Vehicle Detection

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.

In this project we are using traditional computer vision approach for object detection which consists of following steps:

1- Visual Feature Extraction

2- Search images for detection

3- Track those detections from frame to frame in video stream.

Dataset

I started by reading in all the vehicle and non-vehicle images from both GTI and KITTI dataset. The first code cell in the Jupyter Notebook reads in all the data and produces some of the statistics on it. It prints out the type and dimension of the images as well as the number of images in each vehicle and non-vehicle dataset.

This Vehicle image is: <class 'numpy.ndarray'> with dimensions: (64, 64, 3)

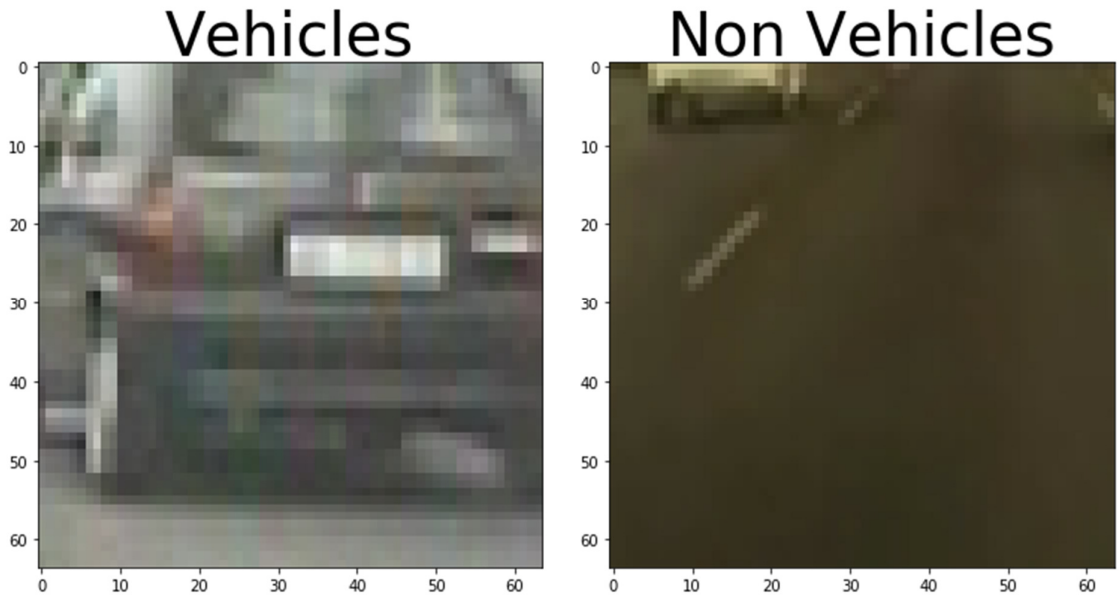
This Non Vehicle image is: <class 'numpy.ndarray'> with dimensions: (64, 64, 3)

Number of Vehicle Images 8792

Number of Non-Vehicle Images 8968

As the number of vehicle and non-vehicle images are almost equal, it is a balanced dataset and doesn't need data augmentation.

Here is an example of one of each of the vehicle and non-vehicle classes:



Feature Extraction

Once you have all the data, the next step is to extract features from it. This is done by the functions defined in code cell 2 of the ipython notebook- bin spatial(), color_hist() and get_hog_features().

Color Binning and color histogram:

Template matching is one way to extract features where you take the raw pixel values and use them to create a signature of the car. The bin_spatial reduces the size of the image and create a signature of the car.

Template matching use the complete image and doesn't work if there is change in shape/size. For these histograms of colours are better option as they preserve the colour information and good for searching for distinct colours but can be less accurate.

Histogram of Oriented Gradients (HOG)

get_hog_features() is contained in the second code cell of the IPython notebook. Next step for feature extraction is the gradient techniques to get the information about shapee/size.

HOG is used to extract the gradient features which divides the whole picture into cells and calculate the dominant gradient direction of that cell by using the histogram of gradients of each pixel in that cell.

