Udacity Self-Driving Car Nanodegree

Project 5 - Vehicle Detection and Tracking

Contents

1 Reference to Project Code and Files

Here is a link to my *project code and files*.

My project includes the following files:

- **Project_5_final.ipynb** notebook containing the code (*link*)
- Test **images**: 2 with straight lines, 6 with curved lines (*link*)
- Test images **after** image processing: same name with "_boxes_1" or "_boxes_2" suffix (*link*)
- Test **video** (*link*)
- Test video **after** image processing: same name with "_boxes_2" suffix (*link*)
- **Project_5_writeup.pdf** summarizing the results (*this memo*)

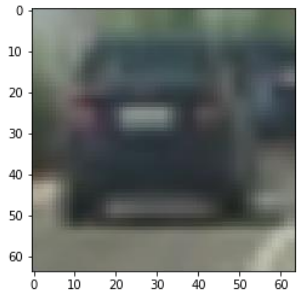
2 Building training set

This part is covered by the *first section* of our Python notebook (*cells 3-5*).

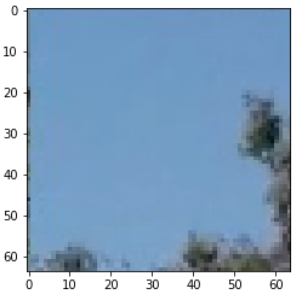
Here we are extracting sample images of **vehicles** (8,792 images) and "**non-vehicles**" (8,968 images). All sample images are **64x64x3** pixels. Those images are mostly coming from the GTI vehicle image database and the KITTI vision benchmark suite.

Here are examples of our training set:

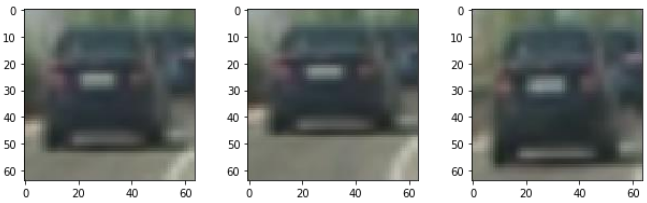
Vehicle:



Non-vehicle:



The number of samples in each class is **well balanced** (2% difference in count), so we do not need to create more samples. Also, it does not look like **data augmentation** such as translation or rotation is required here, since the same image is often repeated from various points of view. Example here below:



3 Extracting features from images

This part is covered by the *second section* of our Python notebook (*cells 6-12*).

The key component here is our **feature extraction** process, which relies on the three following functions:

- **bin_spatial**:
    - This function transforms images into a flat vector of pixel values.
    - As a result, it focuses on spatial features, i.e. **where** objects are located in images.
- **color_hist**:
    - For each color channel, this function stacks pixel values in histograms.
      So it insists on the relative intensity of **colors** with respect to each other.
    - The underlying assumption here is that vehicles detach themselves from the background thanks to their color.
    - Depending on the selected color space, this method captures different color characteristics. For example, under the HLS space, vehicles will probably distinguish themselves from the background thanks to more saturated colors (S channel).
- **get_hog_features**:
    - This function relies on the HOG function, for "**Histogram Oriented Gradient**", from the scikit-image library.
    - This method consists in computing gradient orientation for different regions of images (8x8 pixel blocks in our model) and assigning the output to histograms.
    - After experimenting different values, we chose **nine target bins** for the different gradient orientations. This function is probably the closest to human eyes in our recognition process, since it focuses on **shapes**.

The **extract_features** function combines the three aforementioned functions to build a flat **feature vector** as an **image signature**, to be fed to our classifier.

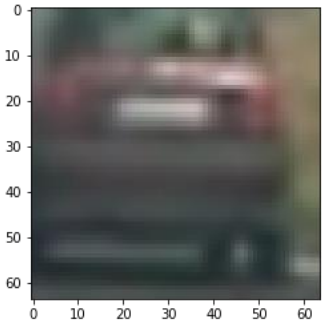
Finally, the **extract_features_all** function applies our feature extraction process to all images from the training set:

- To "car" images. Car images are assigned the **1** label.
- To "non-car" images. Non-car images are assigned the **0** label

