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 used all three channels of LUV colorspace to compute HOGs. Some short experiments showed that LUV (along with XYZ) outperformed other colorspaces and that using all three channels gave most robust results.

HOG features for training are extracted with `cars.processing.get_feature_vector()` on line 17. HOG features for detection in images and videos are extracted inside `cars.processing.get_single_scale_detections()` on line 164. Both approaches use same call to `skimage.feature.hog()` function. The difference is that for training HOGs are computed over a training sample only, while for detection over a large image and then split into subwindows in subsequent steps.

HOG parameters I chose were:

`pixels_per_cell` = (8, 8),
`cells_per_block` = (4, 4),
`orientations` = 9

as per suggestions from this talk by HOG's author. Initially I used original 64x64 size windows for features extraction, but later found that 32x32 windows still give acceptable performance while cutting down significantly on computational time.

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

First important parameter to establish was window size. 64x64 workes well, but leads to long computational time, about ~7sec per image on my machine when using linear SVM classifier.

Initially I used 64x64 windows, but once my whole pipeline was in place tried 32x32 windows and found that decrease in performance was reasonably small while cutting computational time by about 70%.

As for HOGs parameters, author of HOG states in this talk that HOG isn't too sensitive to parameters choice as long as you are in the general 4-12 pixels per cell, 3 to 8 cells per block and 6-18 orientations ballpark figure. I chose values roughly in the middle of recommended ranges without experimenting with them much. My linear SVM classifier had 97% accuracy and 3 degree polynomial classifier was close to 99%, so I was satisfied with results.

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 used a linear SVM with accuracy of ~97% for my initial pipeline. Once I completed the pipeline it was evident that SVM is the major bottleneck in my code and I tried both decision trees and bayes classifier hoping for faster alternatives, but they worked significantly worse than SVM (~82% and ~90% accuracy respectively).

Once my whole pipeline was in place I also tried polynomial SVMs and finally settled down on third degree polynomial SVM that seemed to offer best performance (very close to 99%) within a computational time I was willing to accept.

SVM is trained in `./scripts/train_classifier.py`.

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?

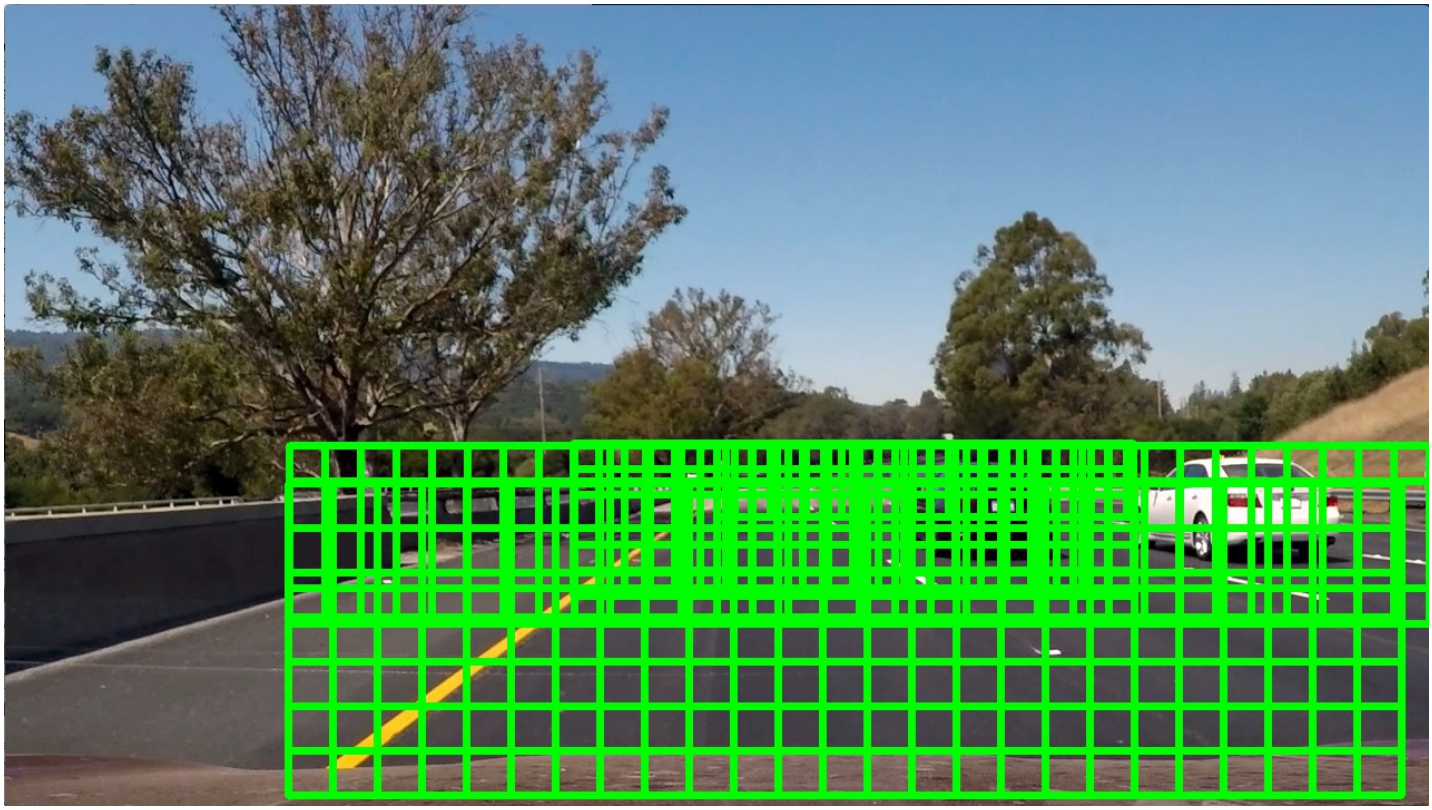
Since classifying a large number of windows is the major bottleneck in the code, some thought had to be put into trying to minimize number of searches. This can be done along four dimensions: number of scales to search, size of each scale, range across which to search at each scale and step size between windows at a single scale.

