P5 Write-up Report

Vehicle Detection jiyang_wang@yahoo.com

1. The Goals and Steps of the Project

- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
- Optionally, apply a color transform and append binned color features, as well as histograms of color, to the HOG feature vector.

Note: the first two steps require normalizing the features and randomizing a selection for training and testing dataset.

- Implement a sliding-window technique and use the trained classifier to search for vehicles in images.
- Run the 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.
- As an optional challenge, combine the pipeline for vehicle detection with lane-finding algorithm from the last project to do simultaneous lane-finding and vehicle detection!

2. Vehicle Detector Overview

My vehicle detector consists of the following 5 major steps, which work as a pipeline: All these steps/functions are called by CarDetector that is defined as a class:

```
class CarDetector()
```

1. It gets the video frames as inputs and extracts shape features (using **HOG**) and color features (using **color histograms** and **spatial bins**). The HOG

features are constructed across a whole video frame only once, instead of in each sliding window used in step 2 below.

2. It then scans through the lower part of the frames (which is the region of interest) by virtually **multi-scaled sliding windows** and applies the above features within the windows to the **SVM classifier** already trained to look for vehicles in the video frames.
3. Finally, it builds the **heat map** to exclude the outliers (likely not cars) by **heat thresholding** in the frames and plots the **bounding boxes** around the vehicles found.
4. The training process of the **linear SVM model** is relatively independent from vehicle detection pipeline. It uses the image dataset provided by Udacity to train a linear SVM model. The input images are pre-processed in almost the same way as in step 1. After being trained, the model is saved as a **pickle** file which will be read by `CarDetector()`.
5. As part of my work on this project, the video frames with the vehicles found and the bounding boxes plotted are fed to **Lane Detector** of my last project so that the final output of the pipeline has both the lane lines detected and plotted as well as the vehicles detected and surrounded by the bounding boxes.

The following sections provide detailed descriptions of each of the above steps that cover all the **rubric points** of this project.

3. Shape Features by Histogram of Oriented Gradients (HOG)

The code for this step is contained in code cell just below "`Shape Features Using HOG`" in the IPython notebook. The function is called

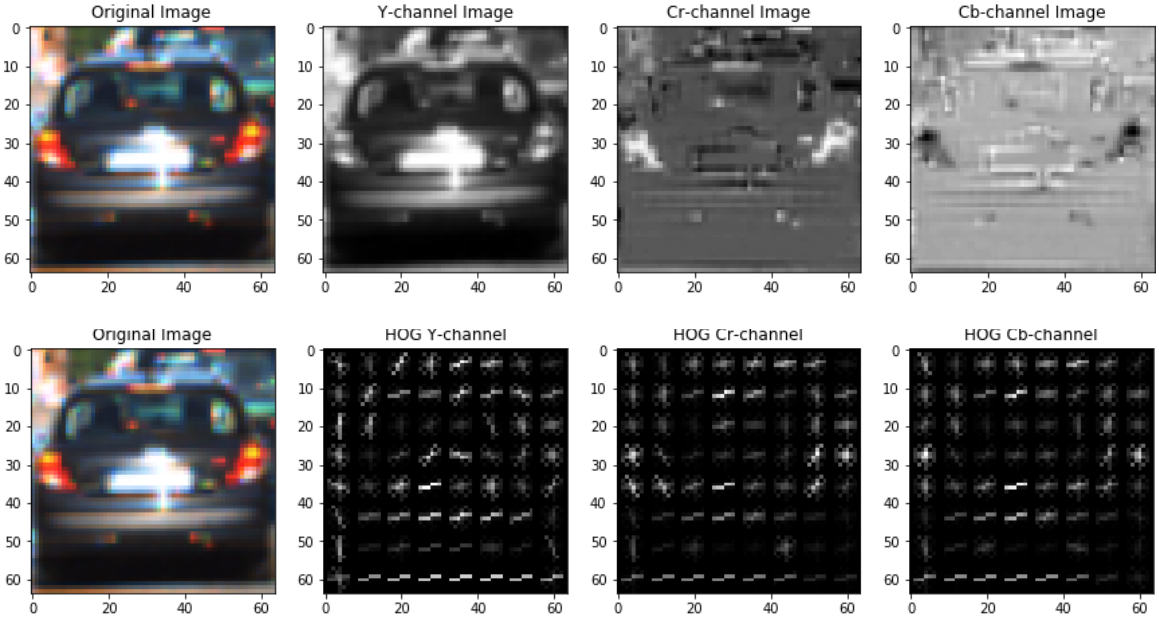
```
get_hog_features(img, orient, pix_per_cell, cell_per_block,
                 vis=False, feature_vec=True)
```