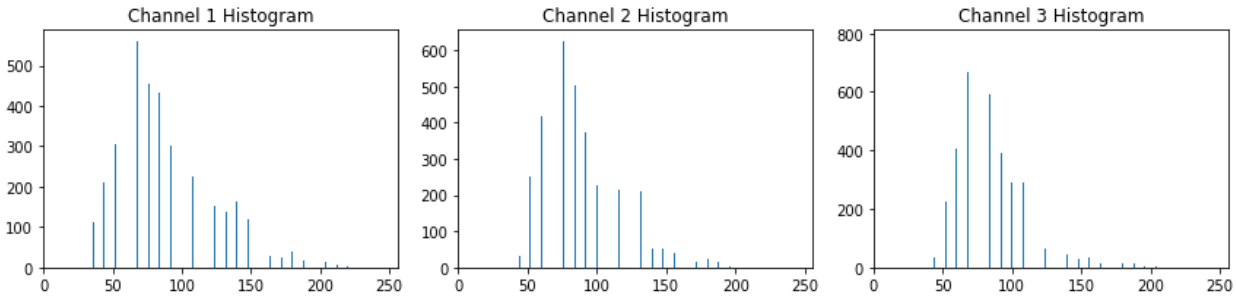
All extracted features are then **normalized** using the Scikit Learn **Standard Scaler**.

Below are some **visualizations** of the aforementioned functions, to illustrate their respective contribution:

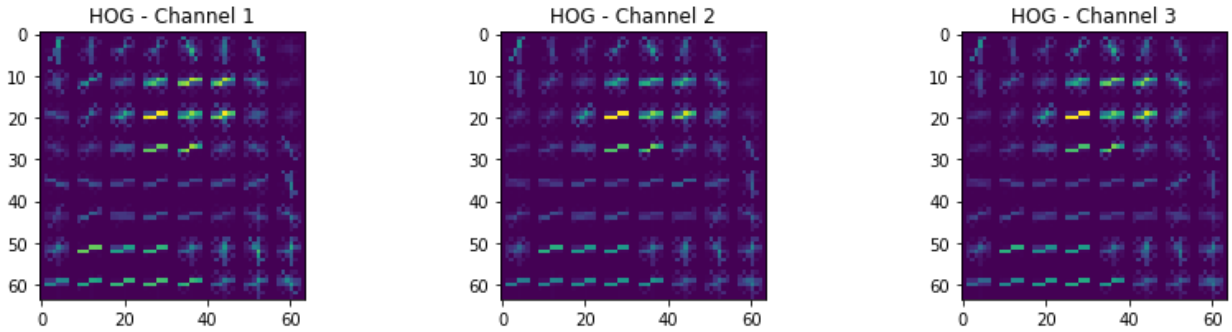
- **Raw** image:



- **Distribution** of pixels for each color channel (output from **color_hist** function):



- Visualization of oriented gradients for each color channel (output from **get_hog_features** function):



After multiple iterations, we are retaining the following parameters as optimal for feature extraction:

- Color space: '**YCrCb**' (color_space variable)
- Spatial size for spatial binning: **16x16** (spatial_size variable)
- Number of bins for color histograms: **32** (hist_bins variable)
- Number of bins for gradient orientation (HOG features): **9** (orient_bins variable)
- Number of pixels per cell for HOG features: **8** (pix_per_cell variable)
- Number of cells per block of cells for HOG features: **2** (cell_per_block variable)
- Color channels selected for HOG features: **All** (hog_channels variables)

Our feature extraction process results in **6,156-long** vectors. This is a high number of features, resulting in significant processing times. Our process could be **optimized** without a major loss in accuracy, namely by:

- Reducing the number of orientation bins to 8.
- Using the HLS color space and focusing on the S channel (saturation).

4 Building vehicle / non-vehicle classifier

This part is covered by the *third section* of our Python notebook (*cells 13-15*). In this section we are:

- Splitting the training set between **training** and **test** images.
- **Training** a Support Vector Machine classifier: after comparing a Naive Bayes and **SVM classifier**, we concluded that the SVM offered the best combination of accuracy and execution speed.
- **Testing** our classifier.
- **Saving** our classifier along with feature extraction parameters to a pickle file, for further use.

Our classifier delivers **98.8% of accuracy** on the test set, which is an acceptable performance for vehicle detection. On the first 15 samples from the test set, 100% of images are correctly predicted.

5 First approach for vehicle detection

This part is covered by the *fourth section* of our Python notebook (*cells 16-22*).

