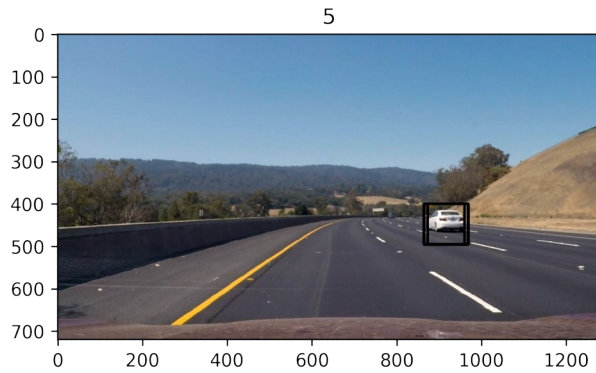
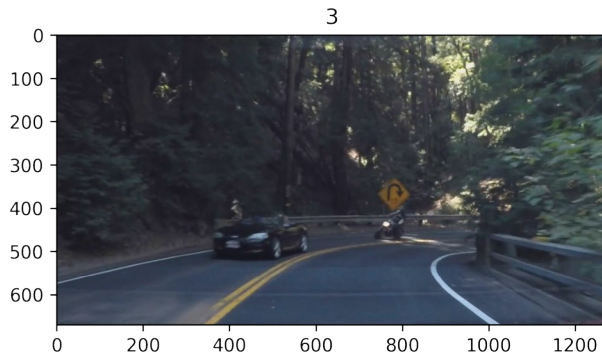
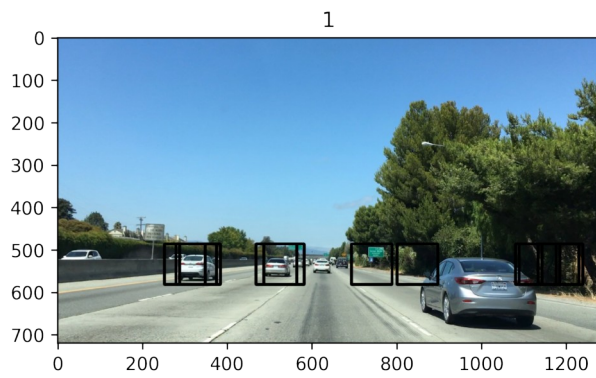
This algorithm too brings its bag of parameters. So, after much tweaking, two parameter sets were chosen (based on the test images). Figures are shown later in this section.

Functions `slide_windows()` and `search_windows()` are used to extract feature vectors as described above, for testing on test images.

