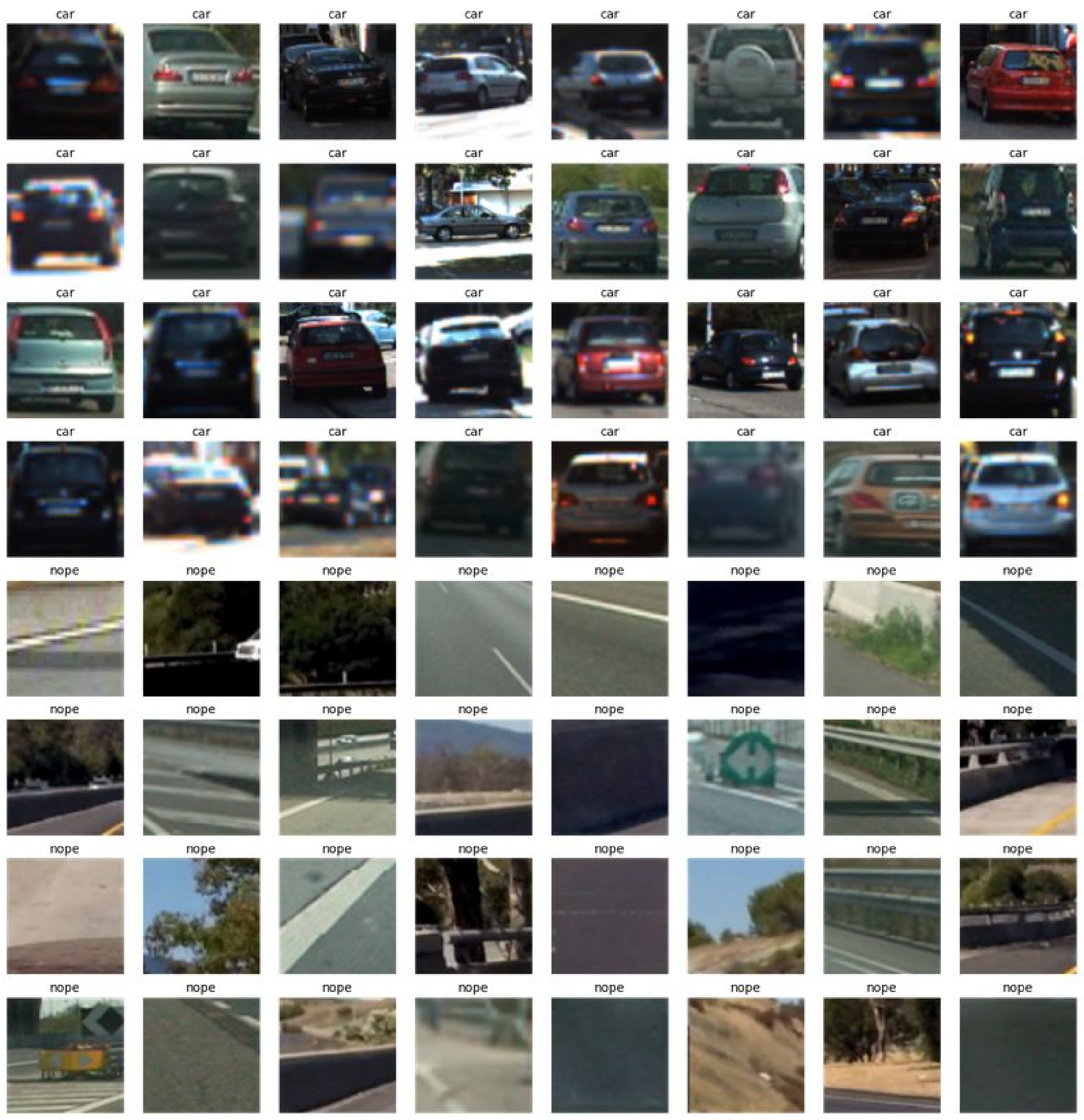
Vehicle Detection project:

1 Data exploration

The goal of this project is to detect vehicles in a video stream. The core part of this project is to let our designed pipeline to be able to detect vehicles with sufficient accuracy and minimize false positve at the same time.

Thus, we use machine learning approach to be able to identify cars. To help us solve the problem, we are given more than 18 thousands 64x64 format images including both vehicle images and non-vehicle images for training purpose. Below is some examples:



Some vehicle images are almost identical and some vehicle images are been taken from different perspectives/angles/lightness condition, etc.

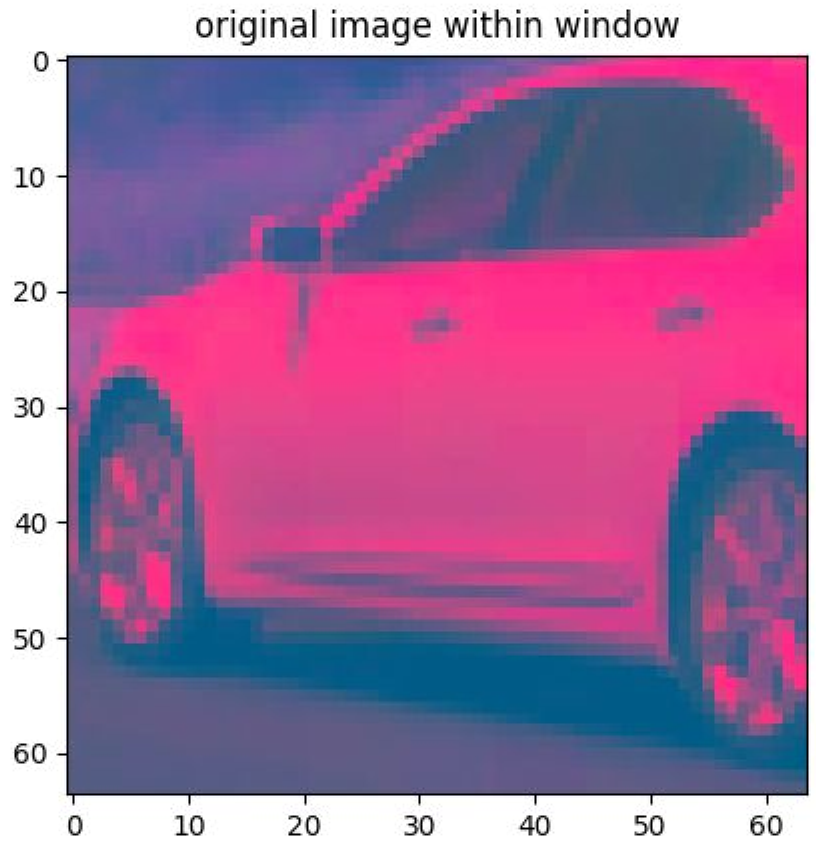
2 Detection training

In this project, we use linear SVM as our choice of classifier for training, thus the selection of feeding features is most important to our results.