whose major part is the hog() function imported from skimage library. After exploring various color spaces, I chose YCrCb as the one for feature extraction. YCrCn turned out to better distinguish the two classes of objects, i.e., cars and not-cars in the images by shape and color features.

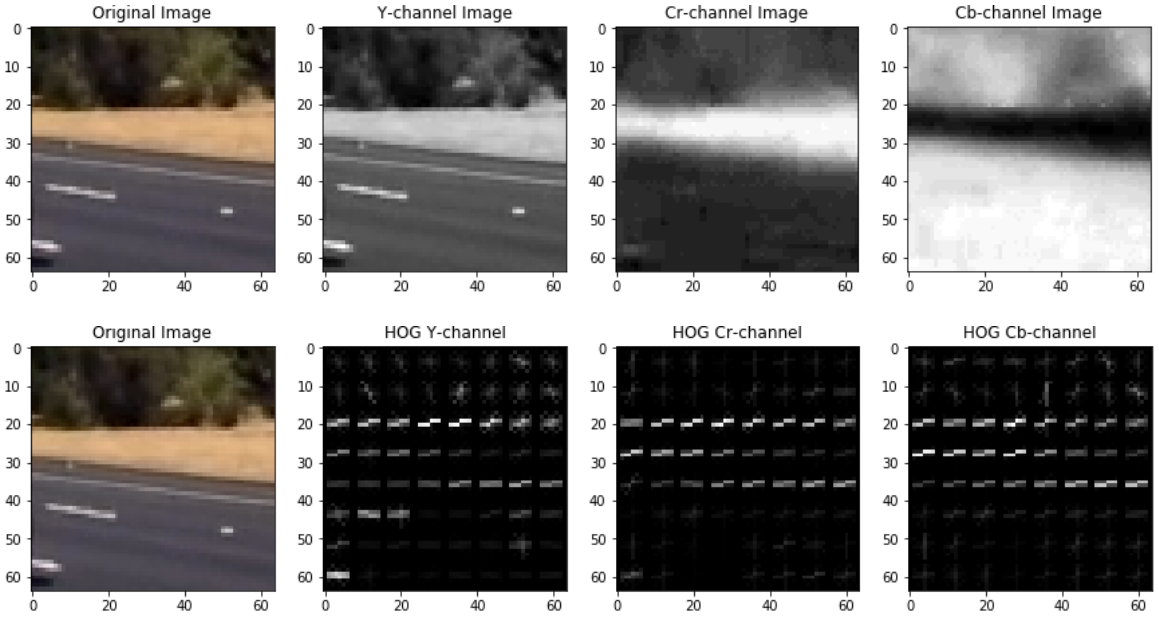
Here are the results of color conversion RGB2YCrCb and of HOG feature extraction of a random example image in the "vehicles" dataset, with

parameters:

orient=8, pix_per_cell=(8, 8) and cells_per_block=(2, 2)



And an example from "non-vehicles" dataset:



We can see that the two types of objects have different patterns of HOG features.

4. Color Features by Color Histograms and Spatial Bins

To increase the reliability and accuracy of the SVM classifier, I concatenate the color features to the HOG features, which form a larger feature vector.