Code: `slide_windows (), [1], line 163`
`search_windows(), [1], line 274`

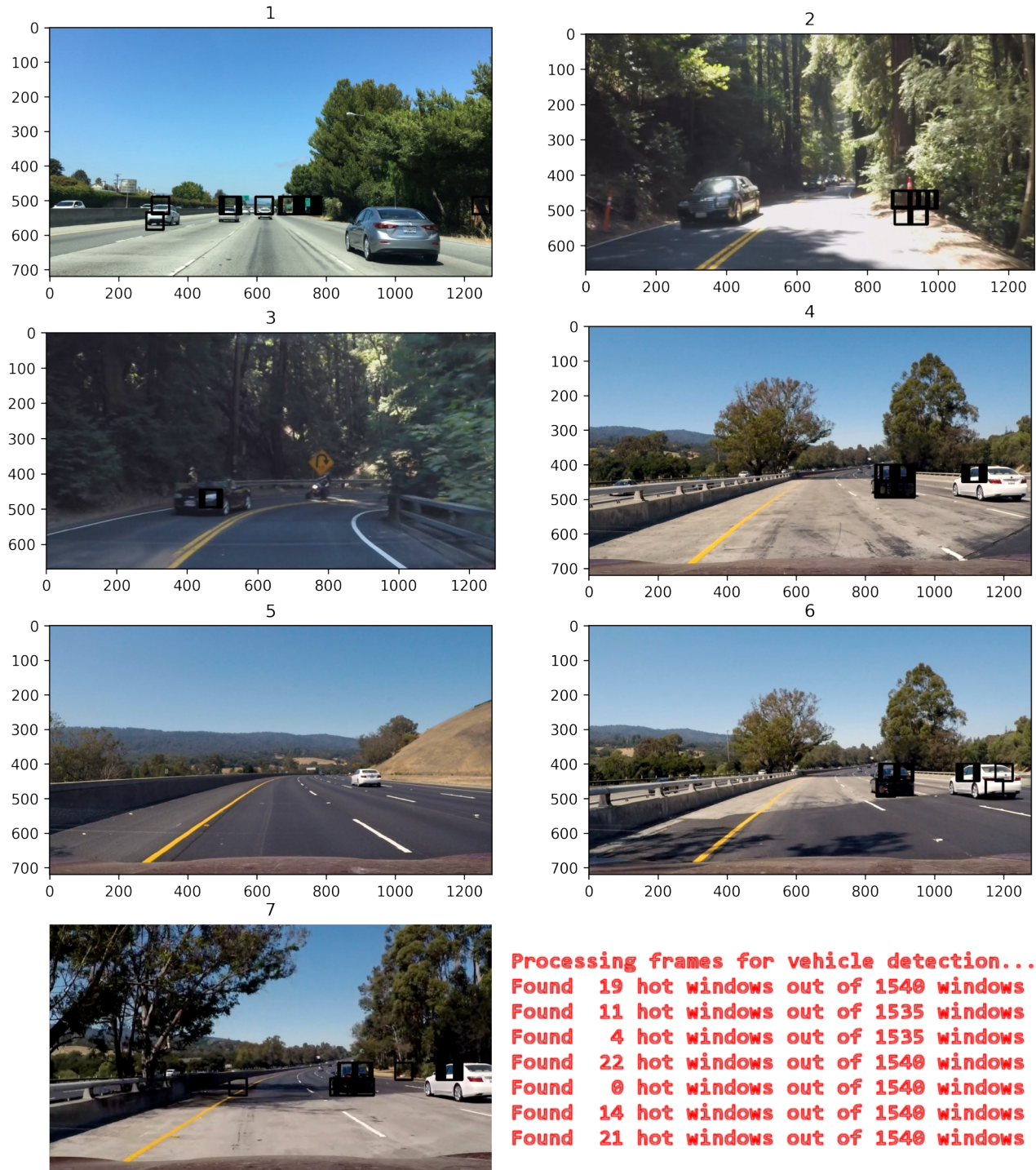
EXPERIMENTS

{ Subwindow size: 96 x 96 pix, (X/Y overlap) = (0.9/0.1) } setting yielded this:
34 ‘hot’ windows, see next image.



Processing frames for vehicle detection...
 Found 11 hot windows out of 264 windows
 Found 1 hot windows out of 264 windows
 Found 0 hot windows out of 264 windows
 Found 6 hot windows out of 264 windows
 Found 2 hot windows out of 264 windows
 Found 6 hot windows out of 264 windows
 Found 8 hot windows out of 264 windows

{ Subwindow size: 50 x 50 pix, (X/Y overlap) = (0.9/0.1) } setting yielded this:
91 hot windows



Note that this subimage size (50x50 pix) runs counterintuitive to the recommended integer multiple of the original (trained) image size (64 x 64), and produced more windows than the earlier setting including detecting a car in the image from P4's challenge video (see image 3).

Automated parameter optimization is recommended here as the next investigation step.

As there was no clear outcome, both settings were run independently and the window lists concatenated thus —

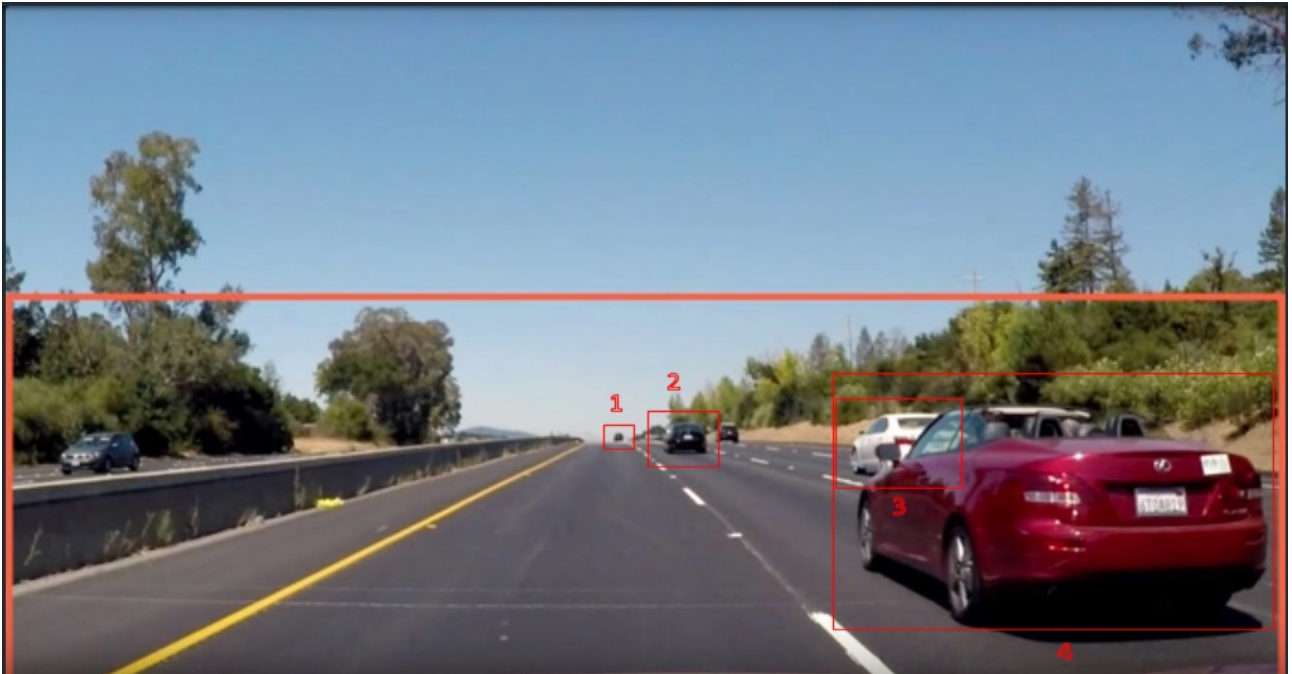
windows = windows1 + windows 2; which produced the following:



This produced 130 hot windows in total including several false positives; but the outcome is acceptable from looking at the quality of detections (and the low number of FPs).

SEARCH WINDOW SCALING

Using different scales to size the search windows up or down (as required) to be able to detect cars far away and the ones close to the camera can be a useful task.