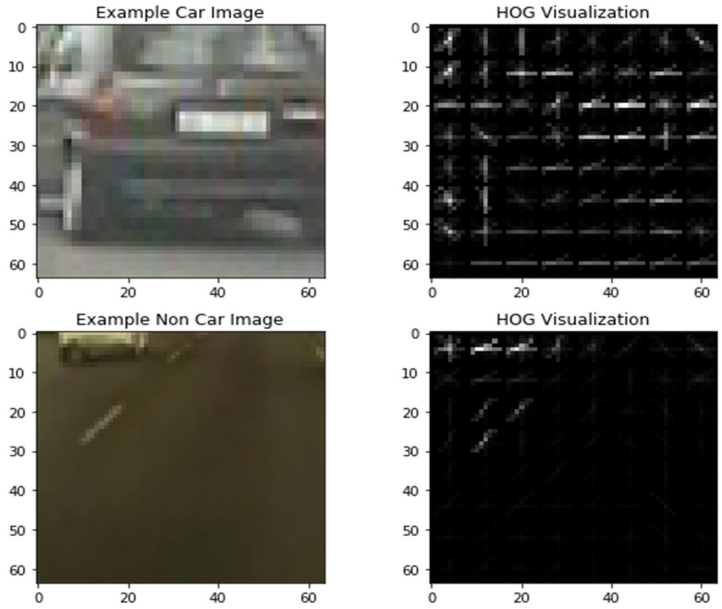
scikit-image HOG is used to implement the HOG. It takes in various parameters:

Orientations- all the histogram bins for gradient orientation

pixels_per_cell - number of pixels in each cell for histogram computation

cells_per_bloc- number of cells in each block for normalisation

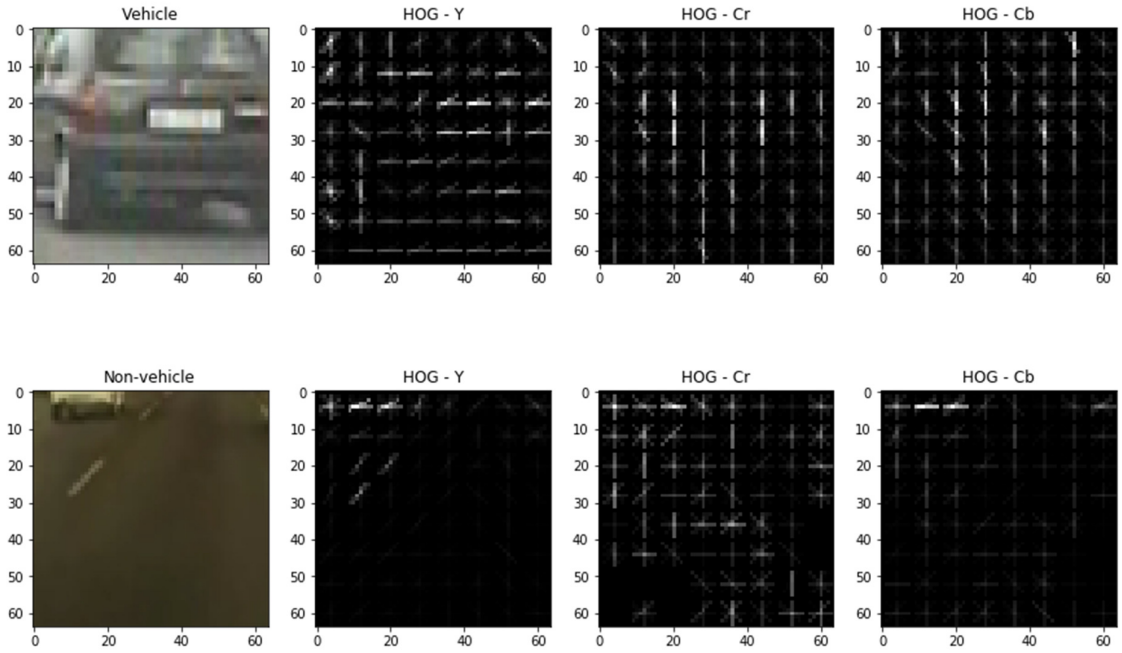
In addition to this, hog features can be applied to different colour spaces like HLS etc. I grabbed random images from each of the two classes and displayed them to get a feel for what the skimage.hog() output looks like.



I then explored different color spaces and different skimage.hog() parameters (orientations, pixels_per_cell, and cells_per_block) in order to generate better distinctions and finally settled for

```
colorspace = 'YCrCb' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 8
pix_per_cell = 8
cell_per_block = 2
```

Here is an example using the YCrCb color space and HOG parameters of orientations=8, pixels_per_cell=(8, 8) and cells_per_block=(2, 2):



Fitting the model and Parameter tuning

I extracted all the features using extract_features() function define in code cell 4 of the python notebook. This function combines all the aforementioned feature extraction functions. Once we have the features extracted from all the data, they are stacked together to form a feature vector. This is done for both cars and non cars dataset.

