CarND-Vehicle-Detection WriteUp

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

Vehicle Detection Project

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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

Histogram of Oriented Gradients (HOG)

1. Explain how (and identify where in your code) you extracted HOG features from the training images.

I would like to introduce the two-part process of developing this solution: the exploratory data analysis, followed by the building of the model and video processing pipeline.

Please note that in the process of building the model, code refactoring may have left the exploratory section in the Jupyter notebooks (namely explore.ipynb) unrunnable, so I will reference its usage in both exploration and the final model.

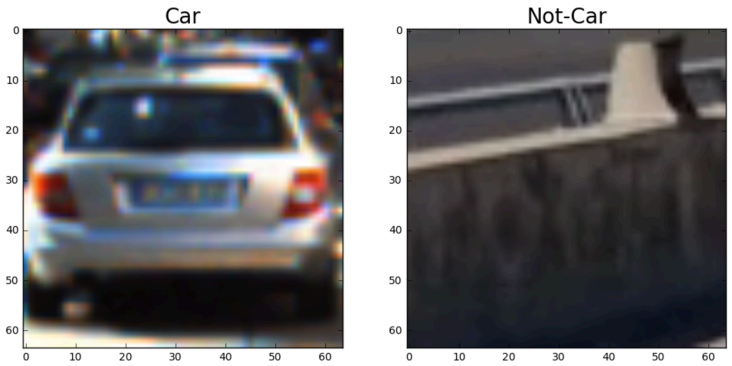
In the exploratory analysis, the section “HOG Feature hyper-parameter training” of explore.ipynb, hyperparameter exploration was conducted with prediction validation performed over a linear SVC model. The next section describes this in more detail. In preparation for this hyper-parameter tuning, a Pandas Data Frame was read in with the columns “category”, “subcategory” and “image”, as such:

	category	subcategory	image
0	vehicles	KITTI_extracted	training_images/vehicles/KITTI_extracted/1.png
1	vehicles	KITTI_extracted	training_images/vehicles/KITTI_extracted/10.png
2	vehicles	KITTI_extracted	training_images/vehicles/KITTI_extracted/1000.png
3	vehicles	KITTI_extracted	training_images/vehicles/KITTI_extracted/1001.png
4	vehicles	KITTI_extracted	training_images/vehicles/KITTI_extracted/1002.png

This data was shuffled and split into ‘train’ and ‘test’ (i.e. validation) sets. Furthermore, in order to reduce validation set correlation error, the data was further filtered into only using the KITTI_extracted subcategory data because the other data sets had unlabeled time lapse information. Manual splitting of the validation/training data could have been performed such that similar images do not end up in both training/validation sets, but in the interest of time this was not done.