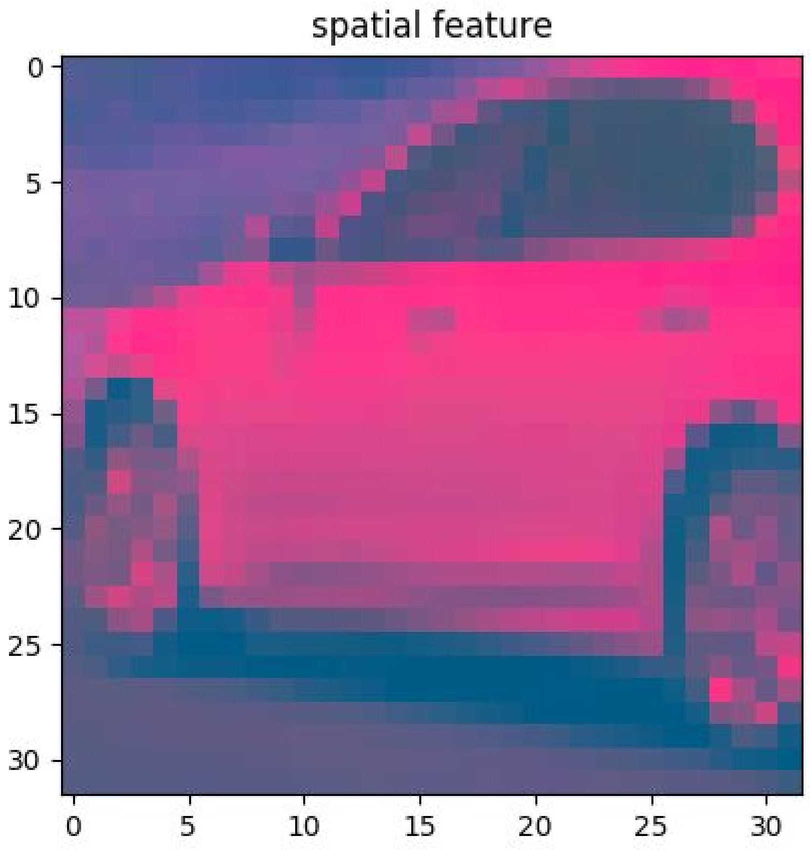
1) Pattern recognition

Correctly identify patterns/features of vehicles is vital to the accuracy of detection and also is the core part thing to be done in most classification problems. There are various aspects that we can start with, like color, shapes, sizes and positions in the image, etc.

We take below image as an example to demonstrate this process:

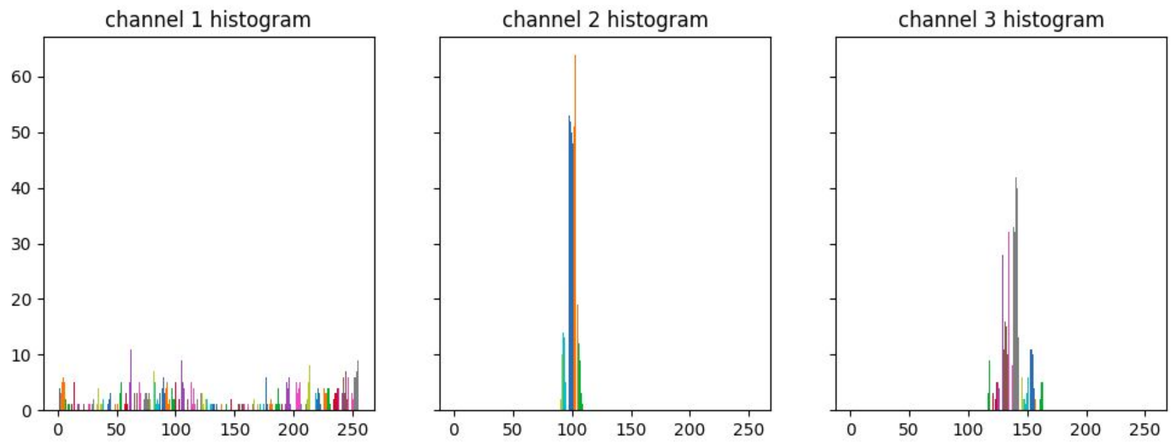


Above is a 64x64 image extracted from an image.
The obvious feature we can see from the image is that vehicle has a more saturated colorness than the surroundings, thus we can extract all pixels values as a feature to feed. Also, we observe that even coarser resolutions can contains enough information like below shows which is a 32x32 image after resizing.



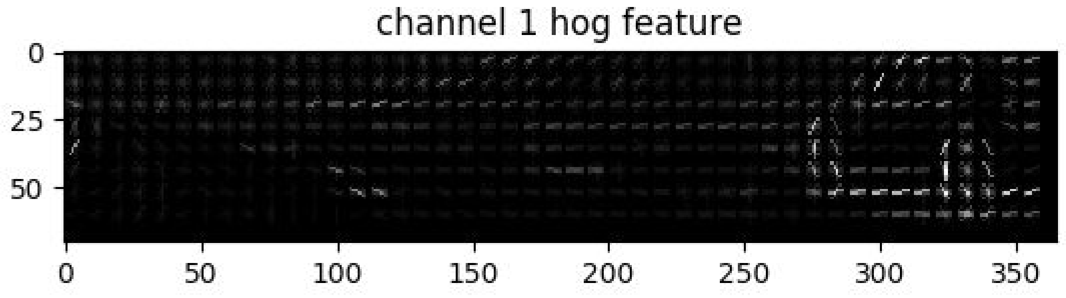
Smaller size of feature can make our training process run faster but without losing any useful information.

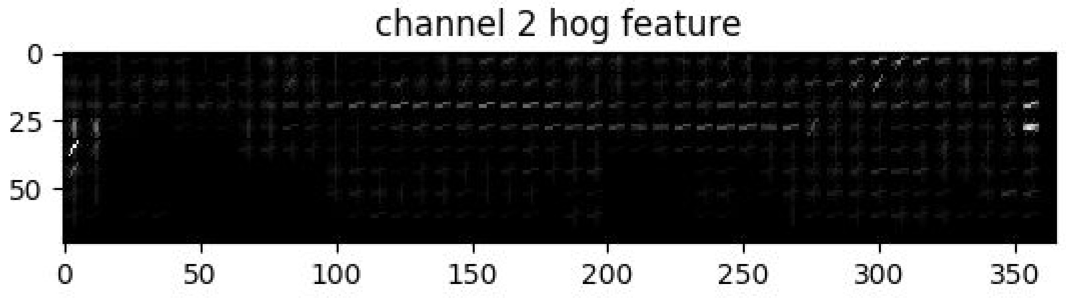
Under diffierent colorspaces, we can also see that some parts of the vehicle have a much higher/lower pixel value than the rest. This is observable by ploting the histogram, below is the histogram in all three channels in LUV colorspace:

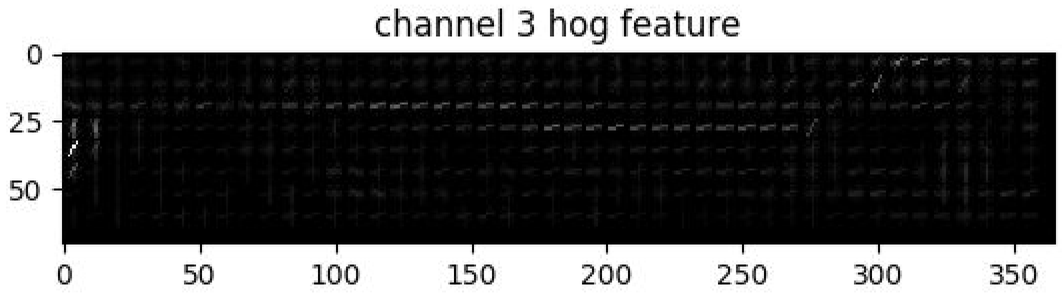


As we can see above, in the second and third channel, there is a significant peak in the distribution of values, which can be another useful features to feed.

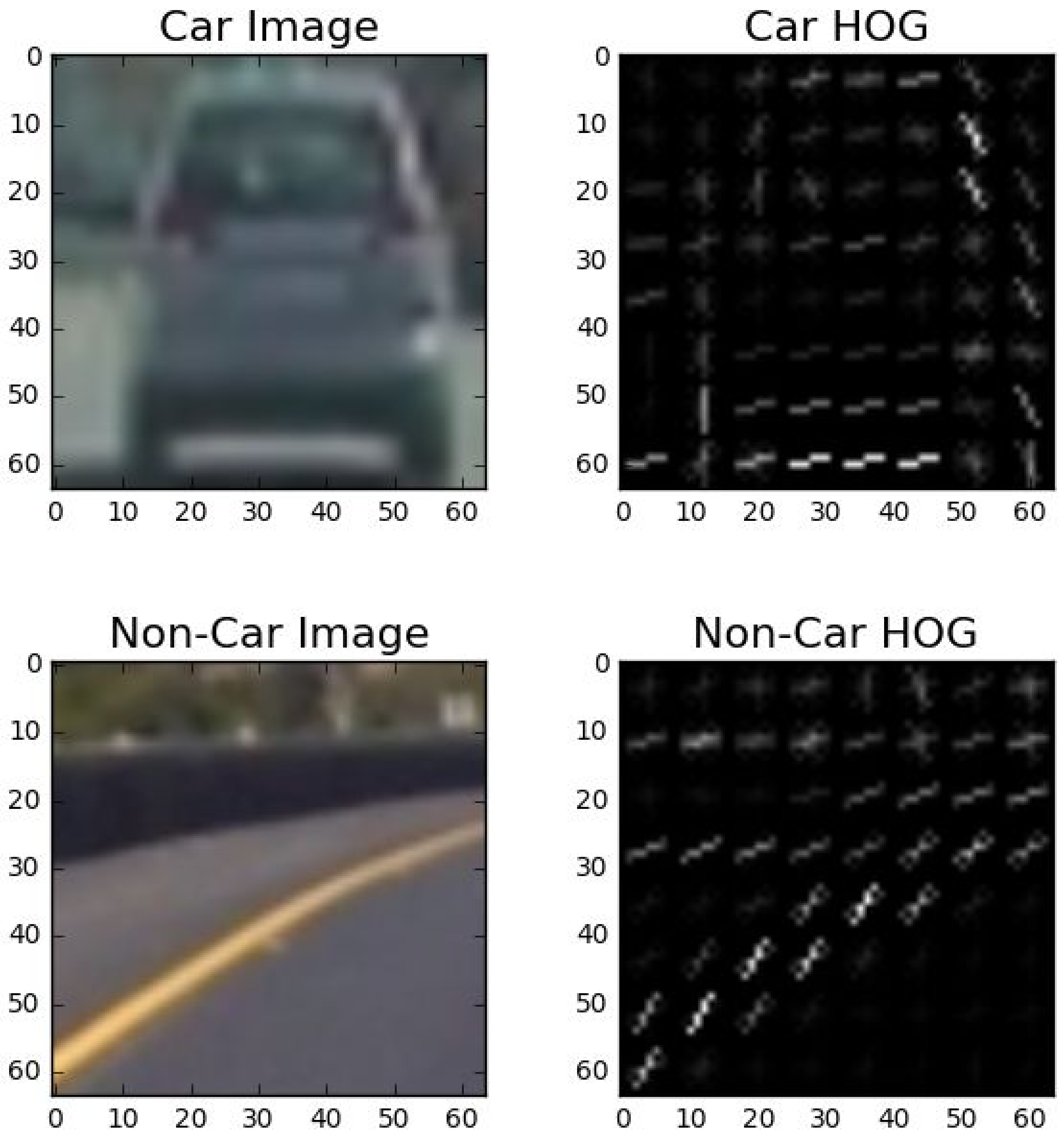
We have explored colorness and their distributions, next we shall look at the shape of the vehicles. The shape of object can be expressed by the results of gradients. Therefore, we can see if the histogram of gradients(HOG) shows any obvious features as well.
Below is the HOG output from the same image in three different channels.







Apparently, the HOG of the first channel perfectly depicts the outline of the vehicle. Normally, HOG is a good representation for the shape and colors of objects.



We can see it makes car image and non-car image above distinguishable by their HOG view.

In summary, we use three aspects as our feeding features, including pixels values in spatial, histogram of colorness and histogram of gradients.