After that, code cell 5 is used to try out different parameters and train the model. To achieve this, following steps are performed:

1- Given lists of car and non-car features (the output of extract_features(), a labels vector is defined as a binary vector indicating whether each feature vector in our dataset corresponds to a car or non-car (1's for cars, 0's for non-cars).

2- Next a Random Shuffling of the data is done to avoid problems related to ordering of data

3- After that, a test-train split is performed with 0.2 split so 80% training data and 20% test data.

4- Normalizing and scaling the data: to avoid individual feature or set of feature dominating the classifier. Scalar is calculated only on the test data and then applied to the training data as well.

A linear SVM is used as a classifier and is iterated through a set of parameters to come up with the following set of feature parameters values which gave me more accuracy in the least amount of time:

```
colorspace = 'YCrCb' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 8
pix_per_cell = 8
cell_per_block = 2
hog_channel = 'ALL' # Can be 0, 1, 2, or "ALL"

spatial_size=(16,16)
hist_bins=32
hist_range=(0, 256)
```

I tried various combinations of parameters and these gave over 99% accuracy and also the fitting time of just about 5 seconds

This is the result from the classifier:

99.28 Seconds to extract features...
Using: 8 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 5568
4.92 Seconds to train SVC...
Test Accuracy of SVC = 0.9938

Sliding Window Search

Once the classic is ready to use, the images are searched to find the detections. This is achieved by extracting features from the image and passing it to the classifier.

The detection is done by the sliding window algorithm

To implement a sliding window search, we need to decide what size window we want to search, where in the image we want to start and stop our search, and how much we want windows to overlap

The sliding window search is applied in the code cell 6, 7 of the python notebook.

Following functions have been defined:

slide_window(): sliding window to genrate the diffenret windows to be searched based on the given X, Y start/stop , window size and overlap

draw_boxes(): function to draw bounding boxes

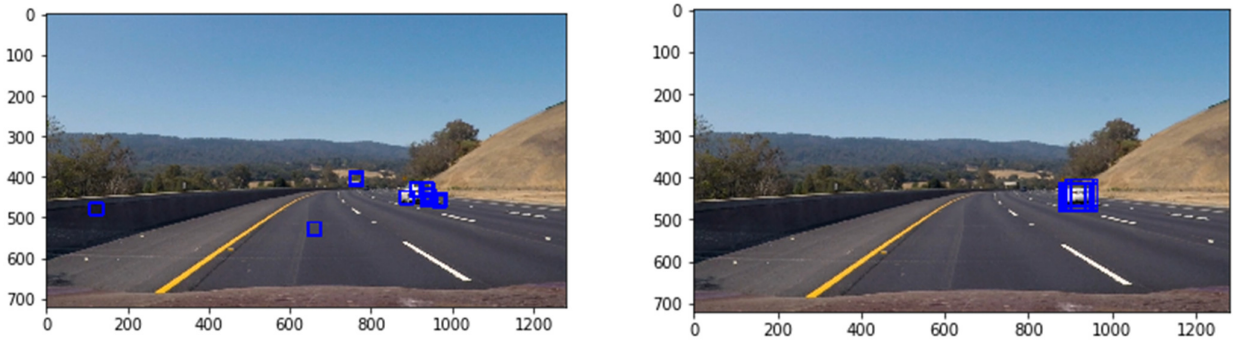
single_img_features() : to extract all the features from a single image

search_windows (): function that takes in an image and the windows to be searched and outputs the windows where the classifier predicts positively

Iterative values are used to come up with the parameter values which would detect cars with more accuracy and have less false positives.