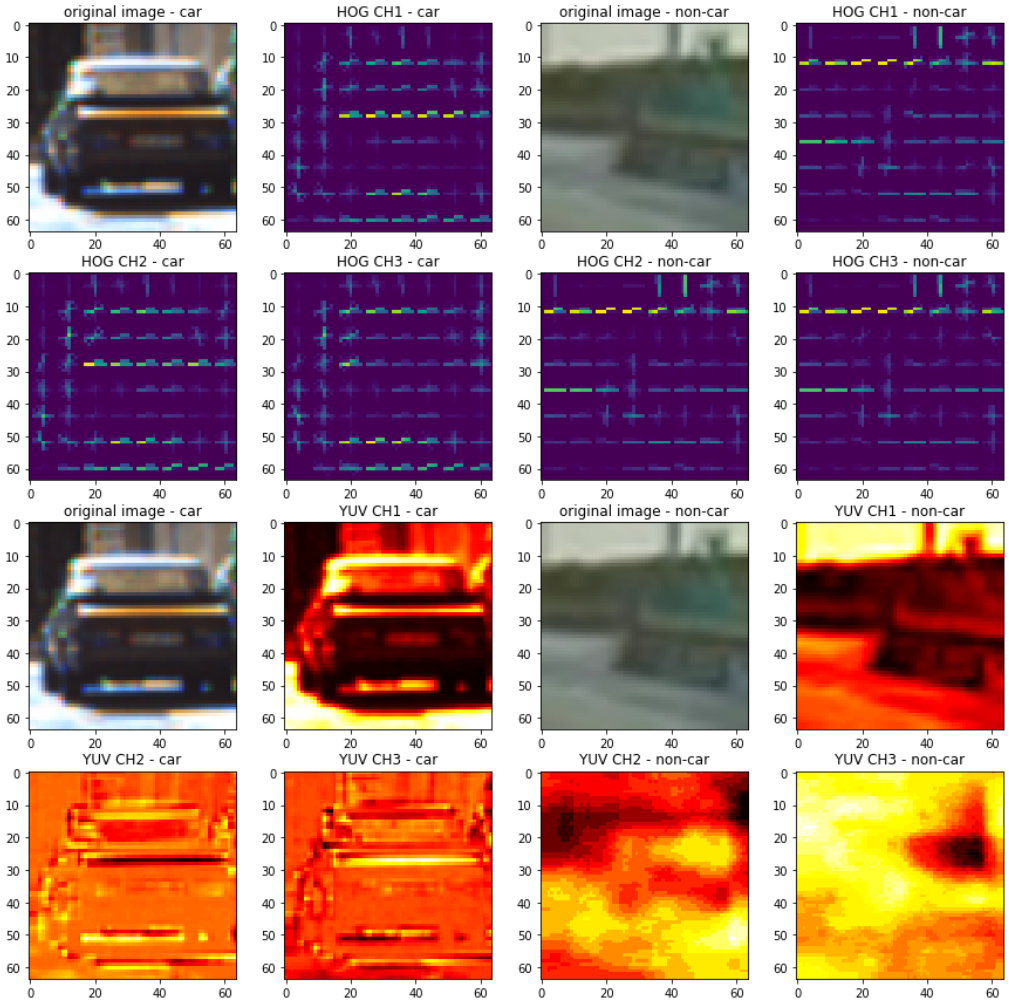
For training the video processing pipeline, all the data was used and includes the extraction of binned color and histogram features – this can be seen at model.py:#train_model (around line 14).

The purpose of building the model was to predict the presence of vehicles on the road – an image classification problem. The category labels `vehicle` and `non-vehicle`. Here is an example of one of each of the `vehicle` and `non-vehicle` classes:



I then explored different color spaces and different `skimage.hog()` parameters (`orientations`, `pixels_per_cell`, and `cells_per_block`). 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.

Here is an example using the YUV color space and HOG parameters of `orientations=10`, `pixels_per_cell=(8, 8)`and `cells_per_block=(2, 2)`:



2. Explain how you settled on your final choice of HOG parameters.

In explore.ipynb, I started with an exhaustive search of hyperparameters and validation result optimization. I realized that some parameters can be tuned optimally with little correlation to other hyperparameters, so I prioritized these parameters to be trained and optimized first.

Beginning with some ballpark reasonable hyper parameter figures:

- Colorspace = RGB
- Orientations = 9
- Pixels per cell = 8
- Cells per block = 2

I optimized each layer *independently*, taking the optimal result from the previous stage as a basis for conducting the next search. Taking this approach ensured that training time did not blow up exponentially whilst allowed for a wide range of hyperparameter tuning.

This process yielded the optimal result with 0.9995 accuracy having:

- Colorspace = YUV
- Orientations = 12
- Pixels per cell = 8
- Cells per block = 2

Later on, through experimenting with processing the video, it was discovered that the quality of frames extracted differed vastly from the images used in the validation data from exploratory data analysis. Hence, colorspace was eventually settled to be 'HLS' instead, which performed better in generalizing for the video.

3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

I trained a linear SVM using the HOG features for all the channels of the HLS images, concatenated with color-binned images resized to 16x16 pixels and color histograms binned into 32 segments for each color channel. See model.py#train_model (around model.py:44-47) for more details.

