Vehicle Detection and Tracking using Machine Learning and HOG

Introduction

The basic objective of this project is to apply the concepts of HOG and Machine Learning to detect a Vehicle from a dashboard video. Wait a minute? Machine Learning and that too for Object detection in 2018? Sounds outdated, isn’t it? Sure, the Deep Learning implementations like YOLO and SSD that utilize convolutional neural network stand out for this purpose but when you are a beginner in this field, its better to start with the classical approach.

Collecting Data

The most important thing for any machine learning problem is the labelled data set and here we need to have two sets of data: Vehicle and Non Vehicle Images. The images were taken from some already available datasets like GTI and KITTI Vision. The images are of size 64x64 and somewhat looked like this-:

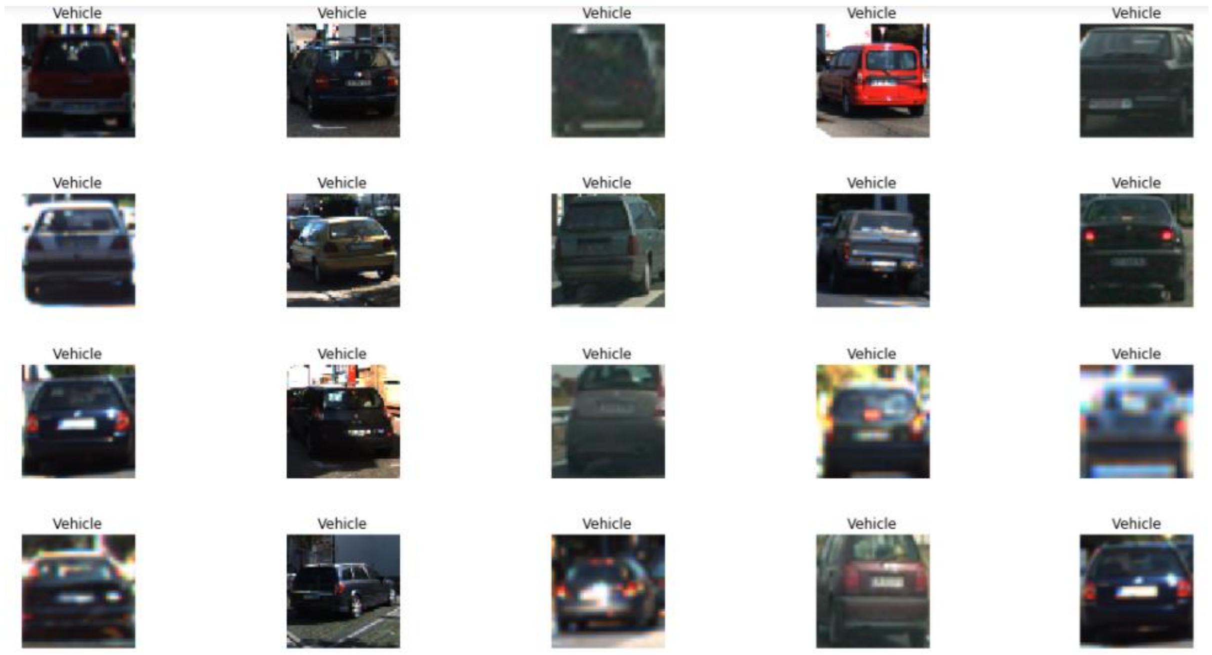


Figure 1. Vehicle Images

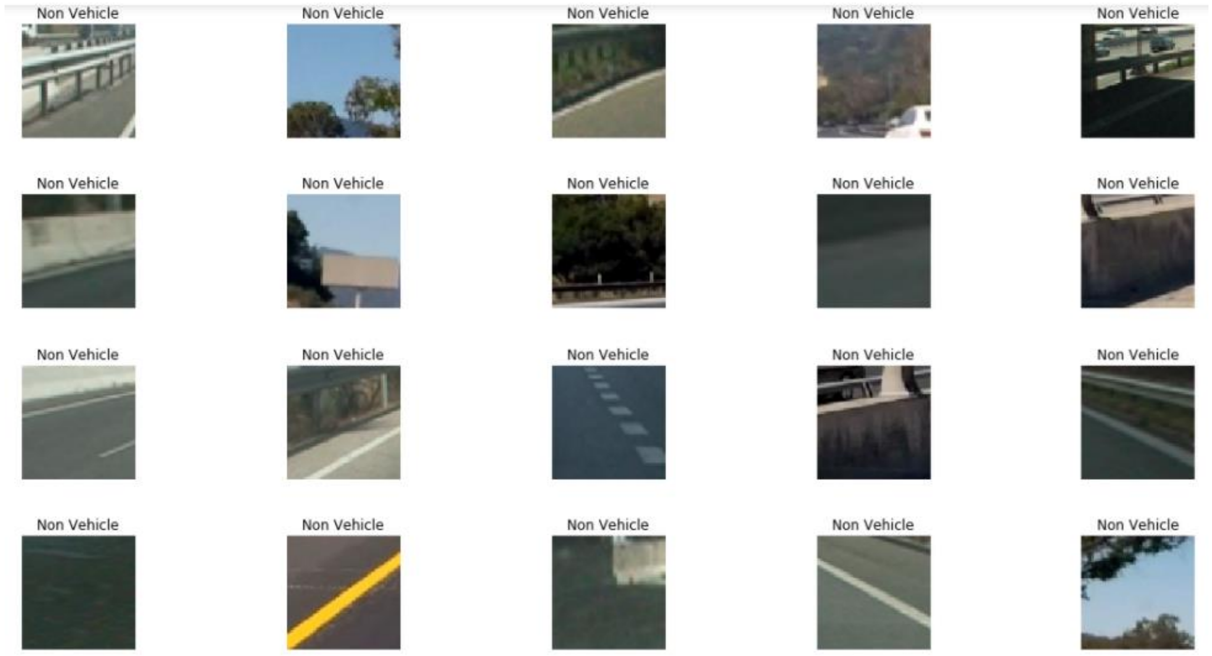


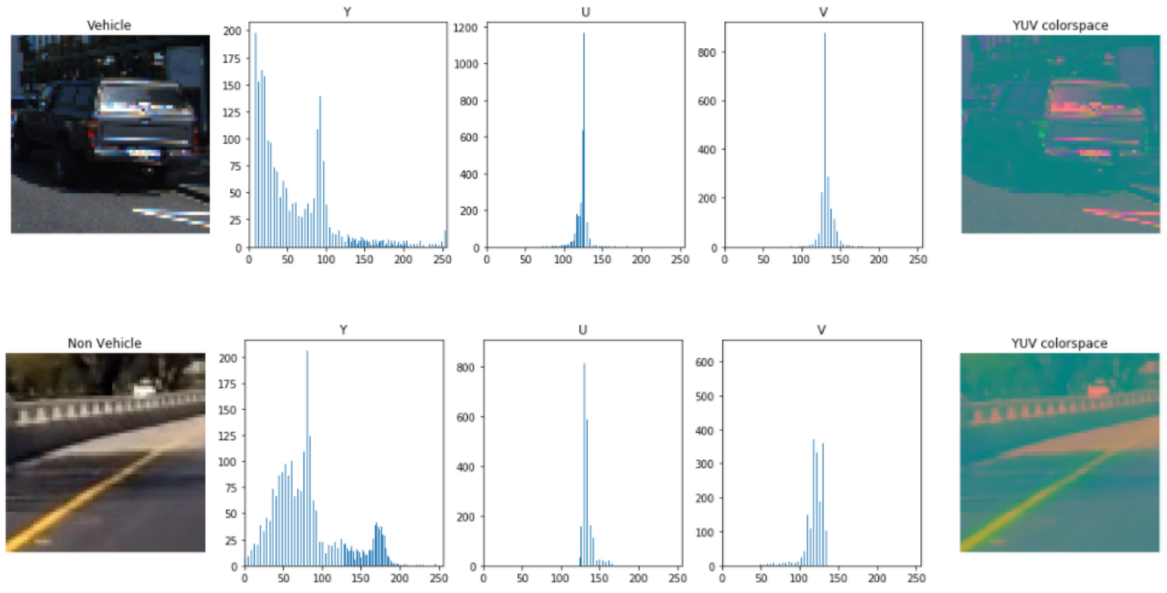
Figure 2. Non Vehicle Images

Extracting Features

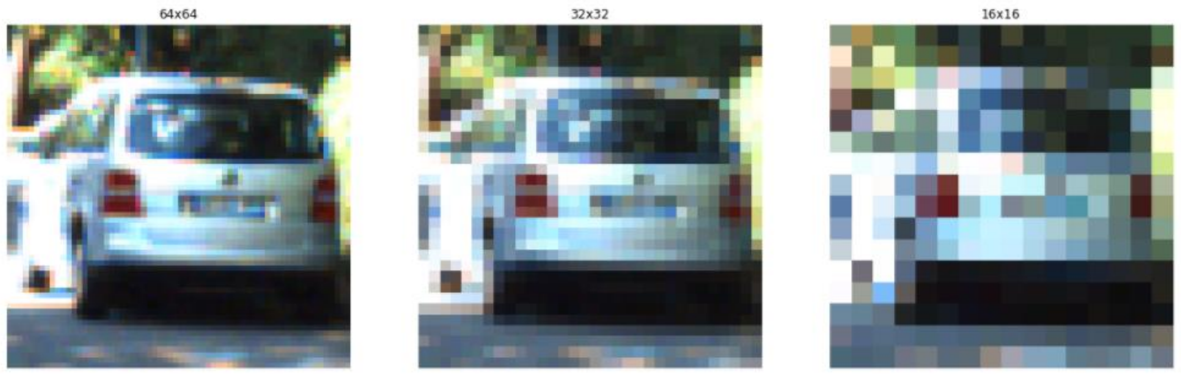
Once we have got the dataset the next obvious step is to extract the features from the images. But why? Why can’t we feed the image as it is to the Machine Learning Classifier? if we do so It will take ages to process the image and just a reminder we are not feeding images to CNN here and this is not a Deep Learning Problem after all!

How to extract the features? Well, there are three good methods if you want to extract features from the images.