For a given image from a video stream, the approach consists in:

- Focusing on the **lower portion** of the image as a region of interest, where cars are more likely to appear.

- Dividing the region of interest into **windows** of different sizes: the **higher** in the region of interest, i.e. the **further** from our camera point of view, the **smaller** the scale. Windows are built byt the **slide_window** function.

- **Scanning each window**, resizing it to **64x64** (size of images in training set) and run our **classifier** to detect cars or not. If our classifier returns a positive detection (label=1), we are appending the window to a list of positive detections. This is achieved by the **search_windows** function.

- **Drawing** all positive windows on the image, using the **draw_boxes** function.

After getting all windows with positive detections, we are filtering out **false positives** and merging **duplicates**. This is achieved by the **bounding_boxes** function, as following:

- Initializing a **heat map**, with the same size as the original image, featuring zeros for each pixel position.

- Iterating through positive windows and accumulating 1 for each pixel position within windows. This is achieved by the **add_heat** function. The same car is usually detected **several times** by different windows. So when there truly is a car, windows are **overlapping**, which results in pixel positions assigned with **more than 1** as a tag.

- Retaining only pixel positions above a given threshold, using the **apply_threshold** function. This step normally gets rid of **false positives**, since they usually are not "detected" more than once or a limited amount of times.

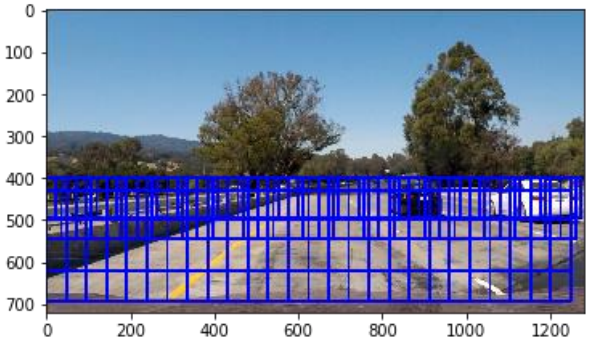
- **Merging windows**:

    - This step relies on the **label** function from the scipy.ndimage.measurements library (link).

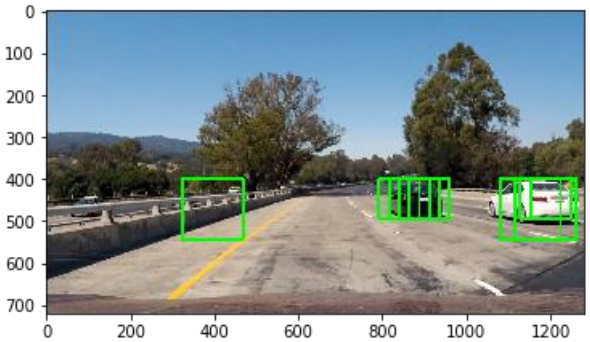
    - The Scikit documentation does not explain the underlying theory, so checking this link might be helpful here. Essentially, the label function allows connecting **adjacent regions** on a given map: each region with common pixels is considered a "label".

    - We are leveraging this function to connect overlapping windows together. Then we use our **draw_labeled_bboxes** function to retain only minimum and maximum x and y pixel positions, which result in merging overlapping windows.

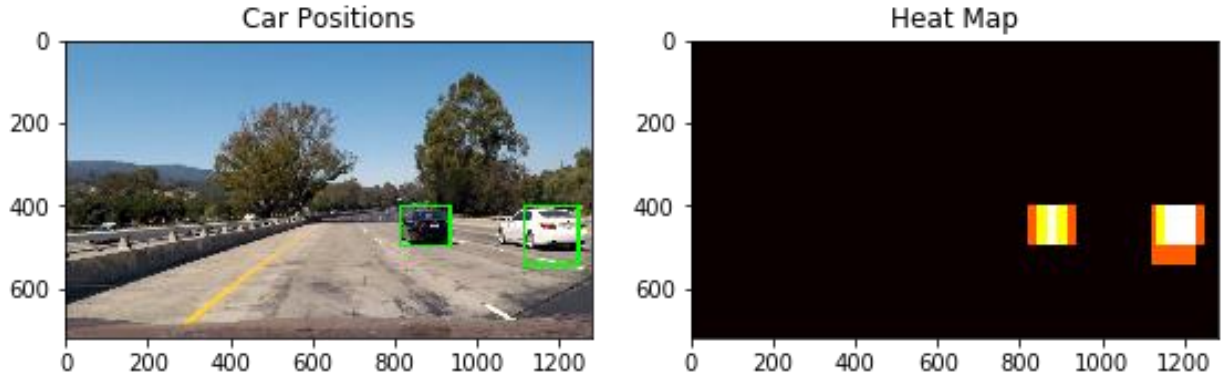
Below is a visualization of search windows as generated by the **slide_window** function for a given image. We can see how windows are getting smaller as we go from the bottom to the top of the lower portion:



Below is the output of our **search_windows** function for the same image. Here we are clearly seeing overlapping windows, along with a false detection on the security rail (left-hand side of image):



After applying our **bounding_boxes** function featuring a heat map, the false positive disappears and true positive windows are merged together:



6 Alternative approach

This part is covered by the *fifth section* of our Python notebook (*cells 23-25*).

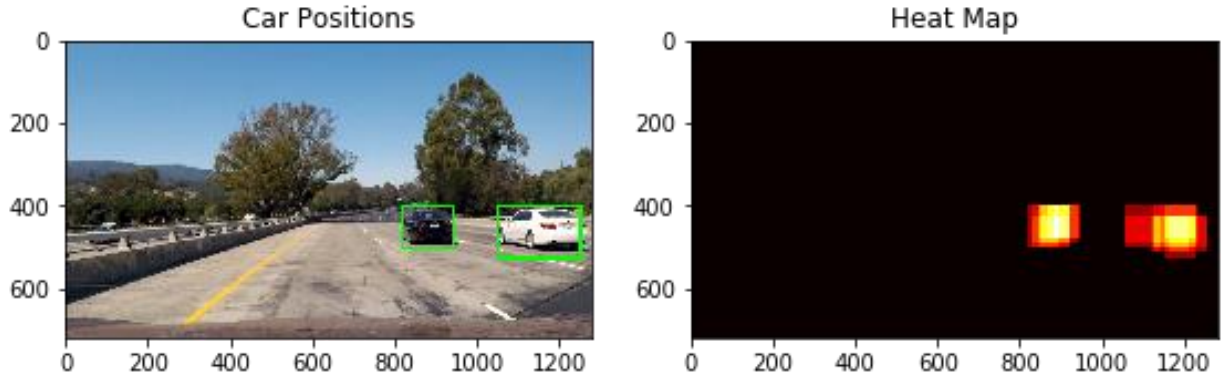
This approach is similar to the previous one. We are still scanning images with a sliding window, but this time our **find_cars** function allows **mutualizing the computation of HOG features** instead of computing them for each individual window.

The find_cars function does not require generating search windows beforehand. It **calculates windows** dynamically, given an incremental step from a window to the next (cells_per_step argument).

Also, our function allows applying a different scale to different regions of images.

Below is find_cars's output for the same image as above:

Then after merging duplicates and eliminating false positives (using the bounding_boxes function):