Sliding Window Search

1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

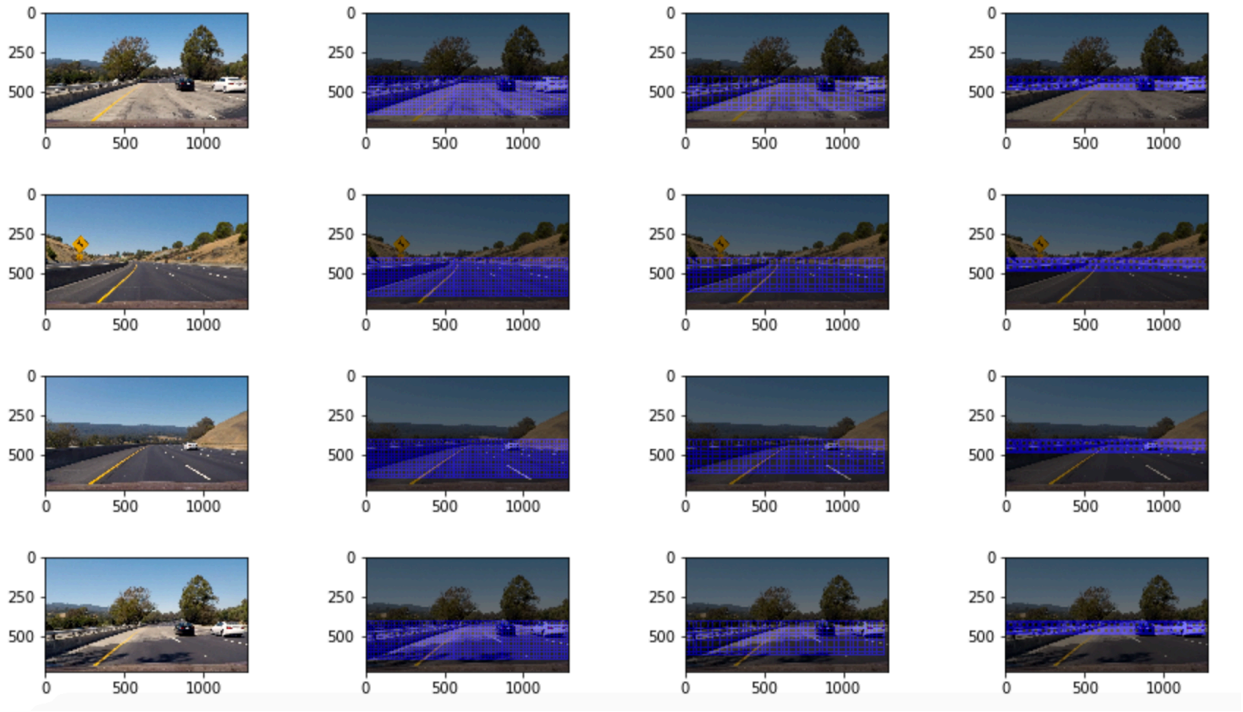
The sliding window search was coded in model.py:# get_sliding_window_preds (approximately lines 94-175).

The way it works is as follows:

1. Converts the input images to the correct color space
2. Crops the image to a relevant portion
3. Rescales the image to be 1 / *scalefactor*, in both x and y directions so that when features get extracted with a fixed window size, they in effect are extract relevant sections of the image at 1 * *scalefactor*.
4. We extract HOG features for the entire image, and then subsample HOG features for patches of that image. Because we need a fixed number of features to feed into the trained model, the patches that get extracted are forced to be 64x64 in size (like the training images), but in effect this is adjusted by the scaling factor of the previous step
5. cells_per_step was chosen to be 2, giving an overlap of 75% (regardless of scale)
    a. This is calculated by 2 / 8 = 25%, where 8 is the width of the window

The chosen scale factors were: 1, 1.5 and 0.75. These were done from a qualitative estimation approach – images closer to the camera require larger bounding boxes and also occur over larger areas of the screen – however, because they are larger, overlap scales proportionally and is thus less expensive. For this reason, 0.75 only searches the pixel region of y=[400, 500], whereas the 1 and 1.5 scale factors span the entire image's image from the horizon downwards (i.e. y=[400, 656]).

The scale factors could be chosen through visual estimation techniques – knowing that a scale factor of 1 is a 64x64 box, a sample image was overlaid in an image editing tool (e.g. Adobe Photoshop) to find scale factors that were reasonable.