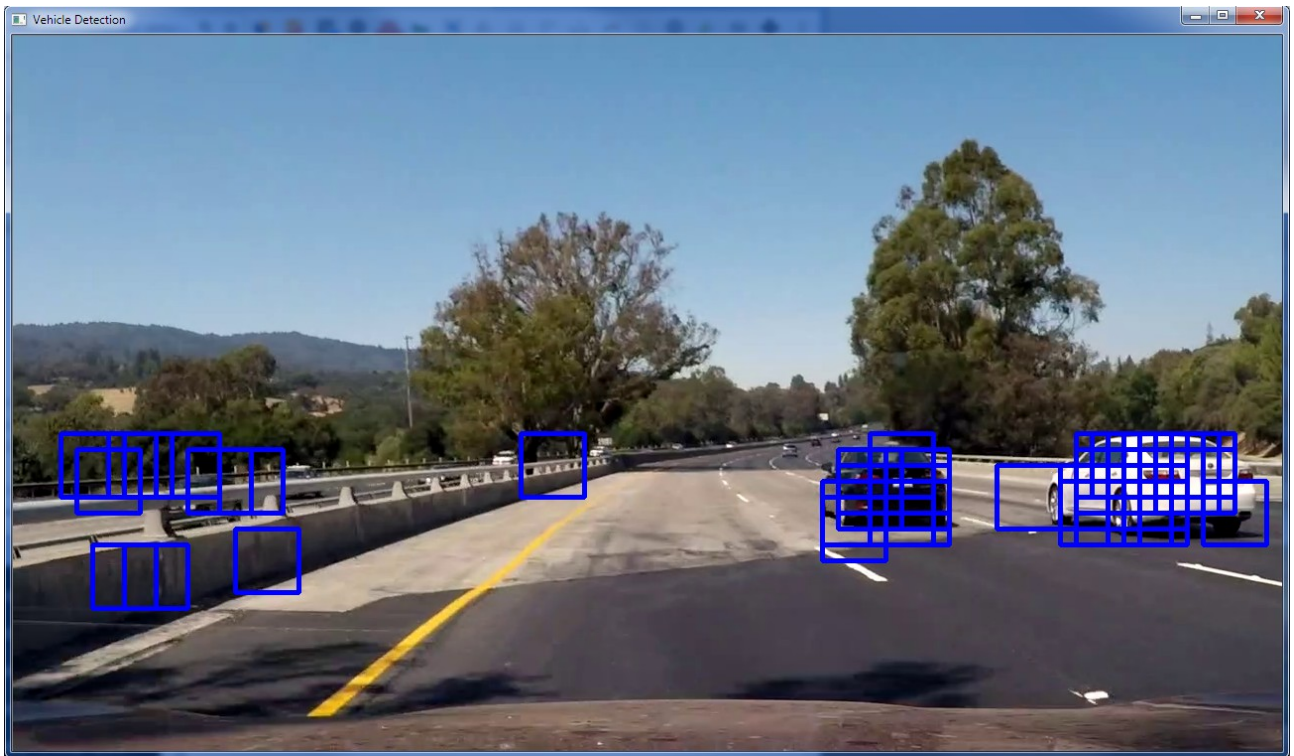


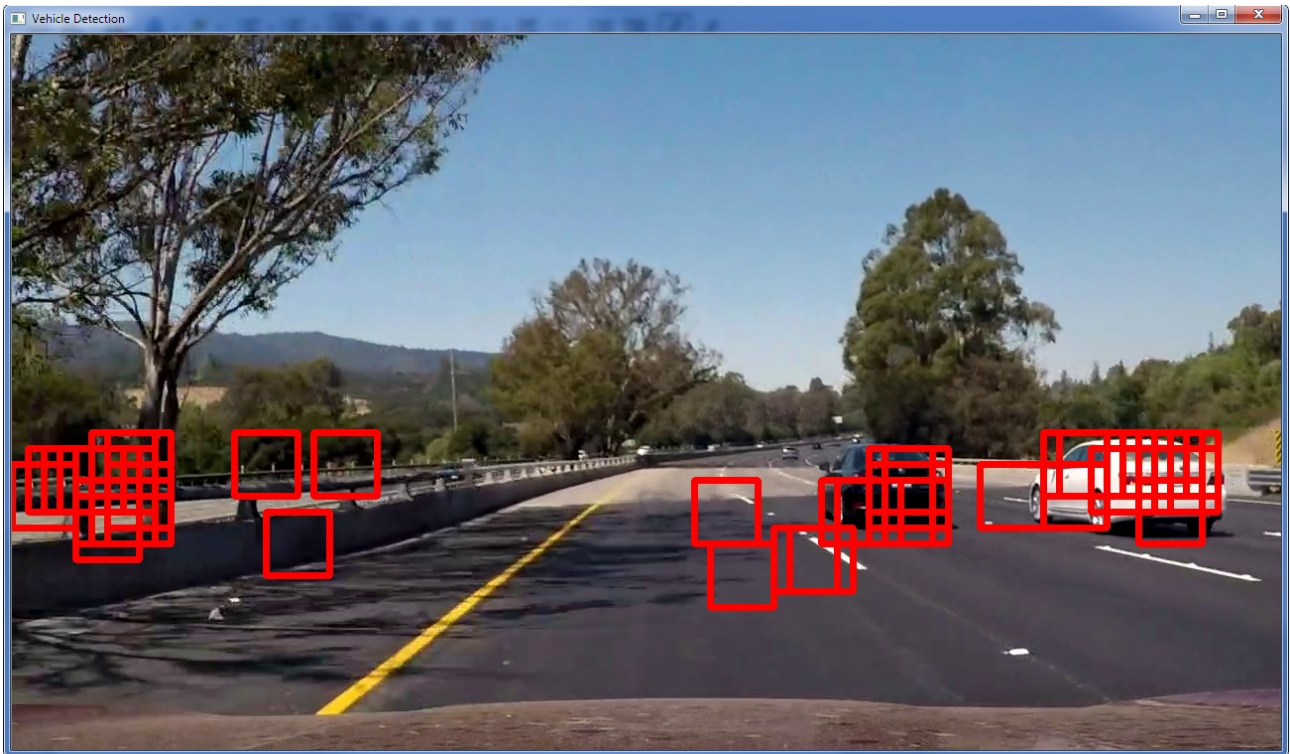
The algorithm shown below was employed to obtain as many detections as possible.