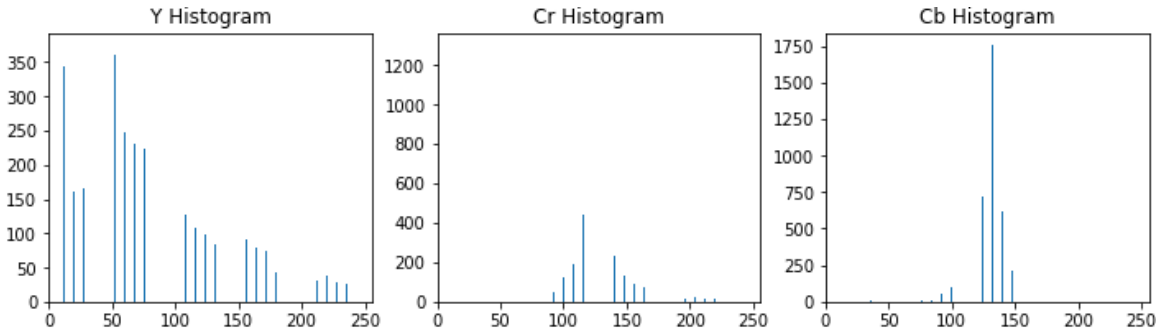
Two color features are extracted. One is the color histogram over 64x64 images, which is implemented by function

`color_hist (converted_image, nbins=32)`.

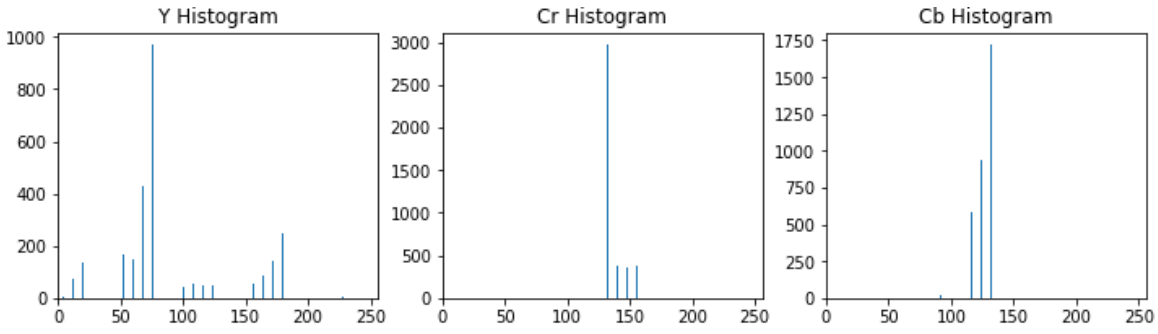
The other is spatial bins in which pixel intensity is split into certain number of bins for each color channel, and the number of pixels whose intensity falls in a certain bin is counted. This is implemented by function

`bin_spatial (converted_image, bin_size=(32, 32))`.

Here are the results of color histogram feature extraction from the "car" image:

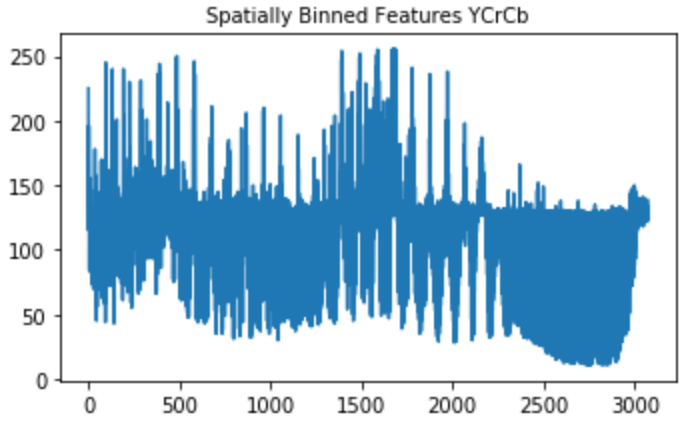


As compared with the color histogram of a "non-vehicle" image:

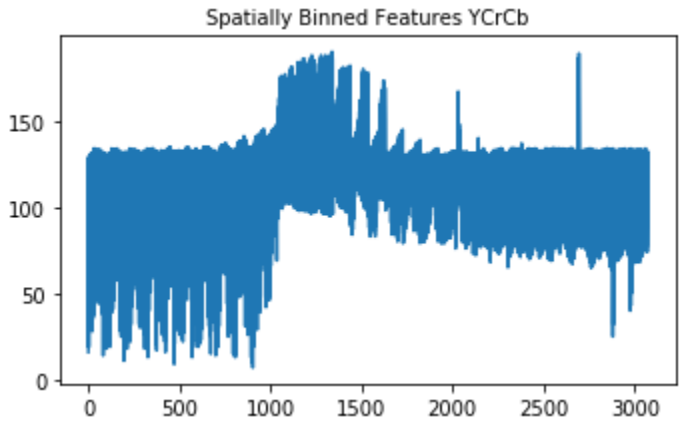


They do have very different patterns, which can be used by the classifier to distinguish the two types of objects.

The spatial bin features of the "vehicle" image:



Versus the spatial bin of the “non- vehicle” image:



And, again, they have clearly different patterns.

5. Training a Linear SVM Classifier

A linear SVM is trained by the dataset provided by Udacity. The dataset has two categories, i.e., 64x64 .png images of vehicles, and 64x64 .png images of various types of non-vehicle objects.

The feature vector is constructed by stacking of HOG features of all the 3 color channels with the two color features described above. Features on all the example images in the dataset are extracted by function `extract_features()`.

Then, feature vector X of “car” and “not-car” needs to be **normalized** by the following code (because different types of features can have very varied scales of values, and some may dominate the classifier while others have little impact):

```
# Fit a per-column scaler for feature normalization
```

```
X_scaler = StandardScaler().fit(X)
scaled_X = X_scaler.transform(X)    # Apply the scaler to X
```

Just before training the model, the normalized dataset is **randomized** and split by 80:20 ratio into training set and testing set by function `train_test_split()`.

Now we can train the linear SVM model:

```
svc = LinearSVC()           # create a SVM model
svc.fit(X_train, y_train)   # train the model
```

After the model is trained, we save it to a pickle file together with X_scaler with the filename `svc_pickleYCrCb3000.pkl`.

6. Sliding Window Search

This function is implemented by `find_cars()`.

A conventional sliding window approach is to use a small window of the same size as the training images, i.e., 64x64 in this project, to slide through the region of interest on the video frames. The sub-regions on the frames that fall in the window at different time are treated individually, through feature extraction and classification on each sub-region one by one to see whether objects of interest ("car" or "cars") in each sub-region. This is very time consuming.

As suggested by the lecturer, a better approach is to run HOG feature extraction over the whole image ROI only once. Let's call it HOG feature map. The HOG feature in each sub-region is just the sub-region on the HOG feature map, i.e., it is from sub-slicing the HOG feature map by the size and location of the sub-region.

As the sliding window scans through the whole image ROI, it obtains HOG feature vector directly from the HOG feature map, which is then "stacked" with color features within the sliding window to form an integrated feature vector, which is then fed into the classifier. The classifier will tell us whether "car" or "cars" are found in this sub-region.

A few important parameters need to be explained:

pix_per_cell: pixels per cell. We use 8x8 cell, so pix_per_cell = 8 (pixels)

cell_per-block: a block means a block of cells that are entitled together to pixel intensity normalization. We use 2x2 cells in each block, so cell_per_block=2.