Seeing as cars in videos (at least nearby ones driving in same direction as test vehicle) span sizes roughly between 64~16- pixels (for vertical dimension when whole car was visible), and my classifier was trained to predict on 32x32 windows, I used scales from 0.2 to 0.5, which mapped cars to patches of sizes from 64 to 160 pixels. I also used two intermediate scales, for a total of four scales [0.2, 0.25, 0.35, 0.5].

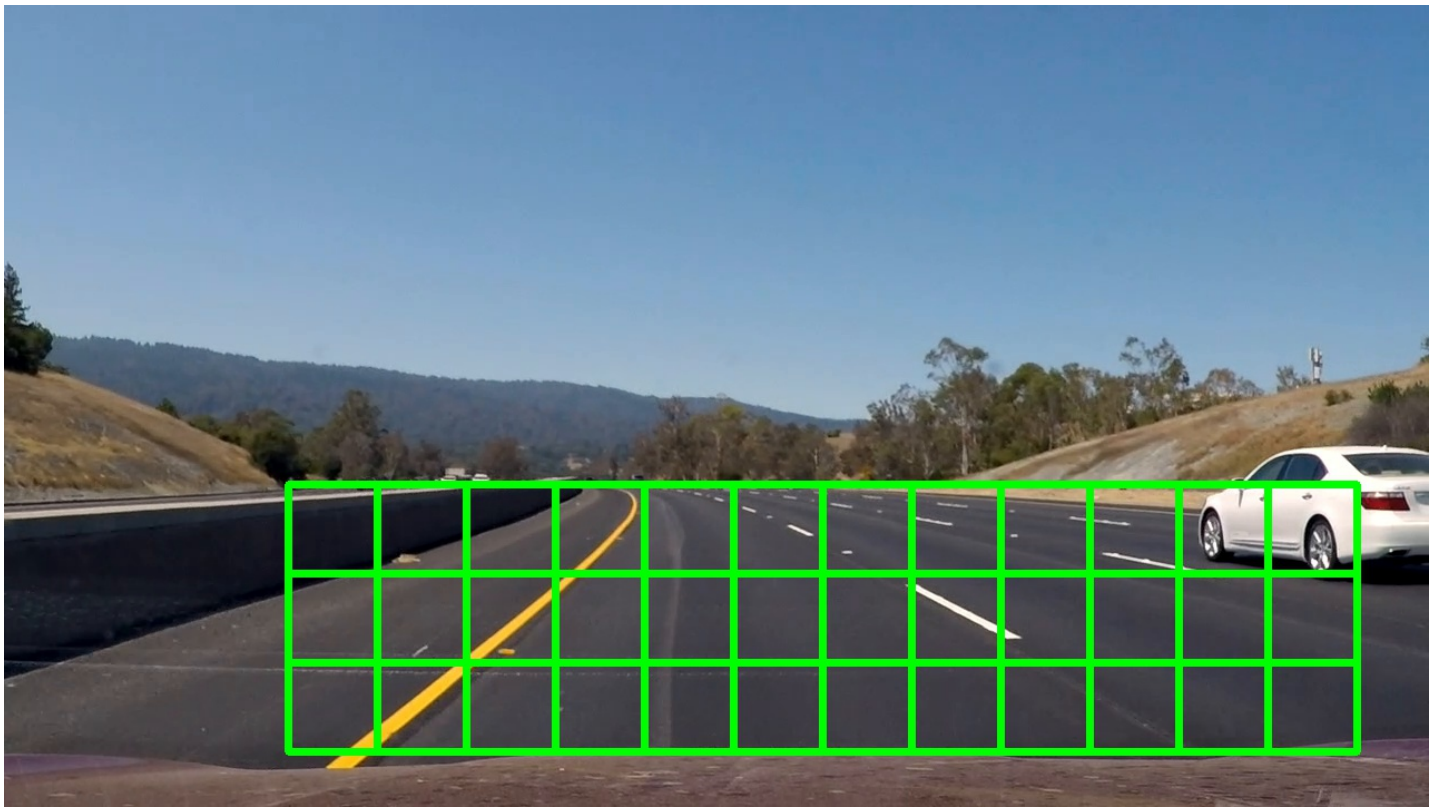
After experimenting a bit with window overlap I found that overlap of 8 pixels, gave best performance - computational time tradeoff for smallest scanning windows (64x64). For larger windows I kept the same overlap size, since that helped to make sure heatmap processing wouldn't cut out valid detections on ground of not enough overlapping detections. Since total number of large scale windows was small compared to small scale windows, small overlap there didn't contribute much to total computational time anyway.