The scales chosen were: {1.0, 1.5, 2.0, 2.5, 3.0}

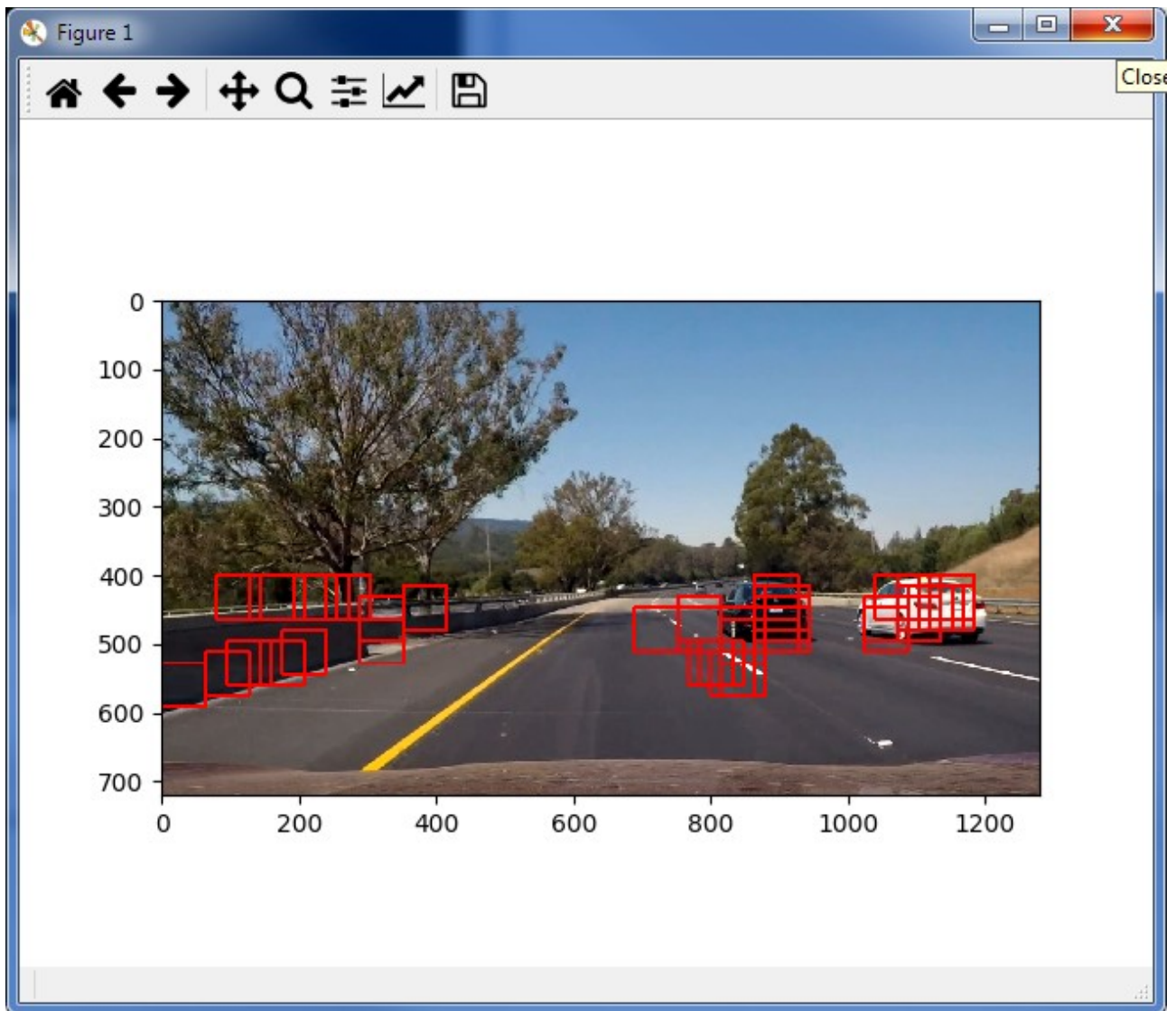
```
#Below Not used, ystart and ystop set to (400,660)
ystart_stop = ([[y1, y2], [y3, y4], [y5, y6], [y7, y8]])

for ystart, ystop in ystart_stop:
    for scale in linspace(1, scale_max, 0.25):
        find_cars(img, scale)
        #Detect false positives
        #Draw video / display image
```



Based on these experiments with scaling, a similar scaling did not perform as well when applied to detecting cars in video frames. So, for the video processing, a scale of 1 is used.



As the images above show, this detection technique detects cars but produces many false positives (FPs). The next section details a technique employed to try removing them.