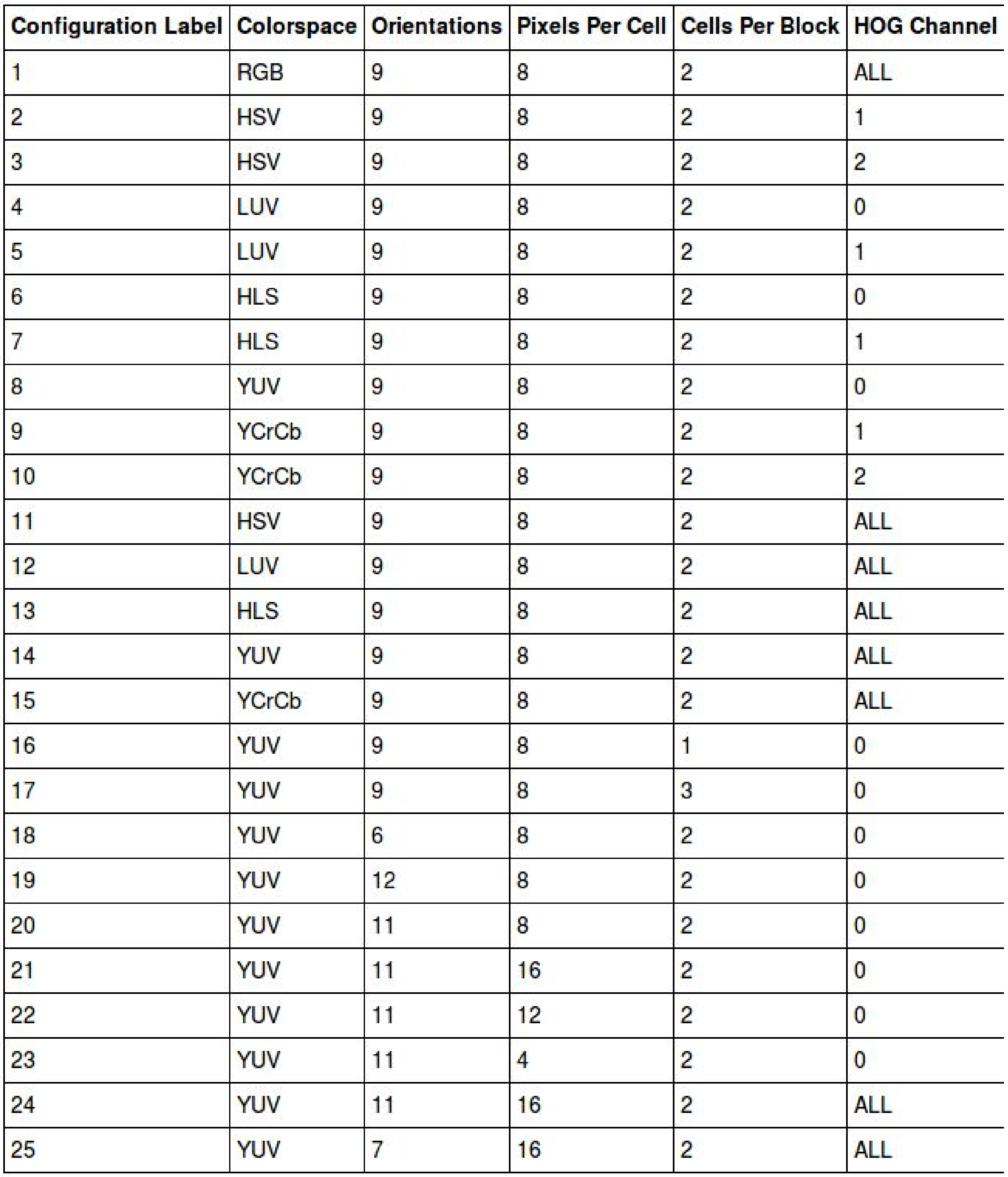
2) Parameter tuning

Since we have successfully identify what features can be used for our training, next problem is to find the best set of parameters that optimize our results.

In the parameter space, we have following parameters to consider:

1) Colorspace
2) Orientations
3) Pixels per cell
4) Cells per block
5) Choice of Hog channel

The 1) parameter is shared by all three features and 2) to 5) are only relates to extracting Hog features. Our approach is to tune the parameters is simply to permutate as many combinations as possible in reasonable consideration. For instance, normally the best choice of 2) is around 9 because more orientations does not do much good to performance and pixels per cell is often 8 in practice, as to the choice of colorspace, RGB is rarely been considered because either LUV or YUV colorspace can contains more colorfulness information.
The possible set of configurations is summaryed in below:



Configuration Label	Colorspace	Orientations	Pixels Per Cell	Cells Per Block	HOG Channel
1	RGB	9	8	2	ALL
2	HSV	9	8	2	1
3	HSV	9	8	2	2
4	LUV	9	8	2	0
5	LUV	9	8	2	1
6	HLS	9	8	2	0
7	HLS	9	8	2	1
8	YUV	9	8	2	0
9	YCrCb	9	8	2	1
10	YCrCb	9	8	2	2
11	HSV	9	8	2	ALL
12	LUV	9	8	2	ALL
13	HLS	9	8	2	ALL
14	YUV	9	8	2	ALL
15	YCrCb	9	8	2	ALL
16	YUV	9	8	1	0
17	YUV	9	8	3	0
18	YUV	6	8	2	0
19	YUV	12	8	2	0
20	YUV	11	8	2	0
21	YUV	11	16	2	0
22	YUV	11	12	2	0
23	YUV	11	4	2	0
24	YUV	11	16	2	ALL
25	YUV	7	16	2	ALL

After comparision, we found that label 12 has the best performance in terms of accuracy and training time. We take configuration of 12 as our choice.

3) Training set and test set

It is worth noticing that there are a lot of group of vehicle images that look alike to each other due to small degree of different perspectives which is ok if there are all in either training set or test set, but it would be problematic if there are mixed in both sets because normally we expect the data in test set is unseen before so that we can estimate the performance of our model accurately. Therefore, to overcome this issue, we manually put those groups of data which contains alike vehicle images in different data set. Such arrangement is only needed to all data in the GTI* vehicle data set, we still randomly split training set and test set in the rest data set.

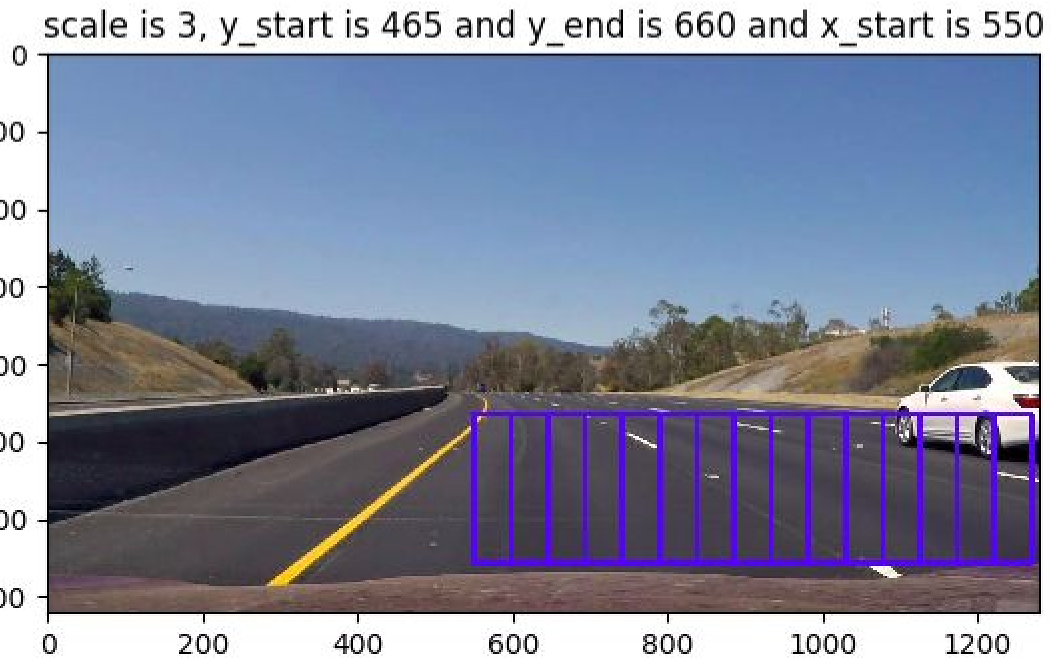
After such arrangement, we see an improved accuracy of performance, up to 98.7%.