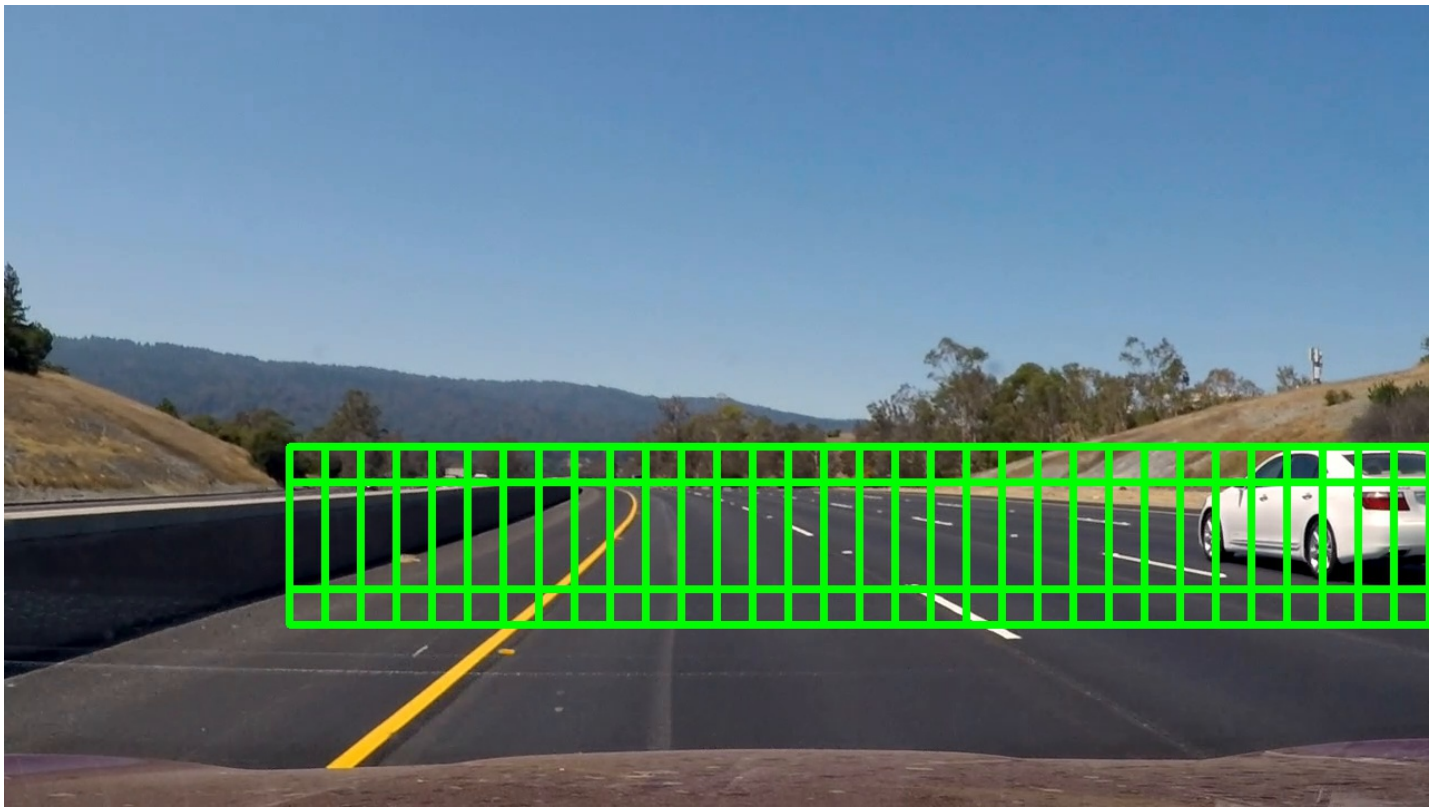
For each scale I defined search window within which it should be used so that we don't perform searches in locations where no vehicles are likely at a given scale. This was important to keep down computational time, since it's directly proportional to total number of evaluated windows.