5. ELIMINATING FALSE POSITIVES (FP)

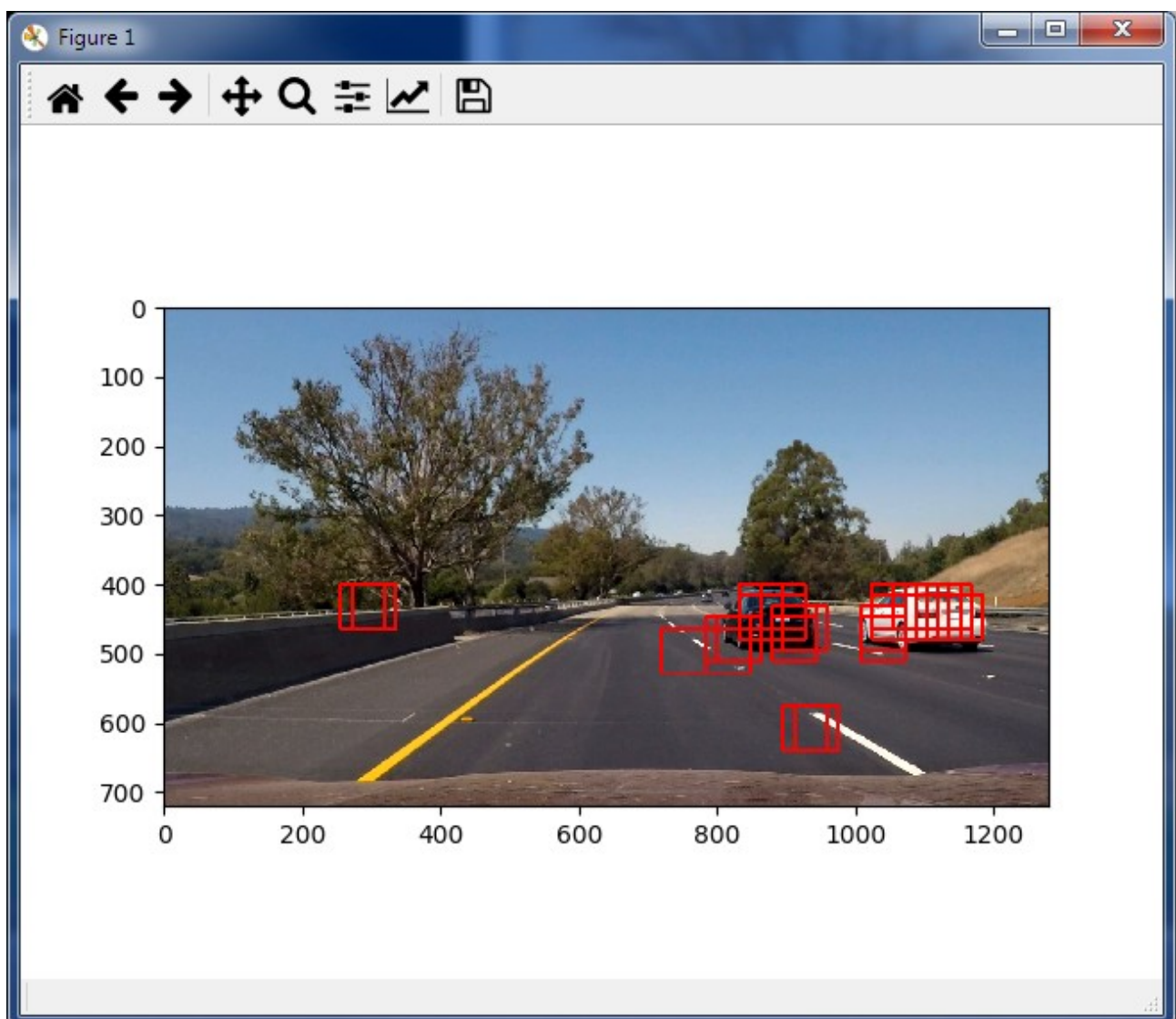
One technique to eliminate false positives is by making a heatmap – a method in which each “hot” window (i.e., a window in which a car is detected) is marked (given added heat) in every frame. Over the course of several windows, the cars if truly present in a frame will get multiple detections thus making that zone “hot”. False positives tend to be isolated cases (with low heat) whereas real car detections tend to cluster around the same area (with high heat) as processing progresses over the frames.