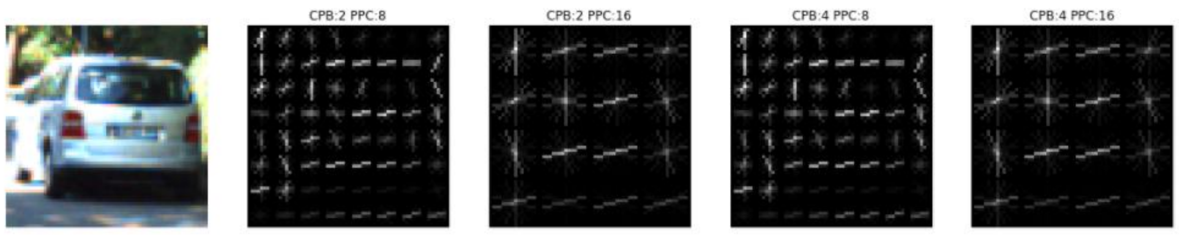
Color Histograms- the most simple and intuitive way is to extract the features from various color channels of the images. This can be done by plotting the histograms of various color channels and then collecting the data from the bins of the histogram. These bins give us useful information about the image and are really helpful in extracting good features.



Spatial Binning- Color Histograms was certainly cool, but if the features of an image are so much important then why can't we take all the features using some sort of numpy function? Well certainly you are correct on this point. We can extract all the information from the image by flattening it using numpy.ravel(). But hold on, let's do some calculation, image size is 64x64 and it is a 3 channel image so the total number of features extracted are 12,288!! Close to 12k features from a single image is not a good idea! So here Spatial Binning comes to picture! What if I say, a 64x64 image gives the same information as 16x16 gives? Of course there is some loss of information but still we are able to extract good features out of the image! So if I apply numpy.ravel() to a 16x16 image, I would get only 768 features!



HOG (Histogram of Oriented Gradients)- The feature extraction techniques discussed above are pretty cool but certainly not much powerful as compared to HOG. HOG actually takes an image, divides it into various blocks in which we have cells, in cells we observe the pixels and extract the feature vectors from them. The pixels inside the cell are classified into different orientations and the resulting vector for a particular cell inside a block is decided by the magnitude of the strongest vector. Note- here we are not counting the occurrence of a pixel in a particular orientation but instead we are interested in the magnitude of the pixel in that particular orientation.



Just a point to note here. OpenCV HOG returns Hog Image and Feature Vectors but the length of image.ravel() is not equal to feature vector length. This is because HOG internally performs some computations and reduces the redundancies in the data and returns optimized feature vectors. Also more the number of lines you see in the image means it will return more features.

Generating Dataset and Data Preprocessing

Ok, Now we know how to extract features so we will process these steps for all images? Yes, you are right but it is not necessary to use all features from all the methods above. Let’s use only HOG for the moment and ignore color histograms and spatial binning.

Let’s decide on the HOG parameters to extract features. After a lot of hit and trials I decided to go with the following-:

- Orientations- 9
- Cells Per Block- 2
- Pixels Per Cell- 16
- Colorspace- YUV

Cool, after running images through the HOG function with these parameters the final parameter size comes out to be 972 which is pretty cool!

Data Preprocessing

Now our features are ready the next step is to pre-process the data. Don’t worry sklearn library is always there to help in these tasks. We can perform following preprocessing-:

i) Shuffling Data
ii) Splitting the Dataset into training and test set
iii) Normalization and Scaling of Data (Fit and Transform of Dataset)

An very important point here to note is that after Step (ii) we have to fit and transform the data, but we should not fit the data in the test set because we do not want our classifier to sneak peak into our data.

Training Classifier

Well, features are extracted, the data is preprocessed! What next? Yup, now comes the turn of our classifier. The choice of classifier is yours but there are a plenty to chose from-:

- Support Vector Machines
- Naive Bayes
- Decision Tree

I decided to use Support Vector Machines because they have good compatibility with HOG. Now in SVM we have SVC(Support Vector Classifier) and here also we have a choice with various kernels and different C and gamma values.

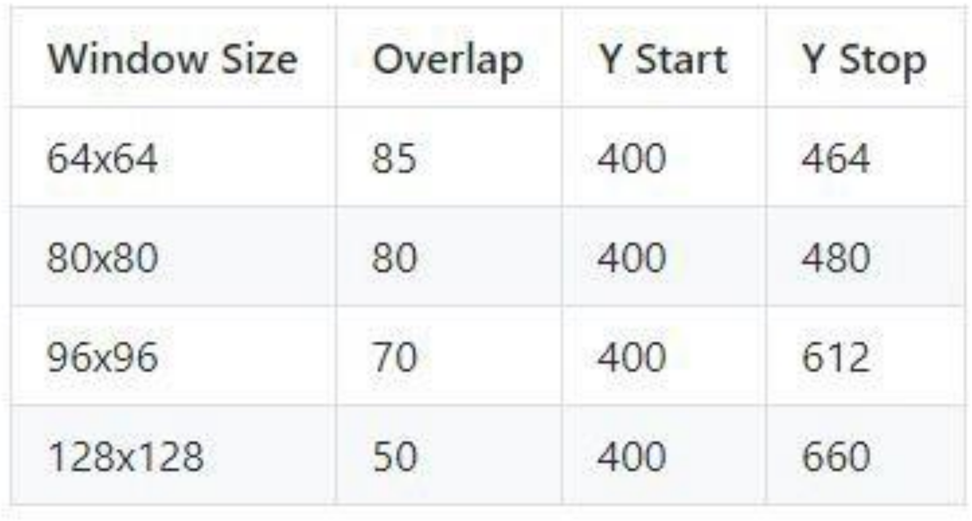
I trained my classifier on both Linear and rbf kernel. The linear kernel took around 1.8 seconds to train with a test accuracy of 98.7% while rbf kernel took around 25 minutes to train with a test accuracy of 98.3%. I decided to use LinearSVC with default parameters solely because it was taking less time to run and it was more accurate than rbf kernel.