3 Searching windows approach

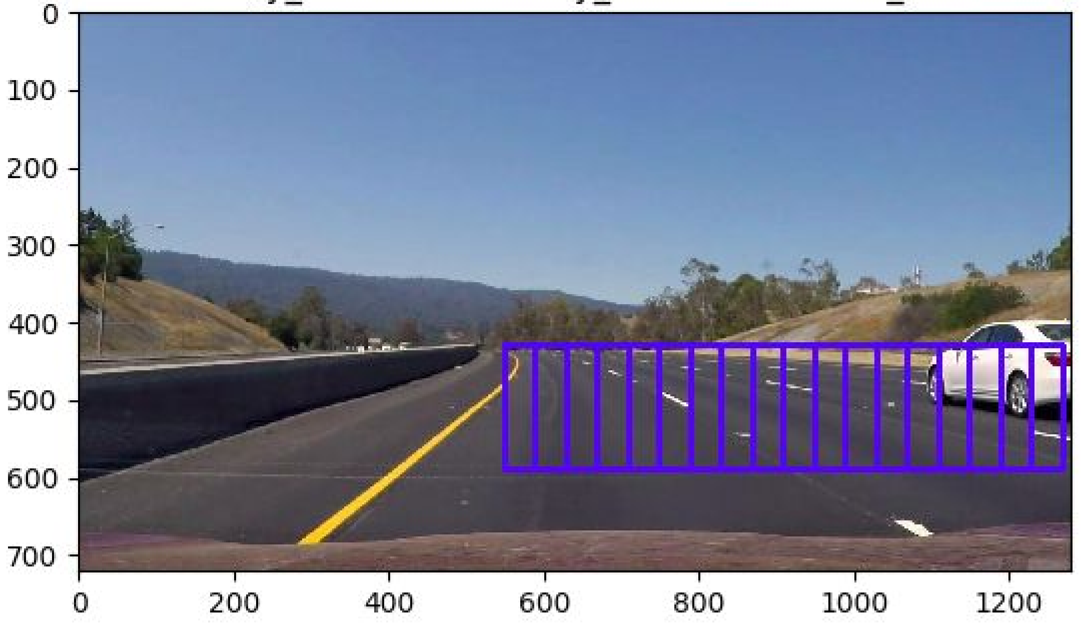
Once we have been able to detect vehicles individually, the next problem is how to locate vehicle positions in an image.

1) Algorithm introduction

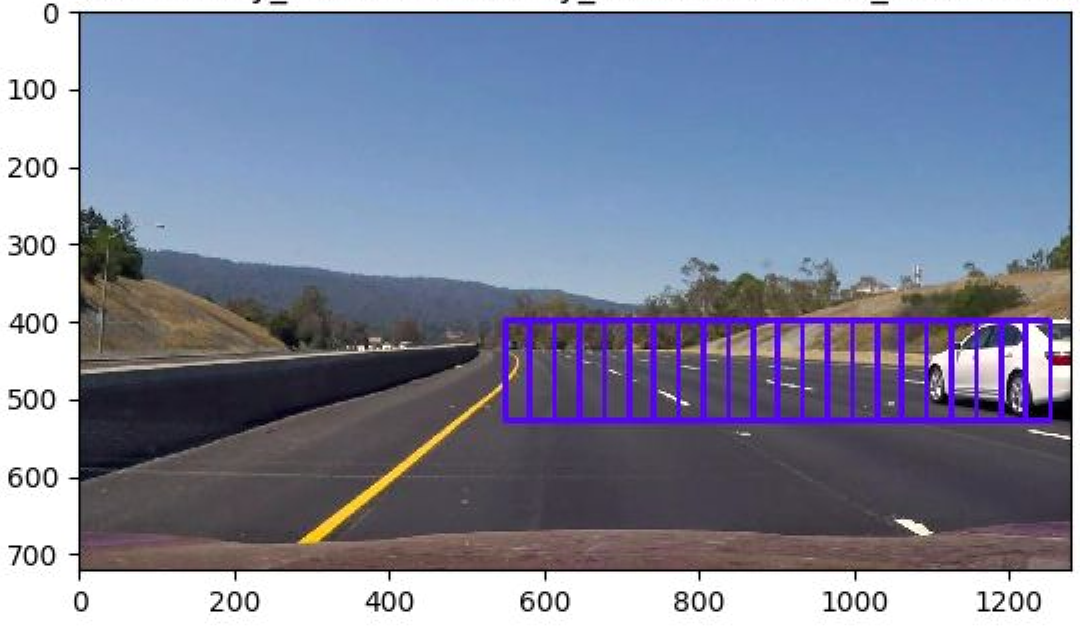
The idea of our approach is to use multiple fixed-size sliding windows in ROI of the input image. The reason we use multiple different size is vehicles will show different sizes in the image due to different positions. The range of different sliding windows in as below shows:

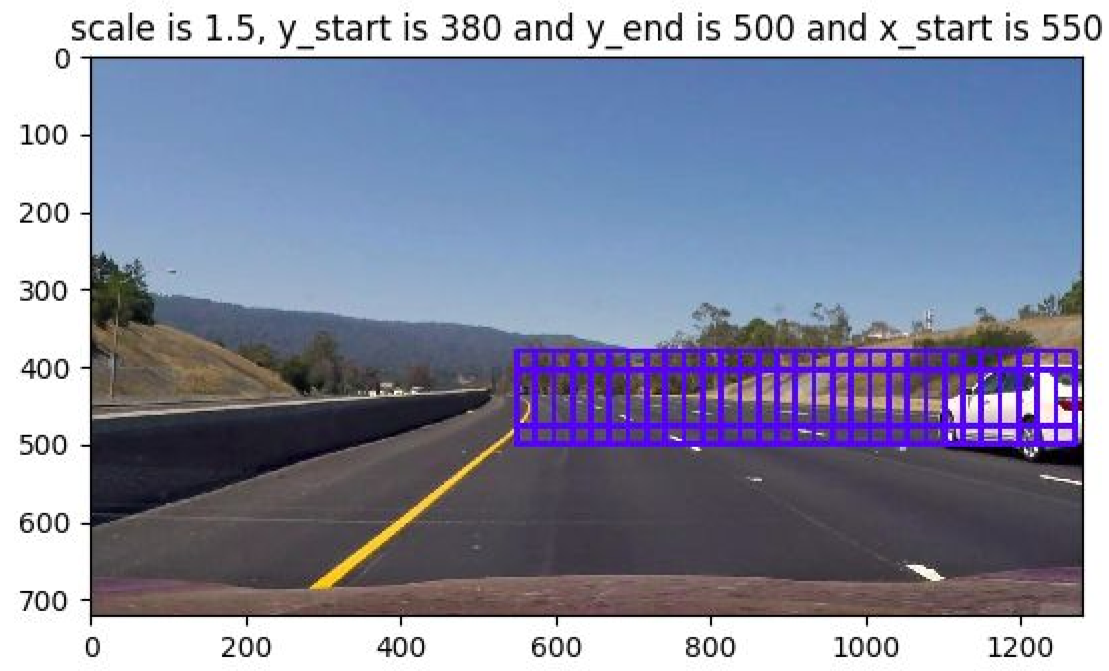


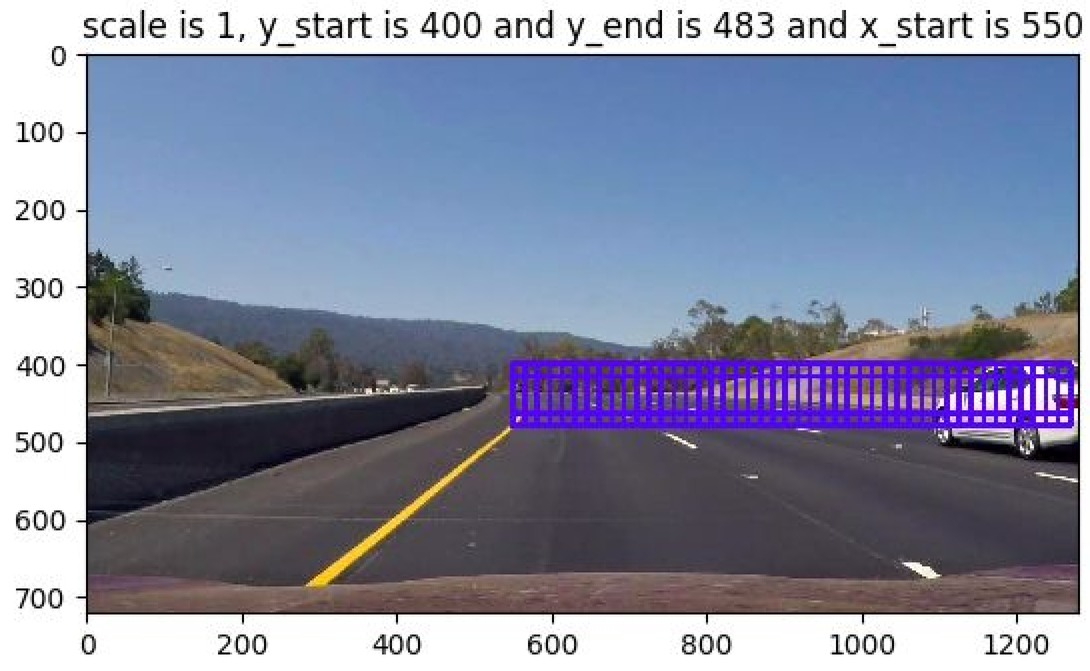
scale is 2.5, y_start is 430 and y_end is 600 and x_start is 550



scale is 2, y_start is 400 and y_end is 540 and x_start is 550

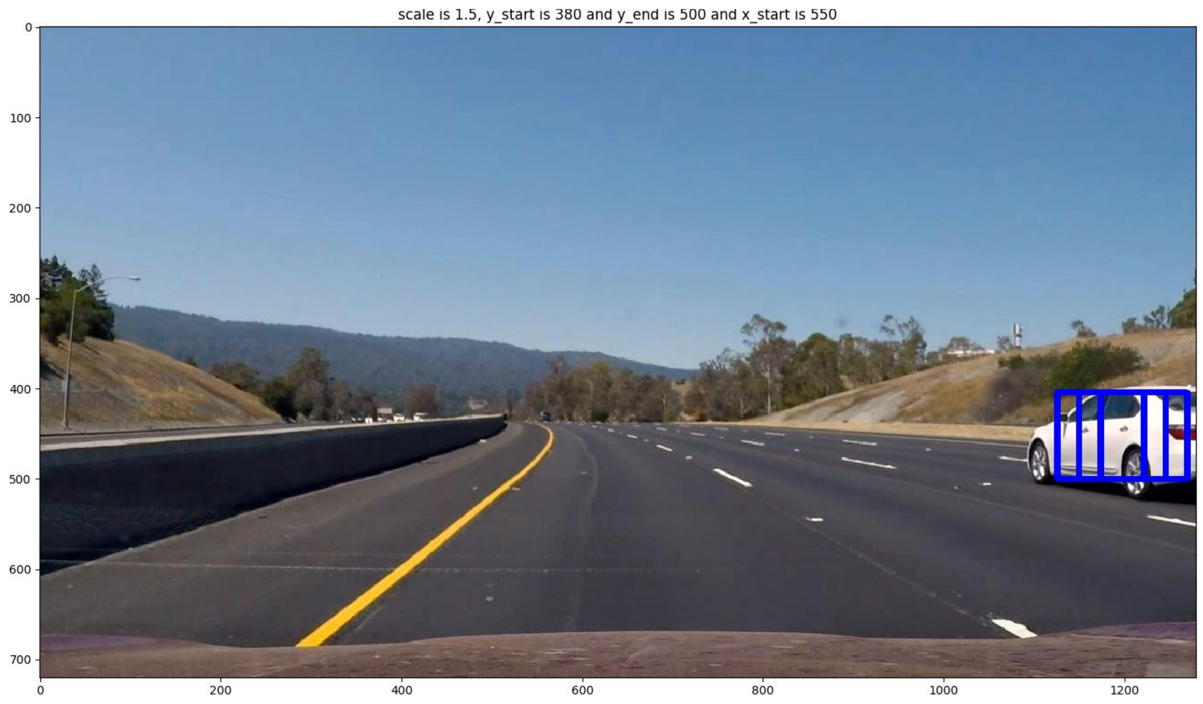






Additionally, the overalp rate is set to 75% which is the best setting in most time because smaller overlap rate will make us lose objects while higher overlap rate will make searching time longer.