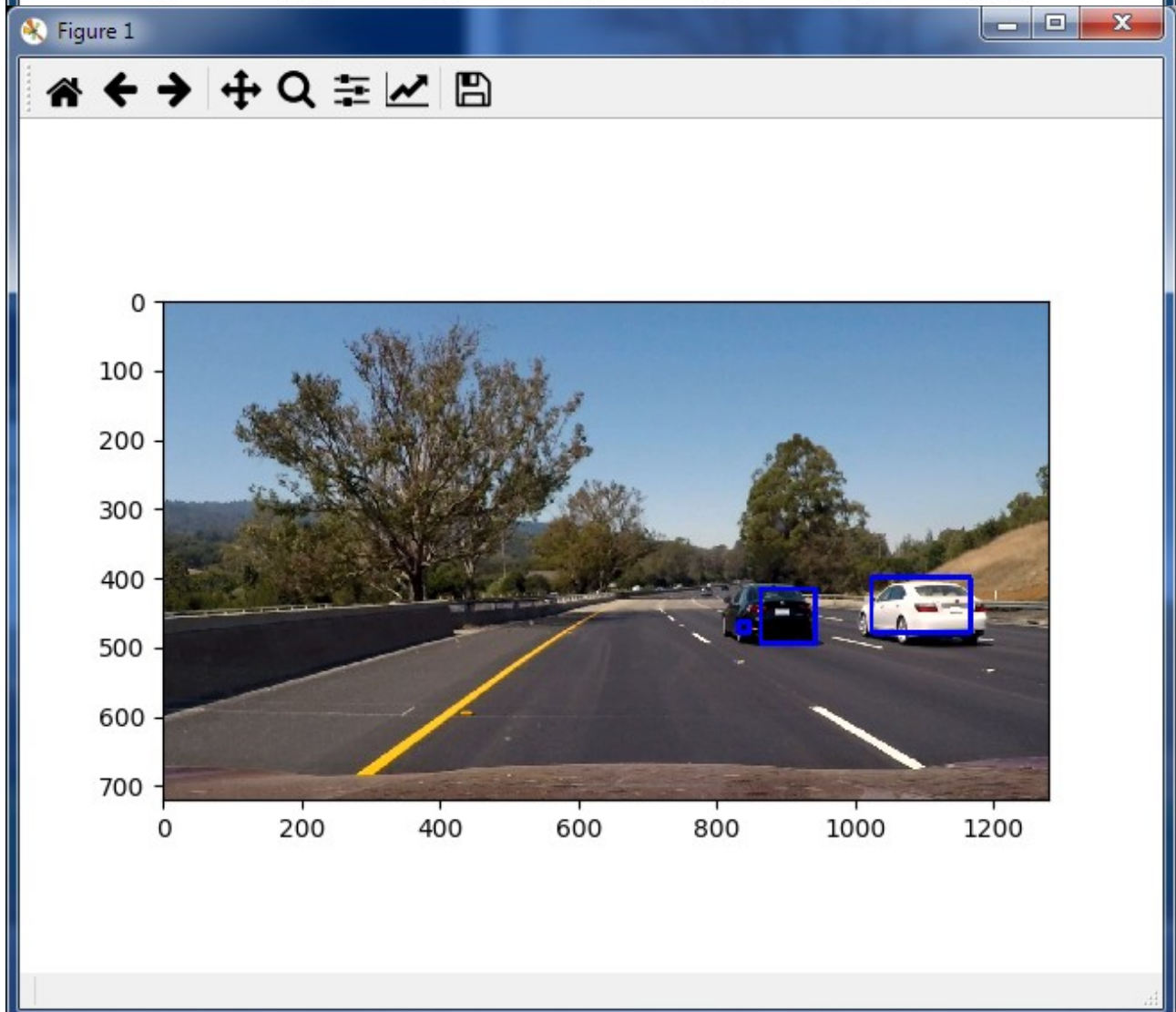
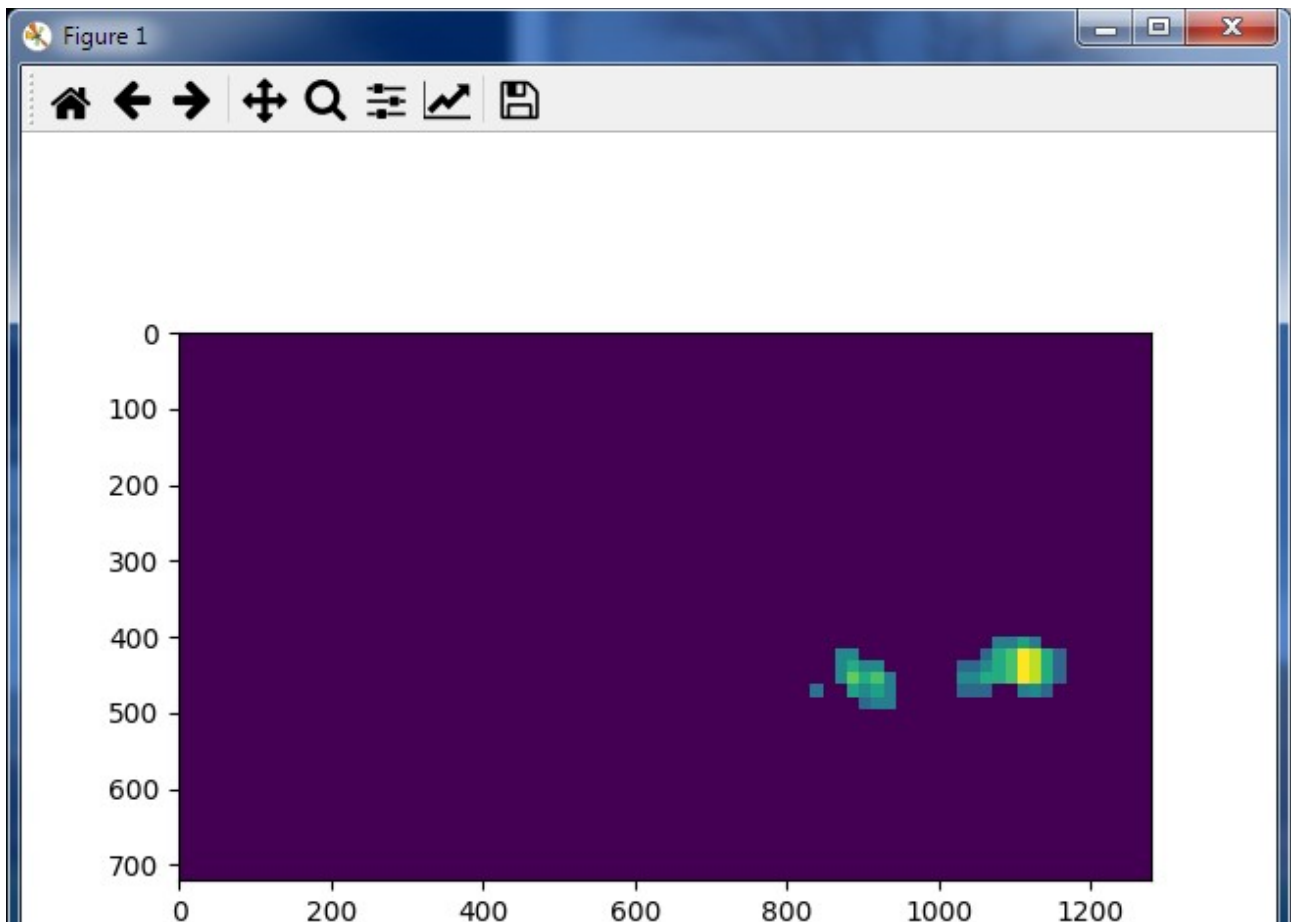
Heatmaps are tracked over many frames (20) and the heat value averaged before a threshold is applied (see class `TrackHeat` definition at the beginning of [1]). Then, a threshold (a limit) is used to filter out low “heat” values (false positives) while retaining “hot” zones, in essence, removing false positives. The `numpy.clip()` function is used to simplify the heatmap.