Sliding Window

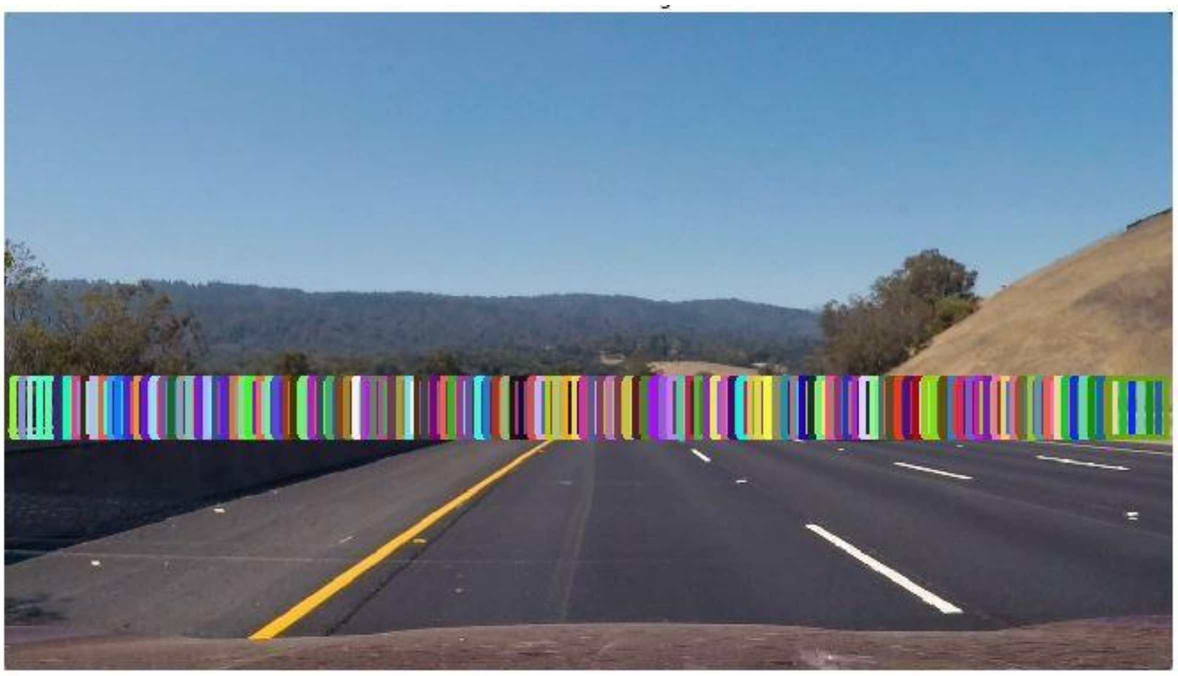
Cool our classifier is now well trained and it will 99% of time will be able to predict vehicles and non vehicles correctly. So what is the next step? Well, the next step is to apply the classifier to patches of your image in order to find where exactly in the image the car is!

But first you need to decide on various important parameters. The first thing is from where do you start searching the car from, obviously you should not search the car in the sky, hence you can ignore the top half of the image, so basically decide a horizon under which you will search your cars. The second important thing is what will be the window size you will look for and how much two windows should overlap? Well that depends on your input image length, since here it is 64x64 so we are going to start with base window size of 64x64 only. The next important thing and very important point here to note is **that you search for smaller cars near the horizon and as you move towards the dashboard camera you search for larger cars**. This is because if the cars are near to horizon they are smaller as they are distant from your car and reverse is the case with the near cars.