The detection results is like:


scale is 1.5, y_start is 380 and y_end is 500 and x_start is 550
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

2) Discussion

There are two things need to consider: 1) the size of the window; 2) hog feature computation.

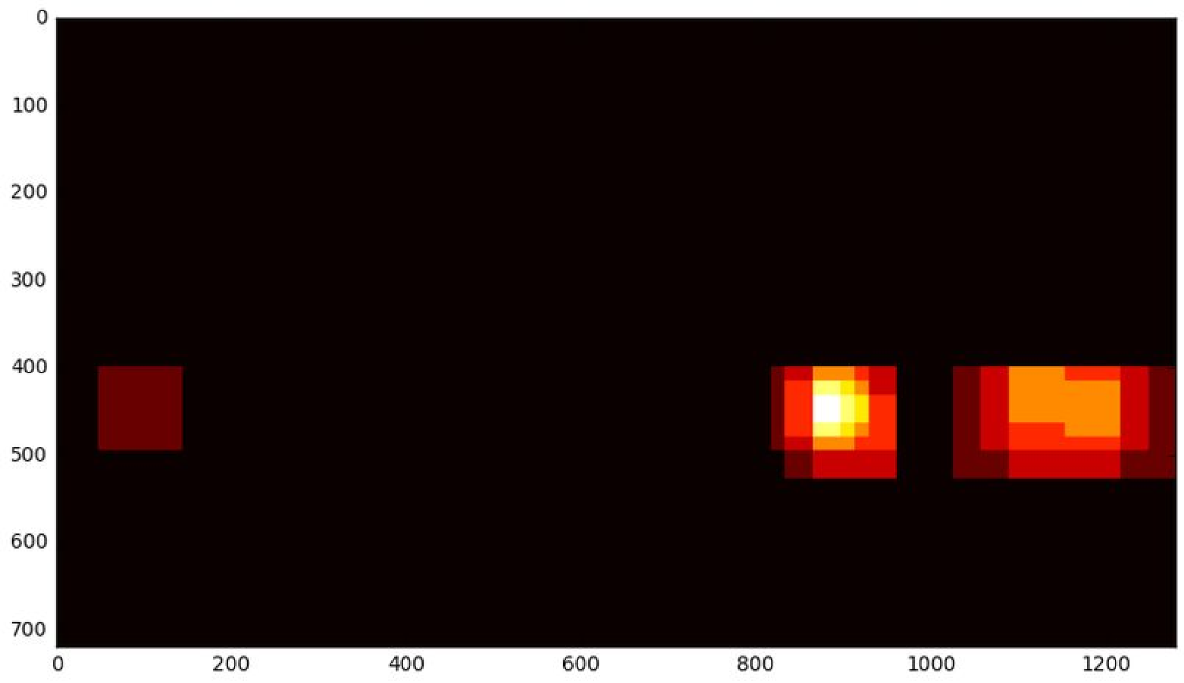
1) Since the format of our training data is all 64x64 image, therefore the window that we want to extract features have to be the same size as our training data. To adapt to different choices of window size, we can either scale up the overall image size if designated window size is smaller than 64x64 or scale down the overall size if designated window size is bigger than 64x64 by extracting the same 64x64 window size.
2) One way to compute hog feature within each window is to clip the window area and then compute each window independently. However, this approach would increase the computational cost. Another way to do this is to compute the overall image hog features as a whole after scaling up/down image accordingly, then subsample the original hog features within the corresponding window by following piece of code:

```
        #extract the whole hog features
        hog_feature_1, hog1 = self.imgobj.hog_features(channel_1,
feature_vec=False)
        hog_feature_2, hog2 = self.imgobj.hog_features(channel_2,
feature_vec=False)
```

```
    hog_feature_3, hog3 = self.imgobj.hog_features(channel_3,
feature_vec=False)

    #num of possible positions for block in x-axis and y-axis
    num_blocks_x = (channel_1.shape[1] // self.pix_per_cell) - self.cell_per_block +
1
    num_blocks_y = (channel_1.shape[0] // self.pix_per_cell) - self.cell_per_block +
1

    #64 pixels has to be 64 x 64 since training data is based on 64x64 !!!
    window = 64

    #num of possible positions for block in window size
    num_blocks_per_window = (window // self.pix_per_cell) - self.cell_per_block + 1
    cells_per_step = 2

    #total number of steps for shifting window in x-axis and y-axis, note if without
dividing by cells_per_step, then cells_per_step is just 1 !!
    num_steps_x = (num_blocks_x - num_blocks_per_window) // cells_per_step + 1
    num_steps_y = (num_blocks_y - num_blocks_per_window) // cells_per_step + 1

    for xs in range(num_steps_x):
    for ys in range(num_steps_y):
        x_s, x_e = xs * cells_per_step, xs * cells_per_step +
num_blocks_per_window
        y_s, y_e = ys * cells_per_step, ys * cells_per_step +
num_blocks_per_window

        #subsample hog features within corresponding window size
        if self.hog_channel == 'ALL':
            hog_feature1 = hog_feature_1[y_s:y_e, x_s:x_e].ravel()
            hog_feature2 = hog_feature_2[y_s:y_e, x_s:x_e].ravel()
            hog_feature3 = hog_feature_3[y_s:y_e, x_s:x_e].ravel()
            hog_features = np.hstack((hog_feature1, hog_feature2,
hog_feature3))
```

4 False positive strategy

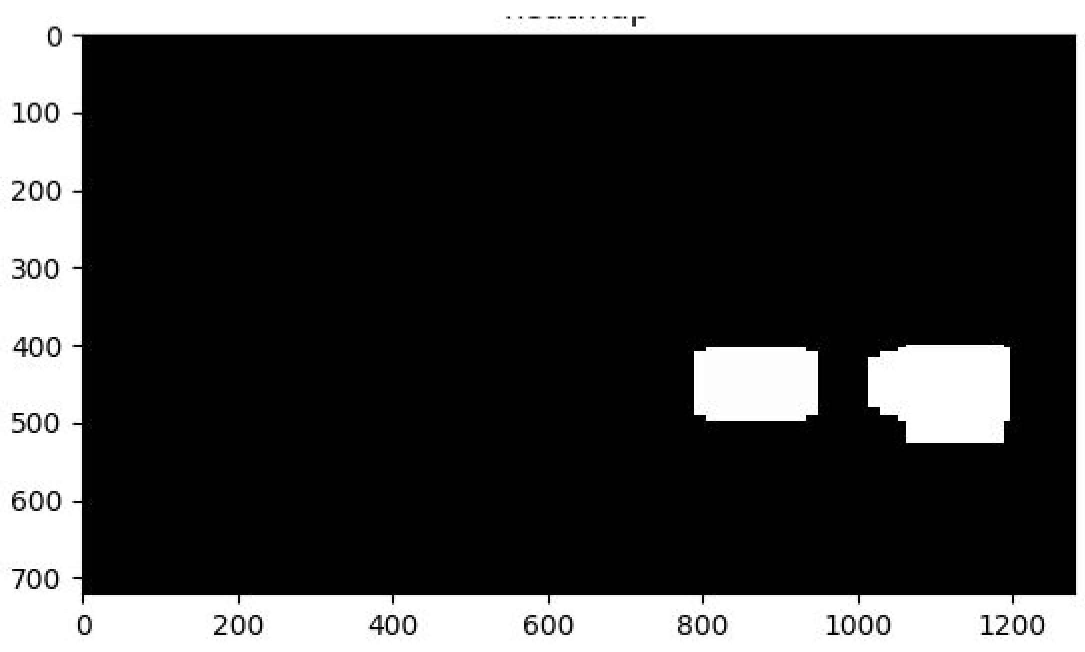
Our detection model does not guarantee the possibility of false positive, to solve this problem, we use the idea of heatmap. The principle of this approach is to use a zero-like image and increment the value if corresponding pixel is within a window that been detected contains vehicle.