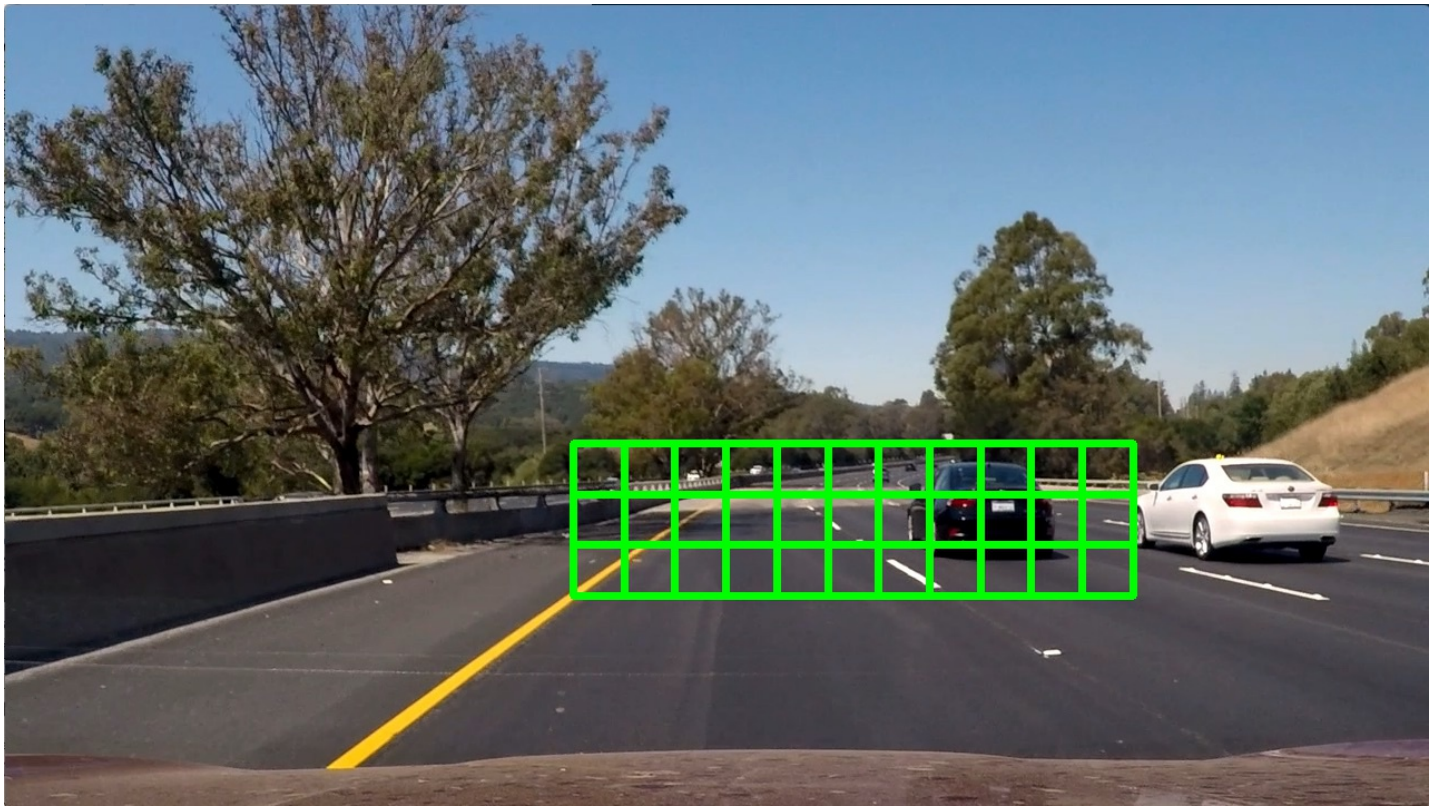
Below is a plot of the total search grid over all scales:



Since overlaying so many scales, especially ones spaced so densly, makes it difficult to get a good view of where is a search on each scale performed, below are plots for various scales I use individually. Window step size is increased, resulting in a less dense but easier to understand grid.