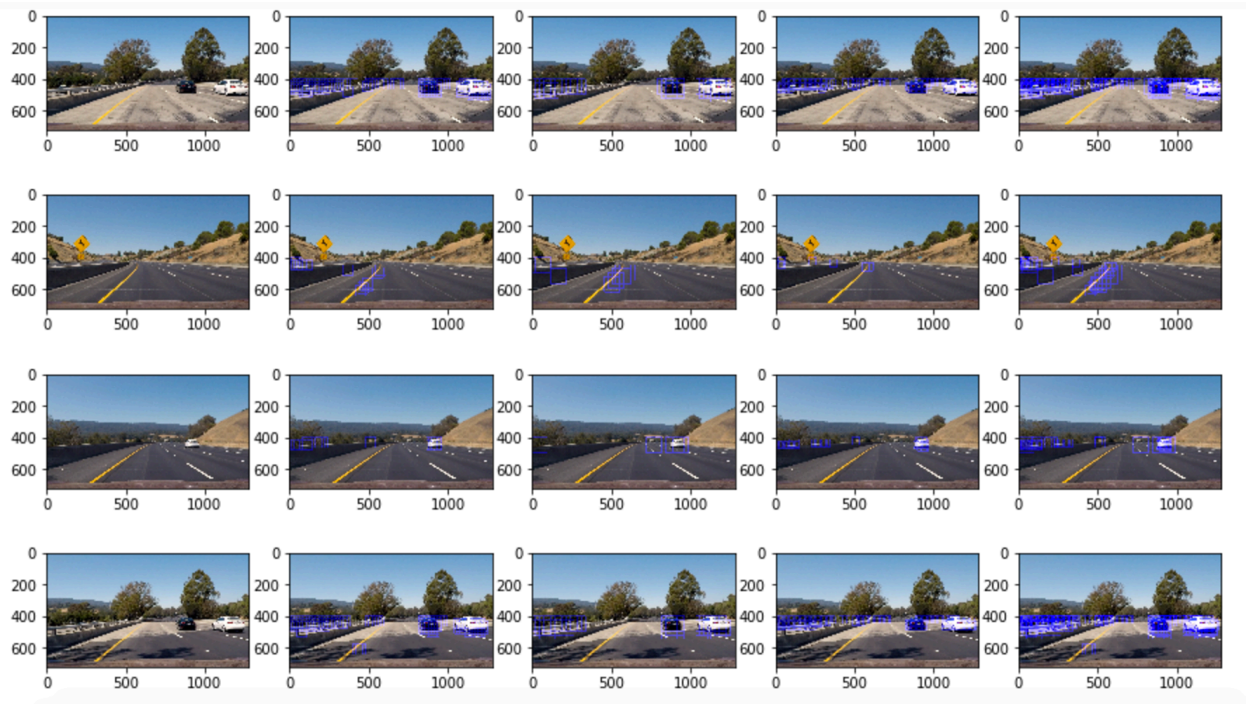


From left to right, the figures represent:

1. The original image
2. Scale factor 1x for y = (400 -> 656) scanning
3. Scale factor 1.5x for y = (400 -> 656) scanning
4. Scale factor 0.75x for y = (400 -> 500) scanning

2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

Ultimately I searched on three scales using HLS 3-channel HOG features plus spatially binned color (16x16) and histograms of color (32 bins per channel) in the feature vector, which produced a workable result, albeit with lots of false positives. Here are some example images:



From left to right, the figures represent:

- The original image
- Scale factor 1x for y = (400 -> 656) scanning
- Scale factor 1.5x for y = (400 -> 656) scanning
- Scale factor 0.75x for y = (400 -> 500) scanning
- Combined overlay of all the different scale factors

As mentioned earlier, the initial parameters of the model (e.g. colorspace, orientations etc.) for HOG extraction were optimized during the training process/hyperparameter tuning. However, given that the quality of the video is very different from the training images and the effects of resizing can cause various degrees of blurring, an important part of the process was to continue optimization on sample video output as well. Using segments of the video (not the whole video for speed purposes), the car detection pipeline was applied alongside smoothing and false positive detection logic and tuned according to the feedback obtained visual inspection of the results.

Video Implementation

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.)