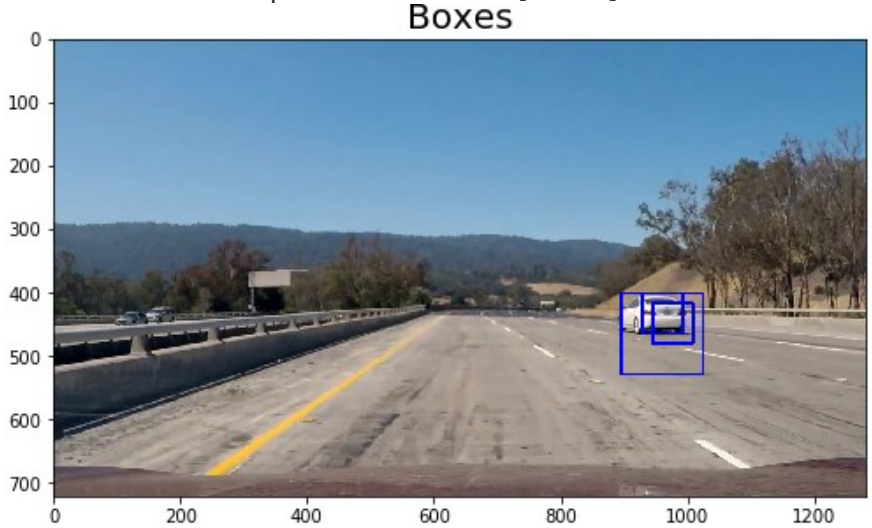
nblock_per_window: this is number of blocks per window. We have the window size of 64x64 pixels, or 8x8 cells. Each block has the size of 2x2, and each time a block makes pixel normalization, it moves one cell in x and y direction. So the number of blocks pre window is 8 – 2 + 1 = 7.

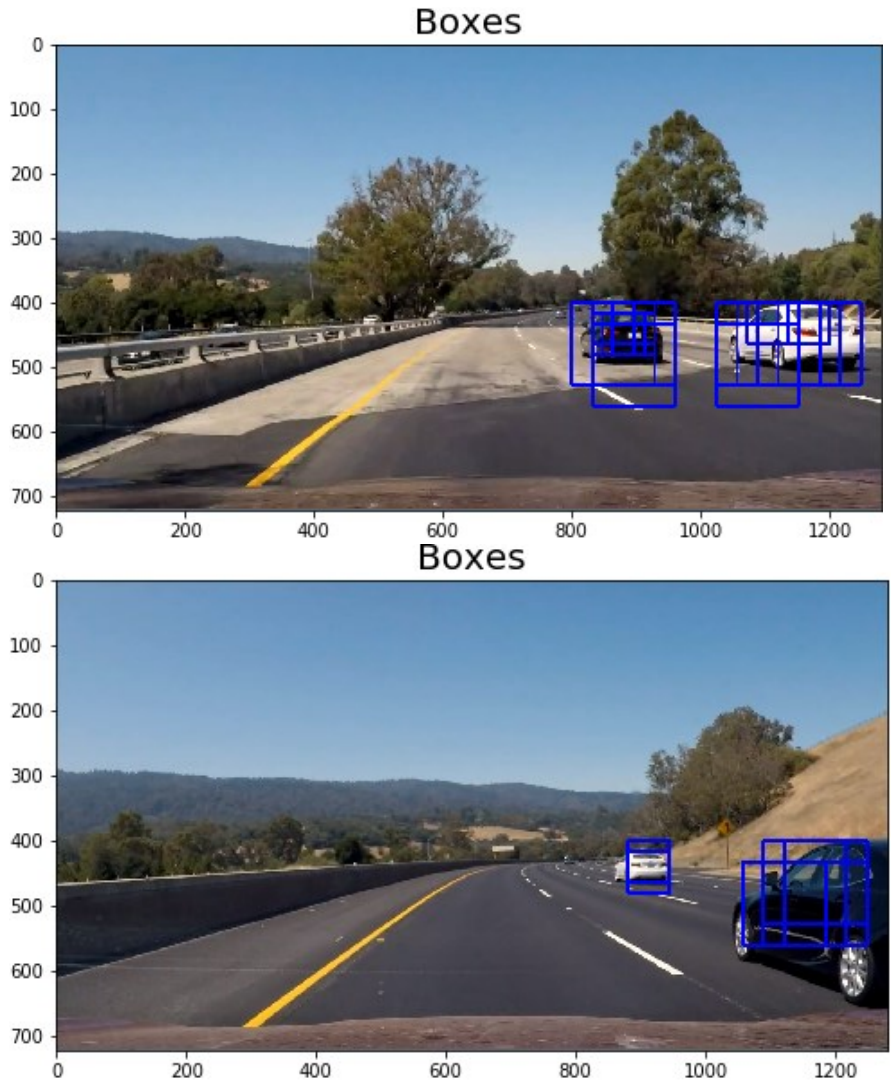
cells_per_step: this is the step by which the sliding window moves on the image. For example, if cells_per_step = 2, each time the sliding window moves two cells in x or y direction.