Employing the very useful `label()` function (“connected component modeling”) from `scipy.ndimage.measurements`, multiple contiguous contours in the heatmap are labeled or bound. These are the detected car positions.

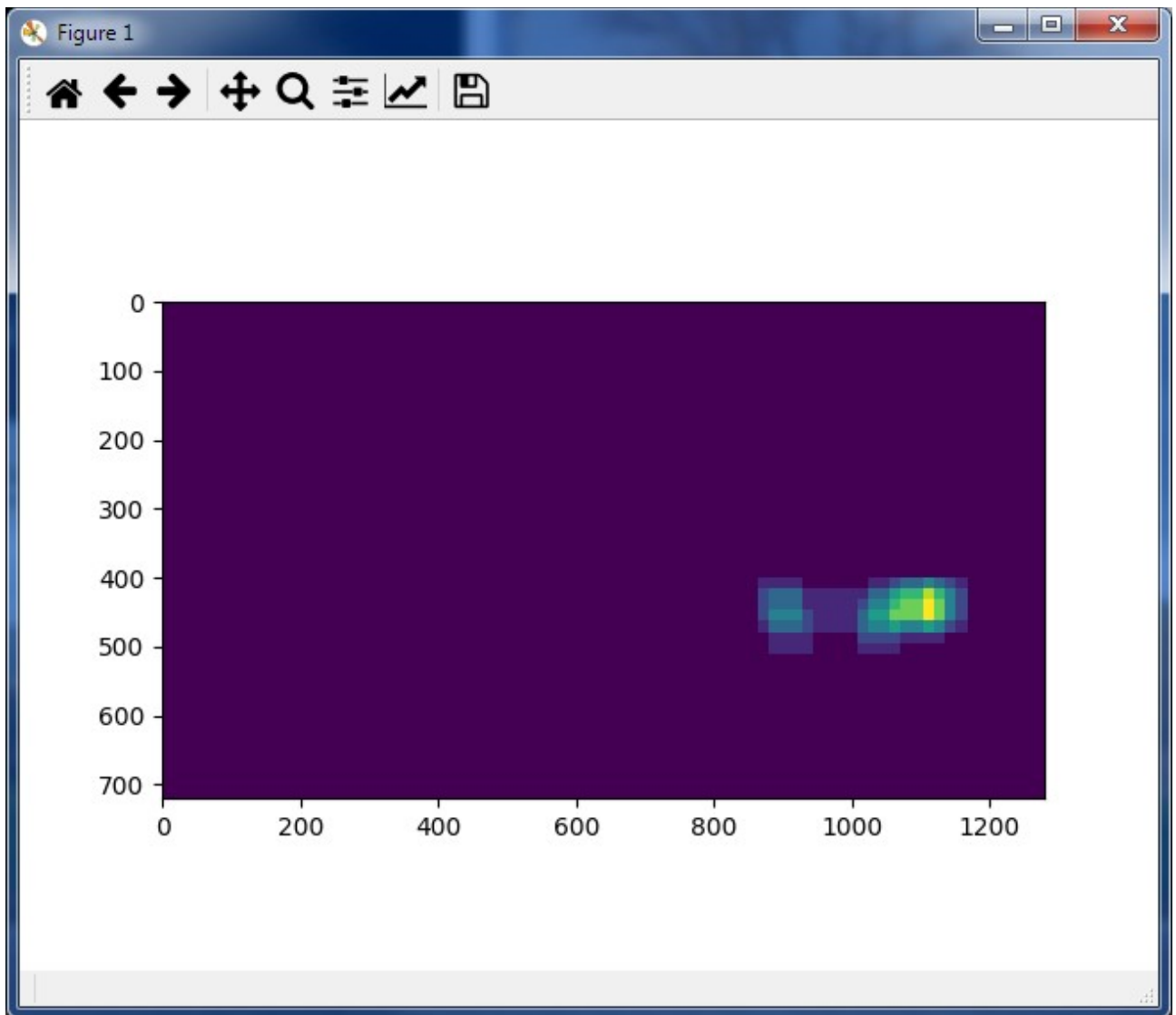
Finally, a composite image comprised of these bound boxes superimposed on the original image is created to show the areas where cars were detected.

A separate function, `Find_cars()` ([1], line 336) is the heart of the pipeline. The action of this pipeline is shown via images next.