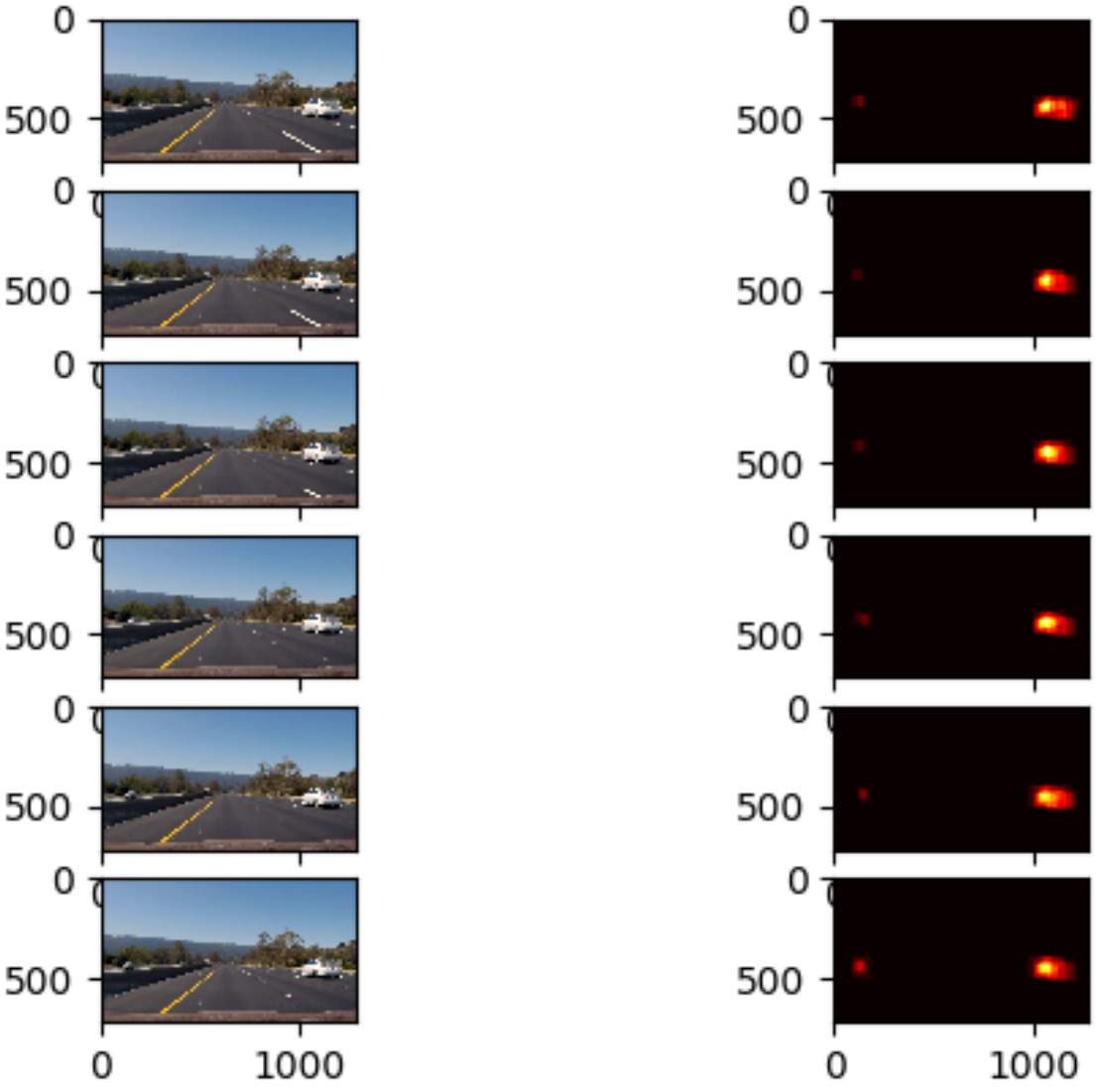
Here's a link to my video result

2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

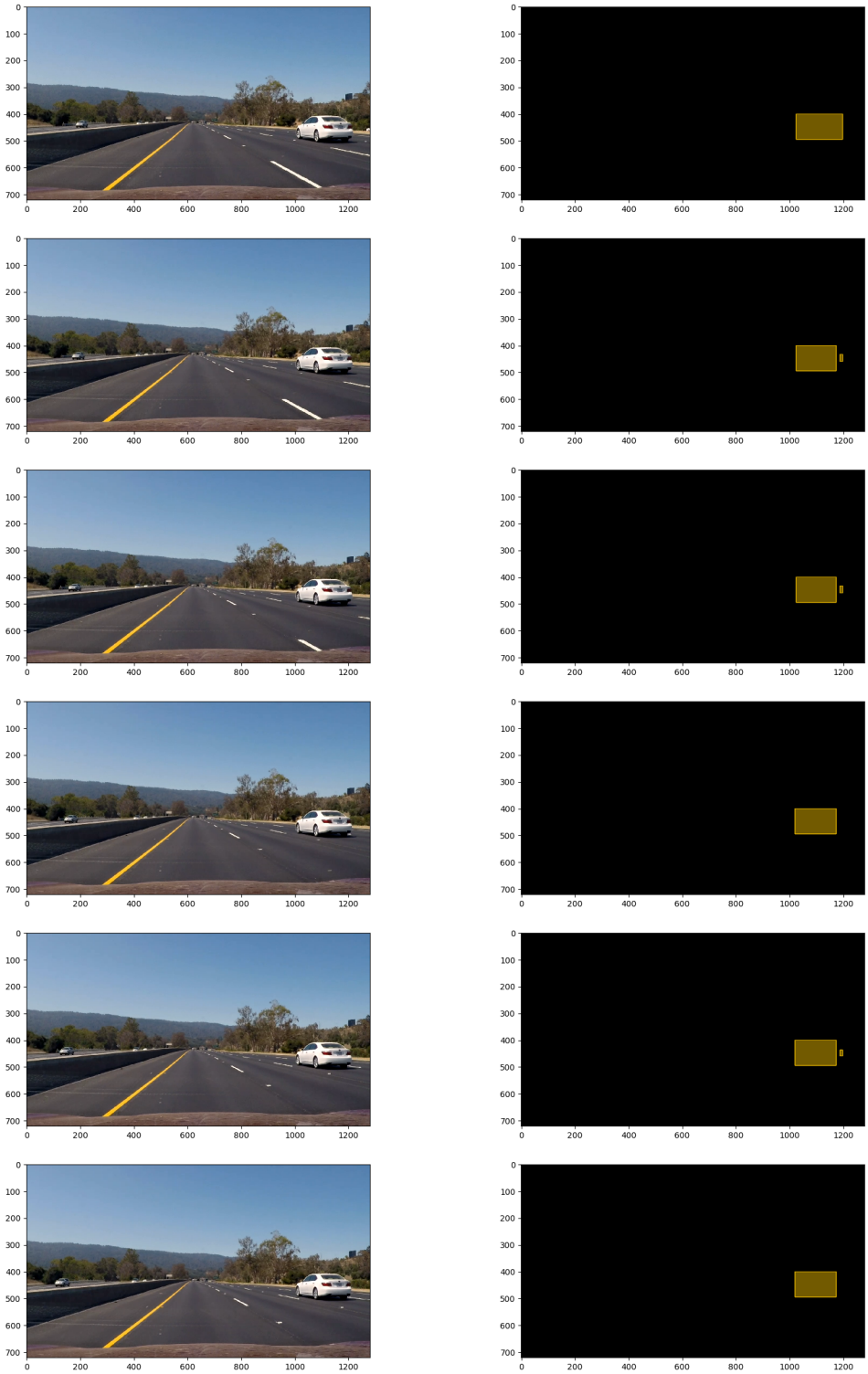
I recorded the positions of positive detections in each frame of the video. From the positive detections and generated heatmaps from them (model.py:# get_sliding_window_preds, approximately line 181). I then combined the heatmaps of several iterations of applying the sliding window predictions using different scales (main.py:#produce_heatmaps, approx. line 67) and applied a rolling average of the heatmaps for smoothing effect (main.py#main, approx. line 115) and removed heat from the map that fell below a specified threshold. I then used `scipy.ndimage.measurements.label()` to identify individual blobs in the smoothed heatmap (postprocess.py:# segment_cars). With the assumption that each blob corresponded to a vehicle, I constructed bounding boxes to cover the area of each blob detected and overlaid this on the original video frames.

Here's an example result showing the heatmap from a series of frames of video, the result of `scipy.ndimage.measurements.label()` and the bounding boxes then overlaid on the last frame of video:

Here are six frames and their corresponding heatmaps (with smoothing):



Here is the output of `scipy.ndimage.measurements.label()` on the integrated heatmap from all six frames



Here the resulting bounding boxes drawn on one of the frames of the output video:

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

The smoothing technique was done over the heatmaps, though locating the centroids of the detections and then doing a sanity check of the movement of the centroid as the frames progress would be perhaps a more powerful way to remove false positives. I would reduce the threshold of the smoothing of heatmaps and implement the centroid smoothing function going forward.

Additionally, I would like to see the model produce fewer false positives – perhaps augmenting the training data to have more examples of non-vehicle data so it can generalize better.