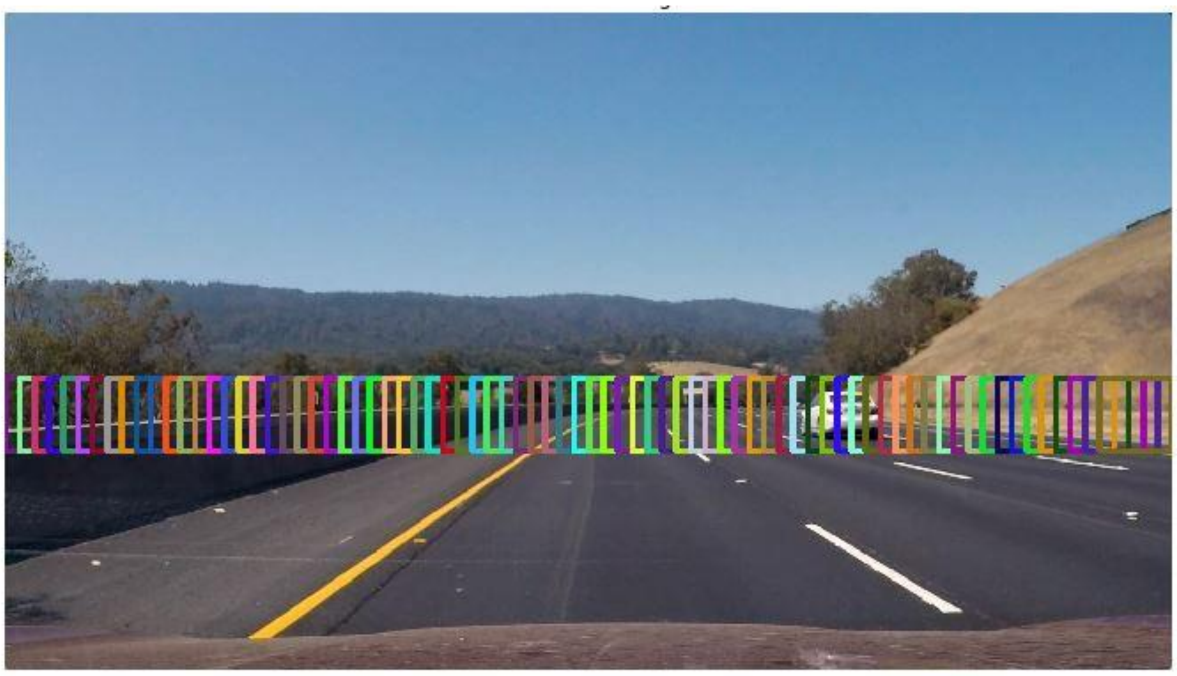
I decided to use windows of 4 different sizes. In the below images I will try to illustrate my search area with the respective window sizes.



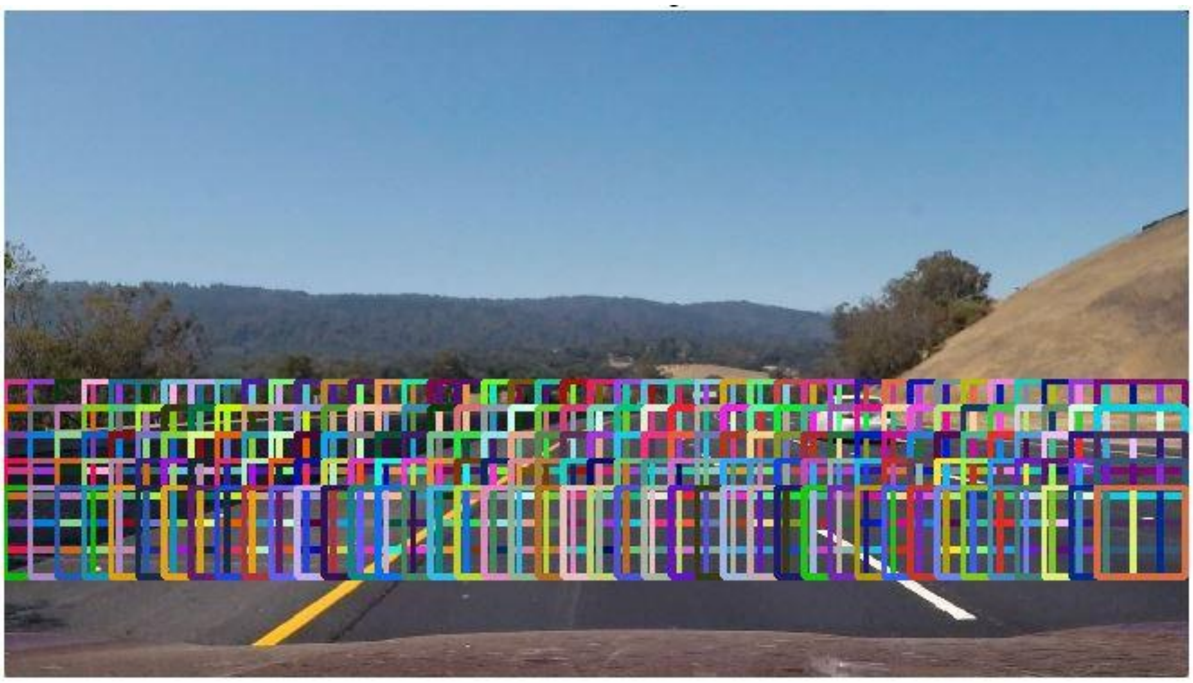
Window Size	Overlap	Y Start	Y Stop
64x64	85	400	464
80x80	80	400	480
96x96	70	400	612
128x128	50	400	660