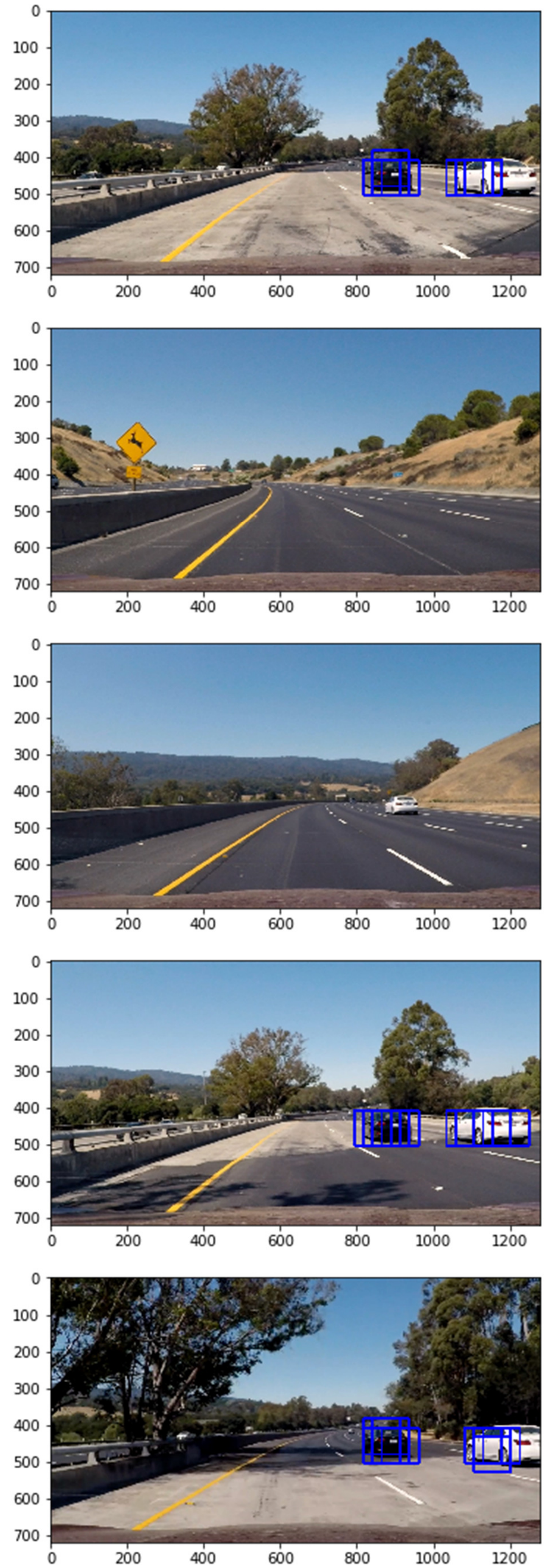
I used 64x64 window size, 85% overlap and y_start_stop= [360, 700] (mainly to crop the image to be searched so that I would not be searching in the sky

As shown in the image, it detected all the cars but it has false positives. Increasing the overlap increases the detection but also increase the false positives. Too large windows were missing the cars or too small windows were taking a long time to compute and producing a lot of false positives



Next a hog sub sampling technique is applied in the code cell 10 function find_cars of the ipython notebook.

The find_cars only has to extract hog features once, for each of a small set of predetermined window sizes (defined by a scale argument), and then can be sub-sampled to get all of its overlaying windows. Each window is defined by a scaling factor that impacts the window size. I used scale of 1.5, window size of 64x64 and cells per step of 2



A) Hog Sampling **B) HOG Subsampling**

False positives and multiple detections

Once the car frames are detected, next step is to remove the multiple detections and the false positives. This is done by creating a heat map. The functions used for this are defined in the code cell 12 of the ipython notebook.

From the positive detections as a result of sliding window implementation, I created a heatmap and then thresholded that map to identify vehicle positions (mainly to remove false positives). I then used scipy.ndimage.measurements.label() to identify individual blobs in the heatmap. I then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected. This removes the multiple detection issue. Here's the example result showing the heatmap for all the test images:


Car Positions
Heatmap
Car Positions
Heatmap
Car Positions
Heatmap
Car Positions
Heatmap
Car Positions
Heatmap

Video Implementation

Main pipeline

process_image() is the main image function defined in code cell that takes in the video frame one by one and detect the vehicles by using following steps:

1-Use findcars() to implement sliding windows using hog_subsmapling

2- Create a heat map from all the detections

3-Threshold the heatmap to remove false positives and label it to remove multiple detections

4- Return the image with final boxes around the heatmap.

Video :

VideoFileClip() and fl_image(process_image) functions are used to create a clip and applying the process_image function to every frame of the video. The code performed good on the test video provided and it is able to detect the vehicles without significant departures or failures.

Problems/Issues

Even though the detection algorithm works, there are still a lot of improvements that can be made. The detection boxes jump from one frame to another and the current pipeline fails to detect the cars under some lighting conditions. Here are some steps that can improve the behaviour:

1- Braking down the sequenced data in the datasets and use of better classifier. The current classifier gives more than 99% accuracy but the data it is trained/ tested upon is a sequenced data. So it might be better to randomise the data

2- use of variable window sizes in sliding window search - using larger windows nearby and smaller windows for further away on the road

3- The heat map boxes jump from Frame to frame. A better approach would be to take the heat map history and have an average over few frames to smoothen them out.

4- We can also predict the movement of boxes from frame to frame to have a better gauge on where the car is moving