scale: cars in longer distance appear smaller in the image than the cars in shorter distance. This fact asks us to use multi-scale sliding windows. A 64x64 window is good for "cars in distance", but it's better to also use larger window to scan the image for "cars close by".

However, instead of using windows of multiple sizes over the same image, we always use the same 64x64 window over the images of different "scale" factors, which means sub-sampling the image if scale>1 when using resize() function. A single window over multi-scale images has the same effect as multi-scale windows over the same image, but has the advantage of always extracting features in a 64x64 window regardless of the image scale.

Here are some examples when scale = [4, 2, 1]





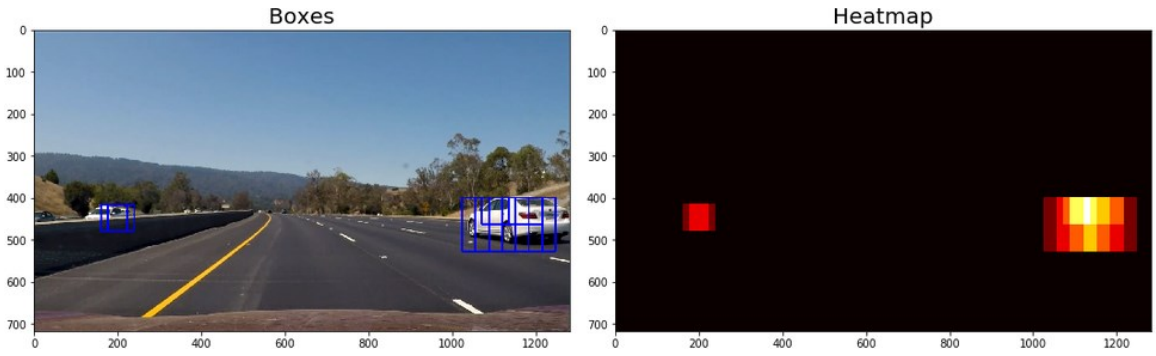
7. Multiple Detections and False Positives

There are cases where the detector gives us overlapping detections over the same vehicle, or false positive detection on “not-car” objects or on the vehicles in the opposite direction. We need to filter them out.

We can build a heat map from these detections in order to combine overlapping detections and remove false positives. Here is an example (next page):