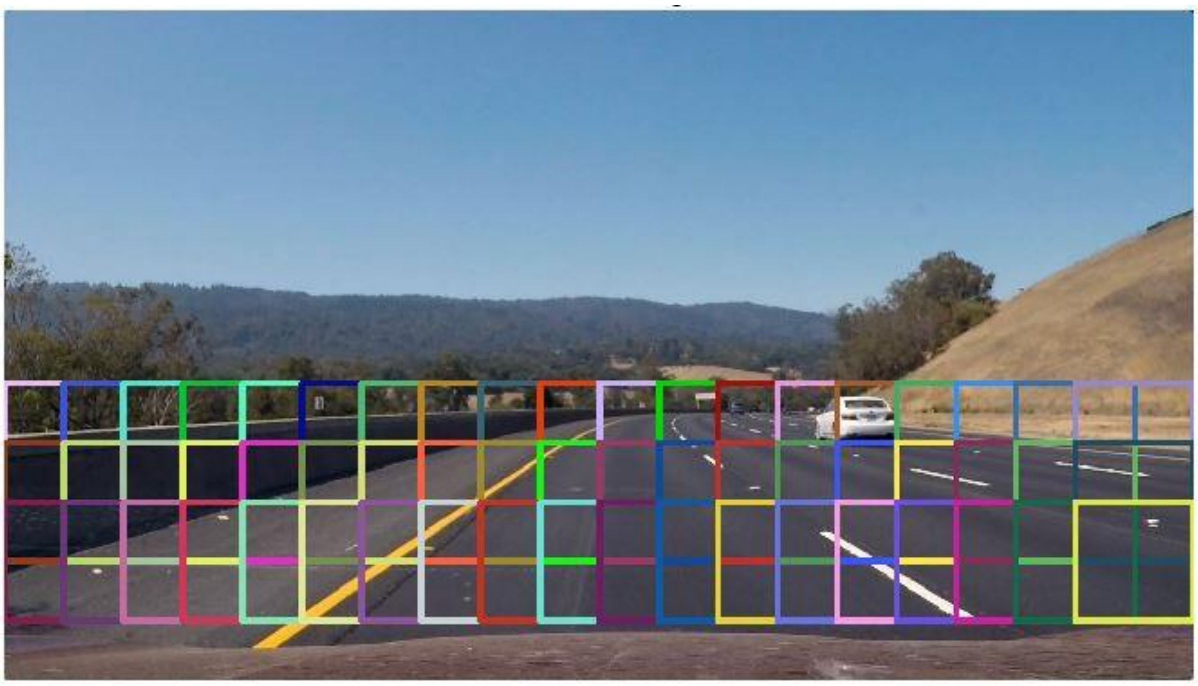
Window Size 64x64 Coverage



Window Size 80x80 Coverage

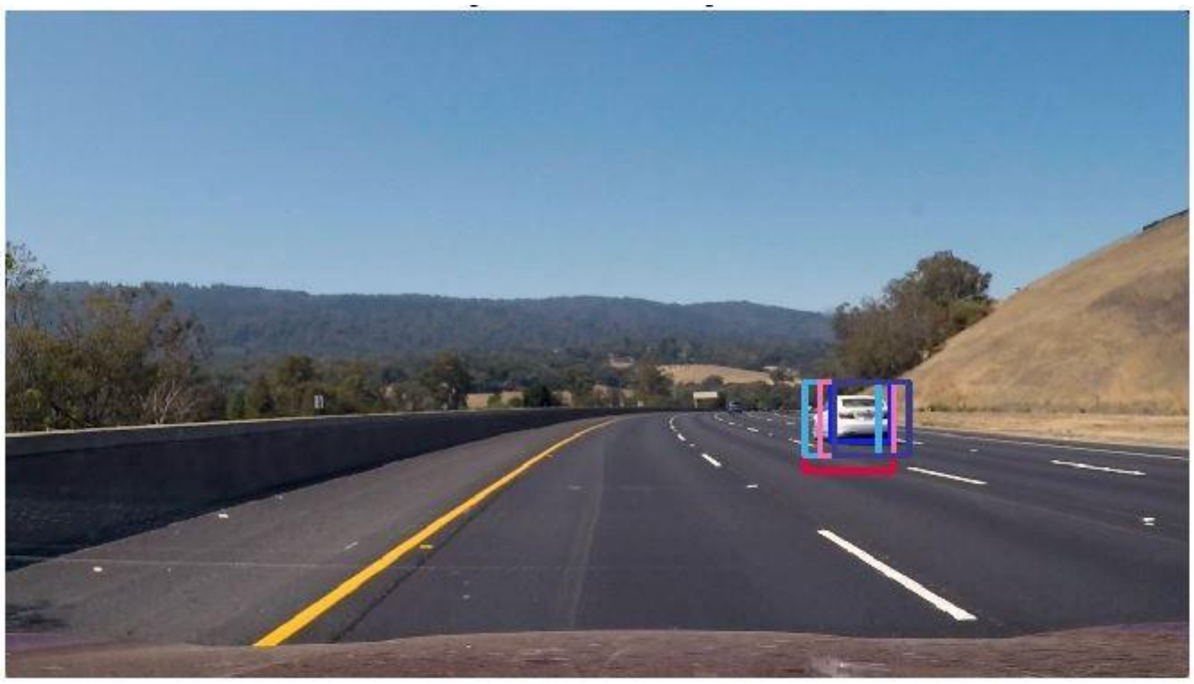


Window Size 96x96 Coverage



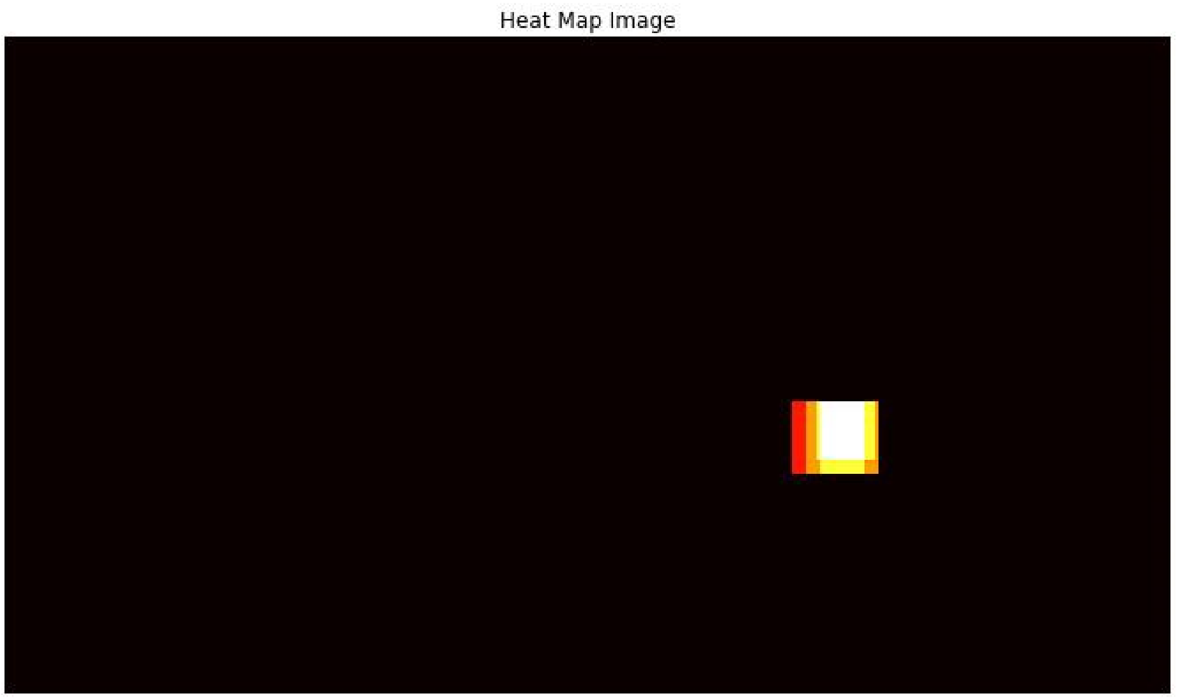
Window Size 128x128 coverage

I had total window size of 470! So once we have defined all the sliding windows we will be searching for, the next step is to extract features of all the patches