A screenshot of the final composite image is shown next from running [1] on the test_video.mp4 video file.

6. PIPELINE

The final output video is generated using the fine moviepy.editor module from Zulko. The ProcessImage() function (l, line 495) runs the pipeline on every frame in project_video.mp4 file.

```
#Will return a single image with labeled boxes drawn on it
def ProcessImage (img):

    img1 = np.copy(img)

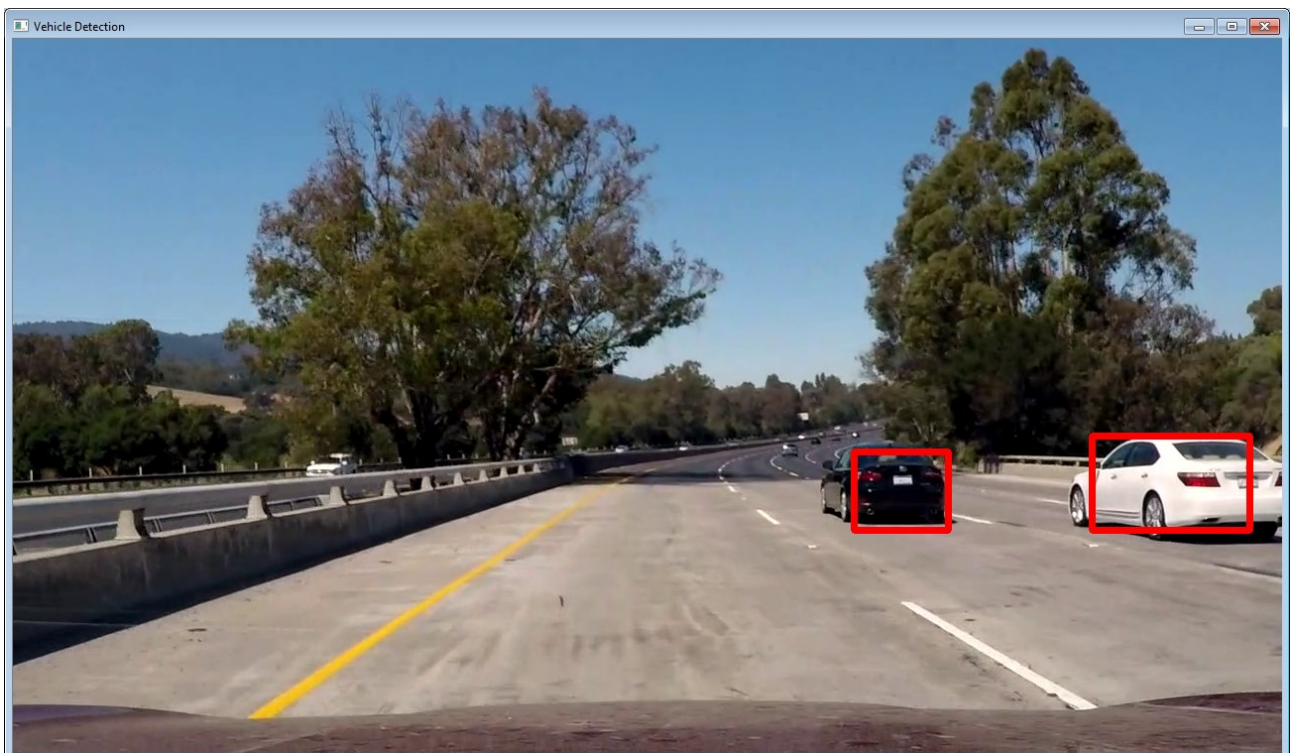
    for s in scales:
        boxes, hmap = find_cars(img, ystart, ystop, scale, svc, X_scaler, orient, pix_per_cell,
                                cell_per_block, spatial_size, hist_bins, spatial_feat, hist_feat, hog_feat)

    track_heat.append(hmap)
    if len (track_heat.hmap_list) > numframes:
        track_heat.hmap_list = track_heat.hmap_list[1:numframes+1] # drop the first one, retain the
                                                                    others

    array_trackHeat = np.array(track_heat.hmap_list)

    heat_avg = np.mean(array_trackHeat, axis=0)
    map = apply_threshold(heat_avg, heat_threshold)
    hmap = np.clip (map, 0, 255)
    labels = label (hmap)

    draw_img = draw_labeled_boxes(img1, labels)
    return draw_img
```



Link to: [Youtube \(project_video.mp4\)](#)

7. DISCUSSION

Several things are noteworthy here.

1. This algorithm is more proof-of-concept than an industry-grade one. Of course, any code can be optimized but the concept does not lend itself to real-time operation, unless more powerful hardware becomes affordable and prevalent for real-time use. The fact that one needs to use `moviepy.editor` to generate a composite video after processing stands testimony to the fact that this technique needs streamlining.
2. With a lot more data (SVM may not be the classifier of choice then), detections will become more accurate.
3. Due to the programmatic nature of the code, again as in P4, I am intrigued with the possibility of using deep learning for this task like several latest firms are demonstrating.
4. My code here is far from perfect. More work, when time permits, is required in the area of solidifying false positives elimination, and experimenting with different classifiers, and code optimization.
5. Due to the vast array of parameters, some degree of optimization work must be performed to gain classification confidence and improve speed.

REFERENCES

- [1]. Source code file, p5-final.py, submitted with this project
- [2]. Udacity CarND Term 1, Course material
- [3]. Aurélien Géron, Hands-on Machine Learning with Scikit-Learn & Tensorflow, O'Reilly: Mar 2017.