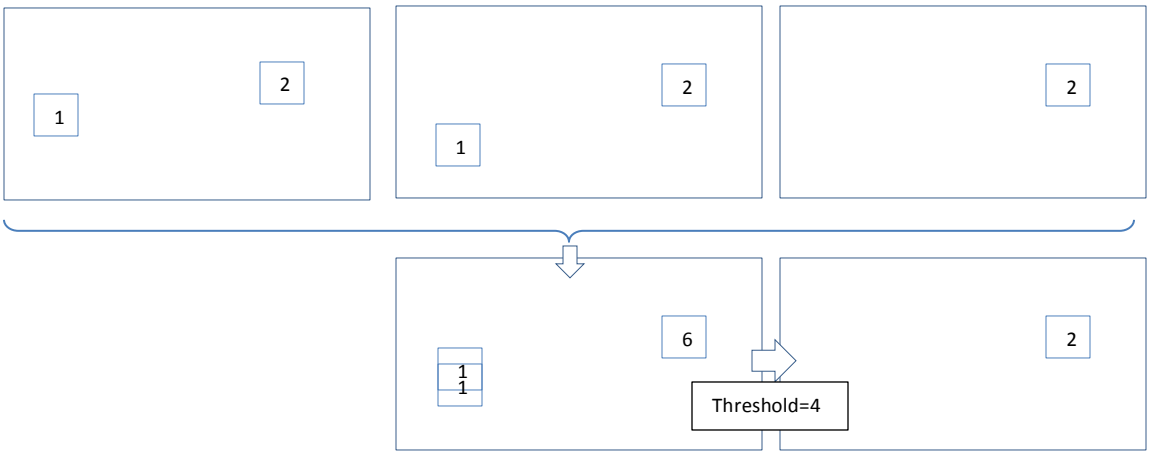
The heat map is built by simply adding up all the duplicated detections. We then apply heat threshold to filter out the false positive detection (e.g., the car in the opposite direction, in the left of the image below).

Images of overlapping detection and resulting heat map:



Below are the images of thresholded heat map and the car position.

Still non-vehicle objects falsely identified by the SVM classifier as vehicles appear to move across different video frames, and vehicles coming from opposite directions do not have relatively stable position in the image, either. To filter them out, a heat map queue defined by `class Heatmaps()` buffers the heat maps of a few previous video frames. When a new heat map is built, it is added to the sum of the heat maps in heat map queue. This allows setting a high heat threshold that can more easily filter out the above two types of false positives while keeping the right vehicles in the video frames. See below the illustration.



Finally, we need to find all the blobs on the threholded heat map by the `label()` function imported from `scipy.ndimage.measurements`:

```
labels = label(heatmap)
```

labels is a 2-tuple, where the first item is an array of the size of the heatmap input image (1 for all the pixels of the 1st blob, 2 for all the pixels of the 2nd object, etc.) and the second element is the number of blobs (cars) found.

8. Plotting Bounding Boxes

This function is implemented by `draw_labeled_bboxes()`. It goes through all the labeled blobs on the heat map that have been indexed by `label(heatmap)`, and plots the bound boxes around those blobs using the minimum top-left and maximum bottom-right coordinates of these blobs as the two diagonal vertices of the final rectangular.

9. Video Implementation

The `CarDetector` pipeline results in a video file called project_video_out.mp4, which is in the submission package.

In order to produce a video that combines vehicle detection and lane detection, I need to import the code of `LaneDetector` that I developed in the last project (P4). However, Jupyter Notebook does not provide a direct way for importing the code from another Notebook file.

There is a solution, though, at this URL: http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Importing%20Notebooks.html. The filename of LaneDector is LaneDetector4CarDetector.ipynb.

After LaneDetector is successfully imported, the new pipeline starts from CarDetector whose output image frames with detected vehicles are processed by LaneDector which then detects and adds lane lines to the video frames.

```
Cardetector = CarDetector()          # Create a car detector
Lanedetector = LaneDetector()        # Create a lane detector
clip = VideoFileClip(video_input)    # Read in video clips
carclips = clip.fl_image(Cardetector.img_process)  # Add cars
laneclips = carclips.fl_image(Lanedetector.img_process)  # Add
lanes
laneclips.write_videofile(video_output, audio=False)  # Write
video file
```