window by window and run our classifier to predict if the found window is car or not. Remember we trained our model on feature extracted from 64x64 image so for windows that are not of the same size we need to resize first them to 64x64 in order to keep the features same. Well lets see how our classifier worked.

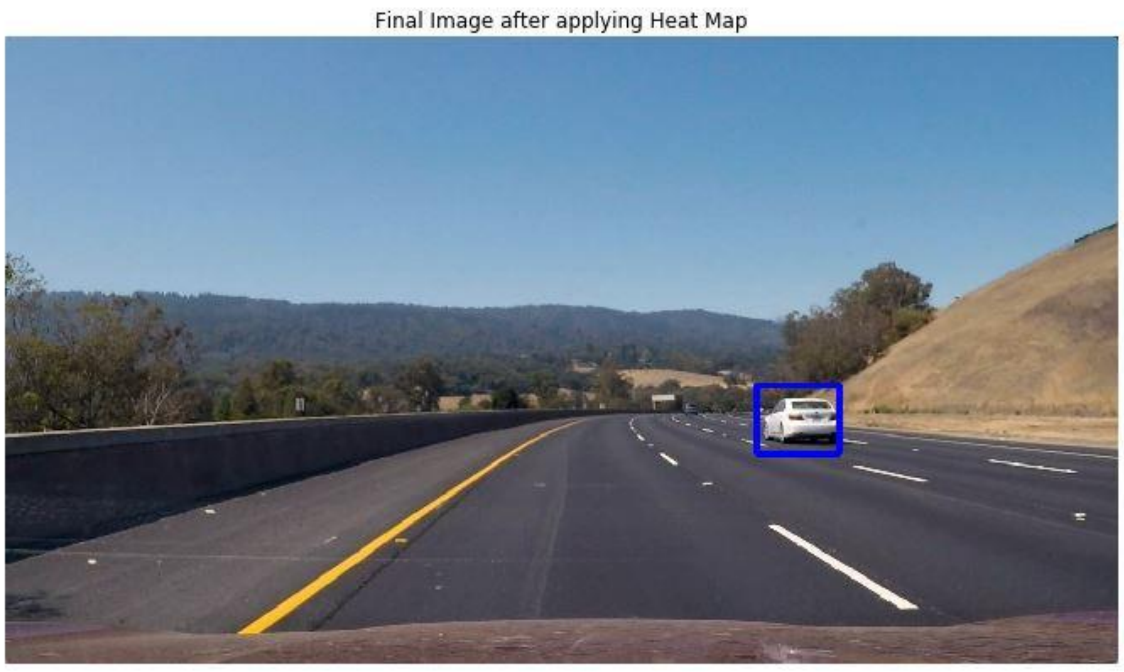

Refined Windows after running the classifier.