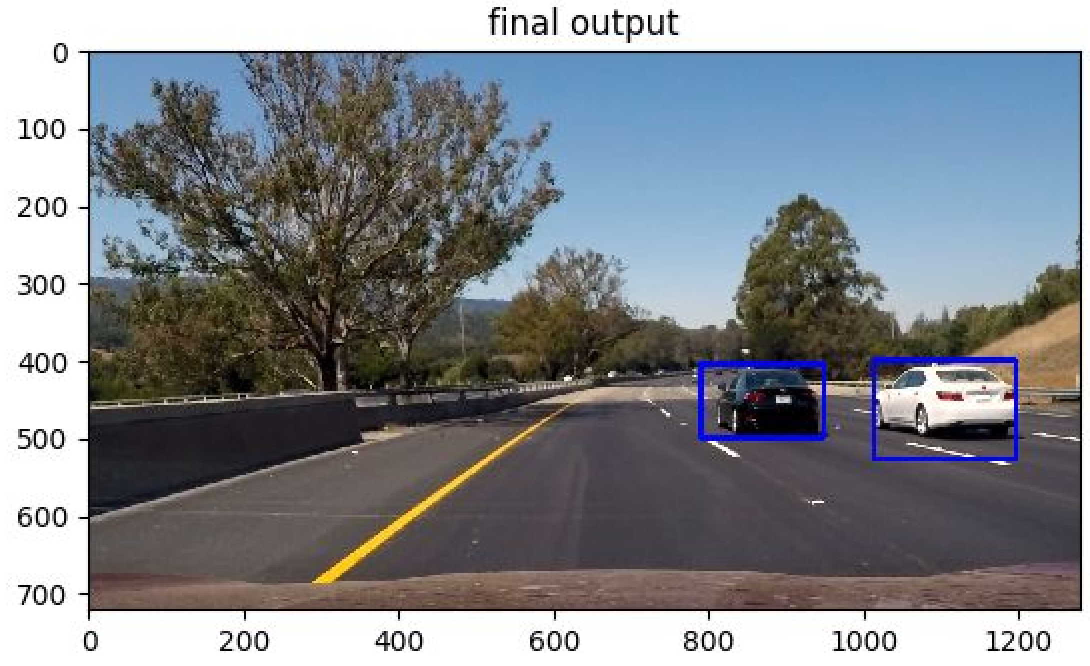
As we can see from above, the right side windows has the higher vaue which means such objects been successfully detected by several overlapping sliding-windows with different positions and different sizes while the left side is more darker which indicates a lower value which is probably a false positive, we can remove false positive results by using a min threshold value, like following code:
heatmap[heatmap <= threshold] = 0

5 bounding box build

To define a bounding rectangle box to indicate the detected object, we use label function which will regard all connected elements with positive value as an object, connected means pixel is reachable via up/down/left/right four directions. For instance, below indicates two objects been detected:



Then, we let the maximum x-axis and y-aixs as the bottom-right coordinate of the bounding box and the minimum x-axis and y-axis as the top-left coordinate of the bounding box, then we have following as our final output:



6 video stream reinforcement

SInce we are not processing individual images, but a stream of video. Therefore, how to set the threshold value which indicated in the last section for removing false positives is important to minimize false positive detections throughout the whole stream. The most naive way of doing this is to set a constant threshold value, however higher threshold value might make us lose objects while lower threshold value could not prevent false positives effectively. In practice, vehicle positions in subsequent frames does not change a lot, thus we can make reasonable assumption that if object does not detected in current frame, it is highly likely not to be detected in a certain range of subsequent frames. On the other hand, if objects been detected in current frame, it is highly likely to be detected in following subsequent frames.

To demonstrate this in our code, we have following logic:

```
    result = 0
    if (self.heatmap > 1).any() == True:
        result = np.int(np.mean(self.heatmap[self.heatmap > 1]))
```

The result value is the average pixel value been detected that bigger than one because one is often tend to be a false positive. Next we define a new class to store result values in previous frames like below:

```
class Record:
    history_track = []
    cache_size = 15
```

History_track is a list of length cache_size which stores result values of previous frames defined above and cache_size defines the size of the list. If the result value is bigger than 0, we append into history_track until exceeds cache_size; if the result value is 0, we clear the history_track. Everytime we deal with a new frame, we let the minimal value in the history_track be our choice of threshold, because minimal value is the most conservative choice since our first priority is to detect objects. If the history_track is empty, we set 2 be a default threshold to use. This is implemented in below logic:

```
if len(Record.history_track) > 0:
    Last_threshold =
mydetect.process_heatmap(np.int(np.min(np.array(Record.history_track))))
else:
    last_threshold = mydetect.process_heatmap(2)
if last_threshold > 0:
    Record.history_track.append(last_threshold)
else:
    Record.history_track = []
if len(Record.history_track) > Record.cache_size:
    Record.history_track = Record.history_track[1:]
```

The approach make sense because frames are related to neighboring frames in a video stream.

7 challenges

The biggest challenge in this project is to define the appropriate sliding window size and their searching range. Good setting will make our algorithm capable of detecting far-away objects because such objects tend to have small size and it is difficult to detect by sliding-windows while large object is relatively easy to achieve.

Our defintion of window size and search range is defined as below:

label	Window size	Start x axis	End x axis	Start y axis	End y axis
1	192x192	550	1280	465	660
2	160x160	550	1280	430	600
3	128x128	550	1280	400	540
4	96x96	550	1280	380	500
5	80x80	550	1280	390	490
6	64x64	550	1280	400	483