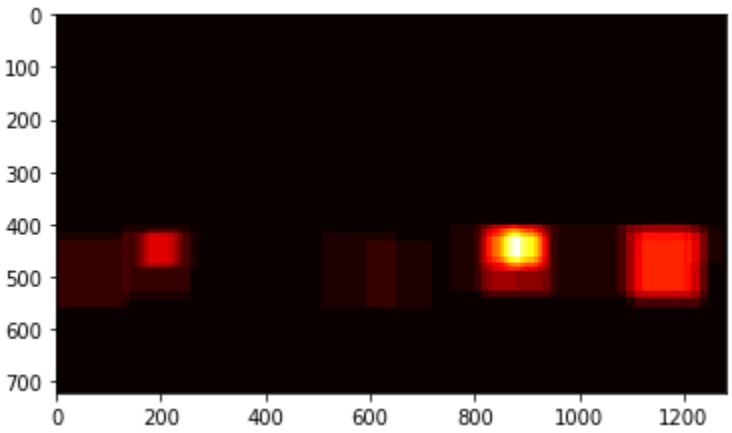
10. Issues and Discussions

10.1 False Positives

Most of the time, false positives are the vehicles coming from opposite direction. They are classified by SVM classifier as vehicles even though none of the example vehicle images in the training dataset are from front views. It's practically hard to

distinguish vehicles from opposite direction as they have similar look to the cars from rear view.

The solution to this problem is to carefully select scales and heat threshold so that false positive vehicles or other non-vehicle objects have lower heat values than the heat values of right positive vehicles. For example, see below:




```
Max Heat from left to right: [30.  6.  93.  39.]
```


```
Threshold = 30
```

The vehicle detector with the current parameter values may not be able to exclude all the false positives in the cases where the lighting and road conditions change rapidly. To be more robust in false positive avoidance, the SVM classifier needs to be trained by larger dataset, the detector needs more tests and parameter fine tuning. Maybe even more powerful method than SVM classifier over HOG-based shape features combined with color features is required, such as neural networks over some other features that are invariant to lightness, contrast, color changes, etc.

The paper "*Object Detection for Autonomous Vehicles*" by Gene Lewis provides a good summary of some of the popular neural networks that are widely used for object detection from which we can draw some better ideas on false positives. While the "heuristic" false positive suppression method, called "Horizon Suppression" proposed by the paper, may work well on the false positives located above the horizon line in the image plane of the video camera, it may not work well on the false positives located below that line.

10.2 Bounding Boxes

Currently the bounding boxes are defined fully by the collection of the contours of sliding windows that found the blobs in the video frames, which are sometimes wobbly and do not look very elegant due to abrupt and random change of the sizes of the bounding boxes, even though summing up the bounding boxes of previous frames may mitigate the wobbly effect. Apparently, better bounding boxes with the sizes that just fit the vehicles in the images are needed. There are a few approaches already developed.

One of them assigns bounding boxes of different sizes to the grids in the image. When "center of mass" of a blob falls in a grid relatively far away from the bottom of the image, it is perceived to be "in distance" and is thus supposed to have small size, while on the opposite, a blob near the bottom is supposed to have large size in the image. As such, bounding boxes of adequate sizes are used instead of the current sliding window-based boxes.

Another approach is to learn the right sizes of the bounding boxes from examples using neural networks. The paper "*Evolving Boxes for Fast Vehicle Detection*" by Li Wang, et al, presented a model with multiple neural networks working collectively to not only detect vehicles but also decide the sizes of the bound boxes that should be used to surround the detected vehicles.

A more sophisticated approach is semantic segmentation. Semantic segmentation tries to assign the category labels to each pixel in the image, which not only identifies the objects (vehicles in our case) but also identifies the shape and exact location of the objects. Semantic segmentation is a well-studied topic. A good article on this topic is "*Beyond Bounding Boxes: Precise Localization of Objects in Images*", by Bharath Hariharan.

To sum up, the work on vehicle detection project has fulfilled all the requirements, and it covers all the rubric points in this report. Furthermore, the final video has both lane detector and vehicle detector combined, making it even closer to the real world applications.