Heatmaps

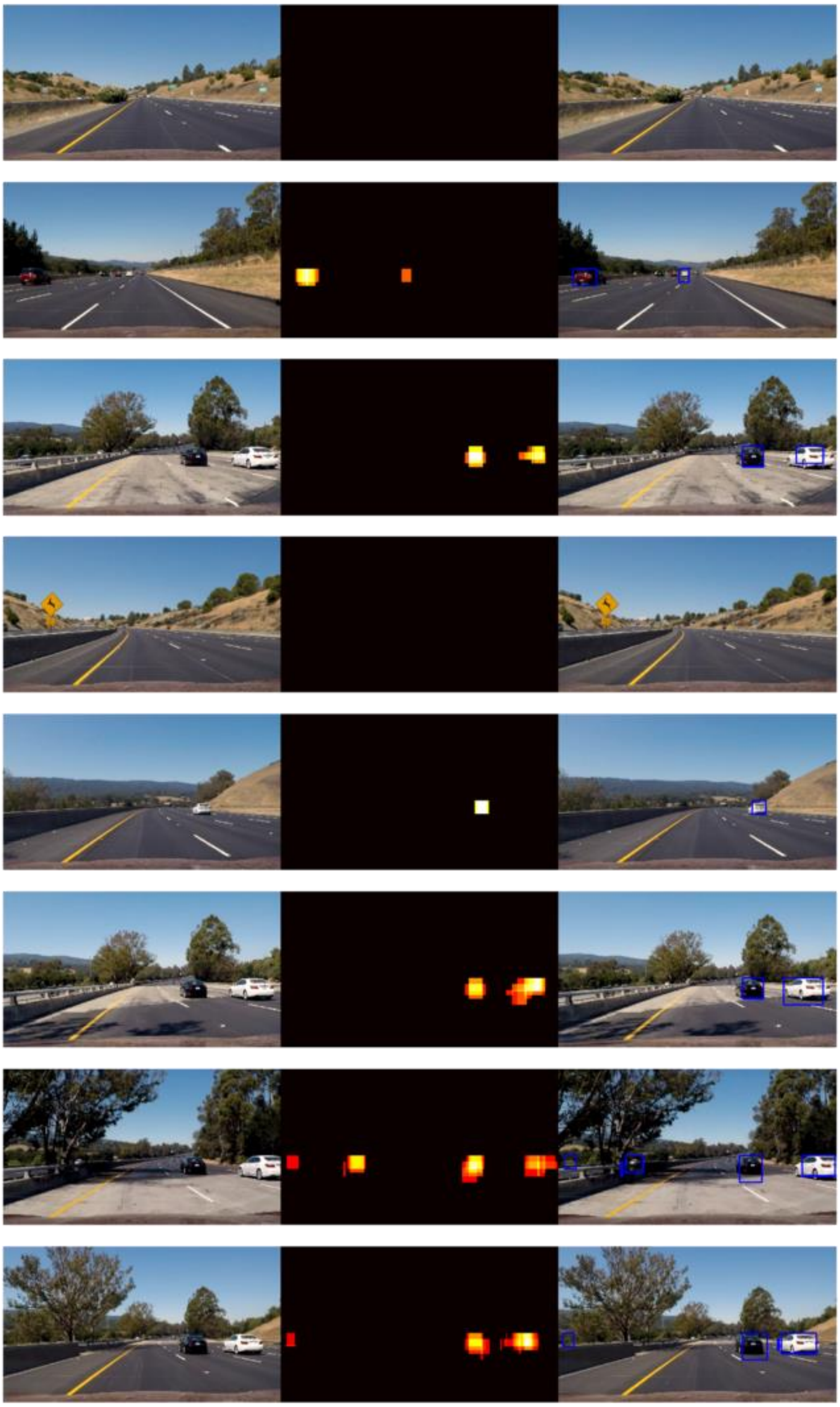
So we are able to detect the sliding windows, but there is a problem. So many windows are overlapping with each other, how to draw the final bounding box? Answer is Heatmap. We will create a blank black image of the same size as that of original image and for all refined windows that were identified we will add the pixels values by one for the whole region of the refined window. In this way we will have regions with different intensity with the common region being the most intense. We can then apply a threshold to clip the final image and get the coordinates of the final box.



. Heatmap drawn after increasing the pixel intensities of refined windows



There is one more problem when you run the code on a number of more test images.



Well as you an observe there are some false positives detected in our image, cars coming from left lane are also detected, so how do we solve this? First of all we need to observe how this problem came up in the first place? Well our classifier had 98.7% accuracy. We had total number of windows to be 470. So in the resulting windows we will have around 6 windows that will be false positives. These windows can appear anywhere if you have a lower threshold in the final heatmap image. So to solve the problem of car coming in the other lane and some false positives can be solved by just increasing the threshold. In my case I set the threshold to value 4 but again thresholding depends on a number of factors, the colorspace used, the SVM accuracy and so on.

Averaging Out

Well we are almost done at this moment! The pipeline works fantastic with the images but still there is one problem if you will run the pipeline on the images coming from a video stream. The final detected boxes will become very shaky and will not deliver a smooth experience, it may be possible that the box goes away in some frames. So what’s the solution? The solution is pretty intuitive, store all the refined windows detected from the previous 15 frames and average out the rectangles in the current frame. Also you need to adjust threshold to a higher level now. Just by doing so the final bounding boxes appear less shaky and delivers a smooth flow.