7 Implementing final pipeline

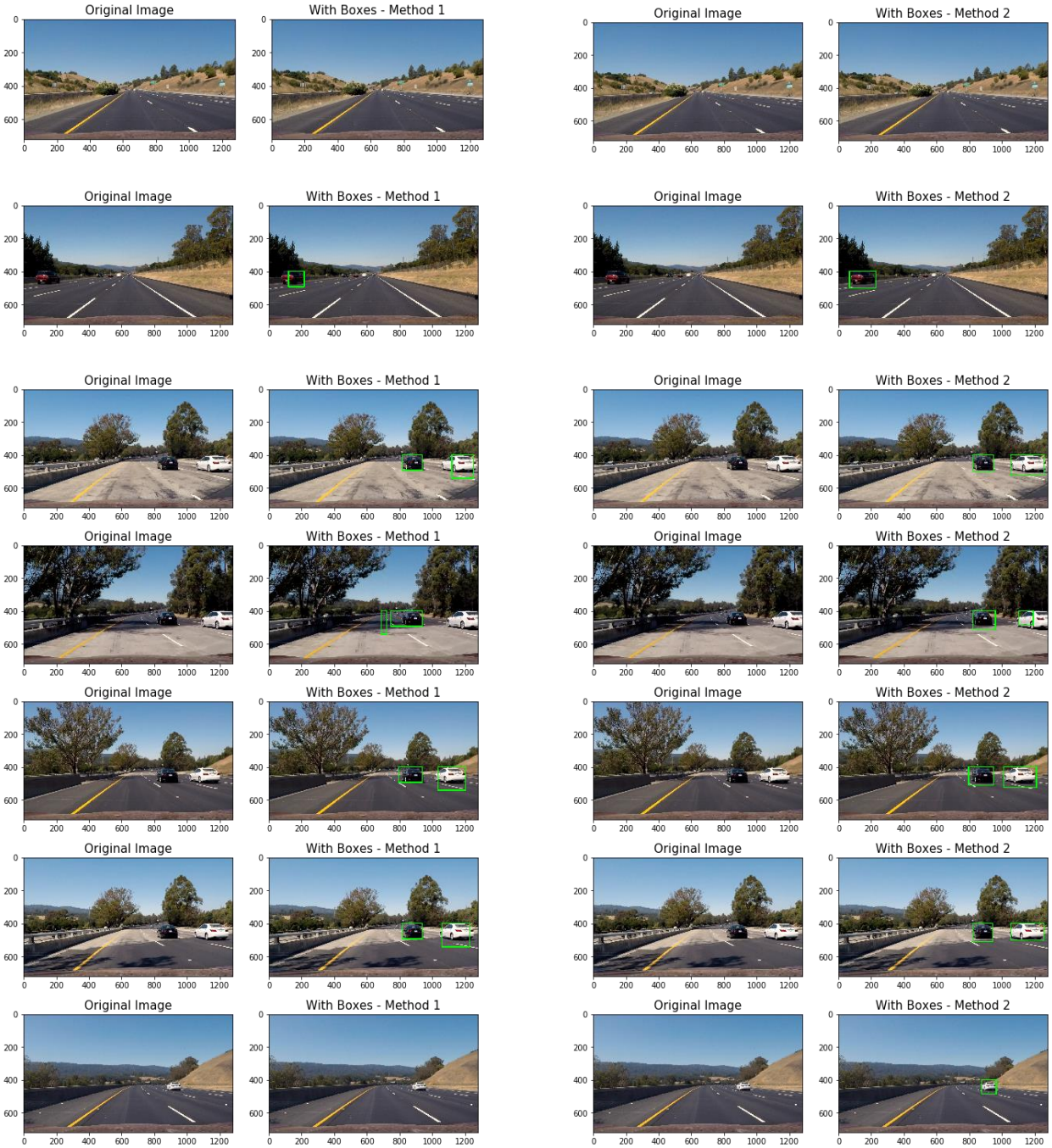
This part is covered by the *sixth and final section* of our Python notebook (*cells 29-31*).

According to our test images (*see images below*), our **alternative method** seems to do a better job at **detecting** cars, removing **false detections**, as well as **fitting** cars in boxes.

It is mostly due to our **variable scales** that are better tailored to the different regions of images.

As a result, we are retaining our second method as a default setting in the **process_image** function (argument method=2 by default).

Below are test images with vehicle detection according to **method 1** (original approach) and **method 2** (alternative approach):



We finally apply our process_image function to the test **video stream**, which results in the [project_video_boxes_2.mp4] file.

8 Discussion

Looking at the final video stream, our pipeline has several areas of improvements:

- As mentioned here above in section 3, our **feature extraction** process generates **6,156-long** vectors, which involves significant **processing times**, especially for real-time treatment.

  We could **optimize** our pipeline in at least two ways:

  - By reducing the number of **orientation bins** to 8.
  - By using the **HLS** color space and focusing on the **S channel** (saturation).
  - After experimenting those options, we found out that they did not significantly deteriorate our classifier's accuracy. But they must be paired with the **next topic** here below.

- Our pipeline does a satisfying job at detecting vehicles but our bounding boxes are permanently readjusted, which results in **discontinuous** / **blinking** rectangles.

  We could mitigate this effect by:

  - **Smoothing bounding boxes** from a given image to the next.
  - To do so, we could create a **Box class** to save the n last detections and draw boxes based on averaged edges. We could also consider putting a **higher weight** on **newer detections** to better track vehicles' movements.

- Finally, we could possibly derive **vehicles' speed** by leveraging our Box class, and display this speed on the screen.