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?

While multiscale search provided "raw detections", these included many repeated detections for the same object as well as occasional false positives. Both problems were tackled with a heatmap that adds up detections over the same region and then discards detections that didn't get enough counts. That way multiple detections are merged into a single detection over region they overlap and many false positives get removed since they didn't get enough hits to be counted as true positives. Code implementing these steps can be found in `cars.processing.get_detections()` between lines 100 and 114.

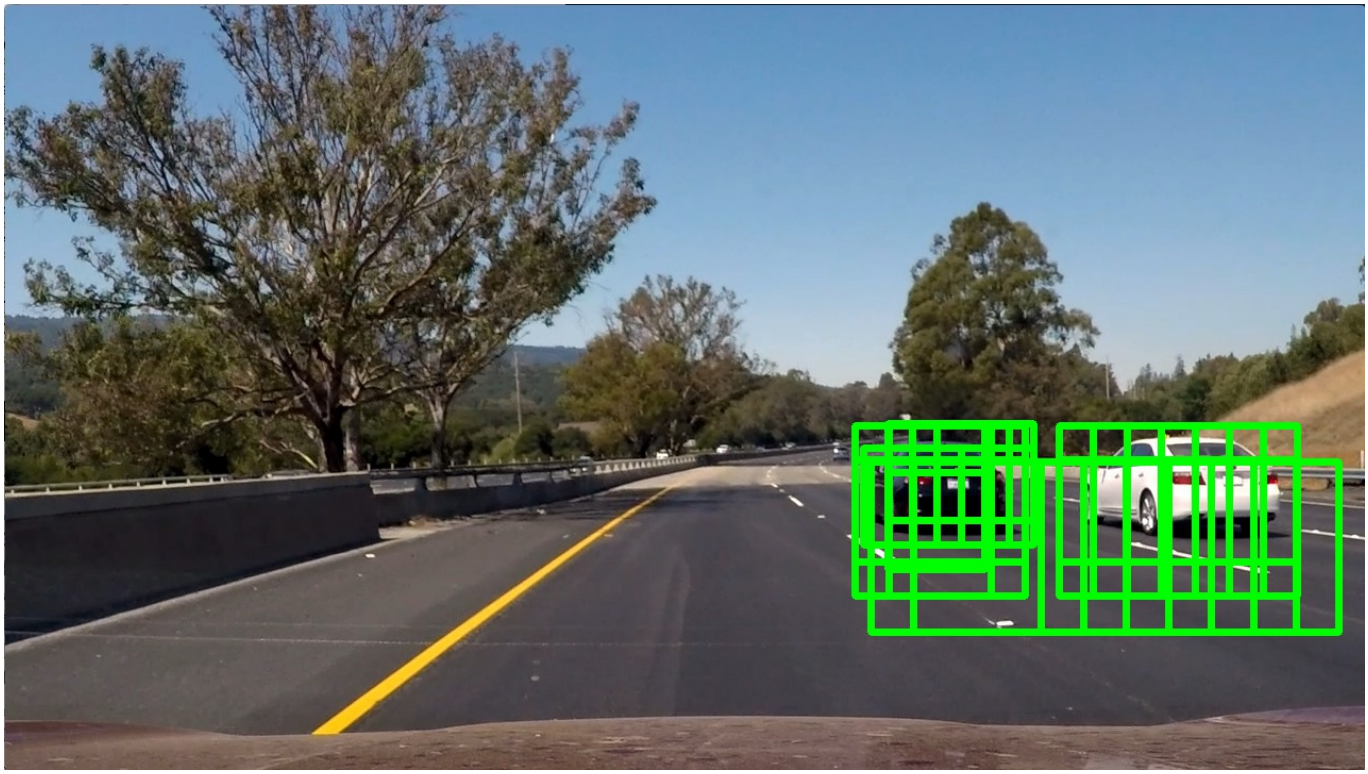
Above scheme works quite well, but tends to produce somewhat loose bounding boxes that span a a large margin around true bounding box of tracked object.

As a final check I added a simple filter that removes bounding boxes that are too small or have strange aspect ratios - this helps to tackle two types of false positives:
- funky detections resulting from two unrelated spurious detections thad a small overlap region that exceeded cutoff threshold
- detections of small metal objects that aren't cars - most notably distant road signs and occasional road barrier.

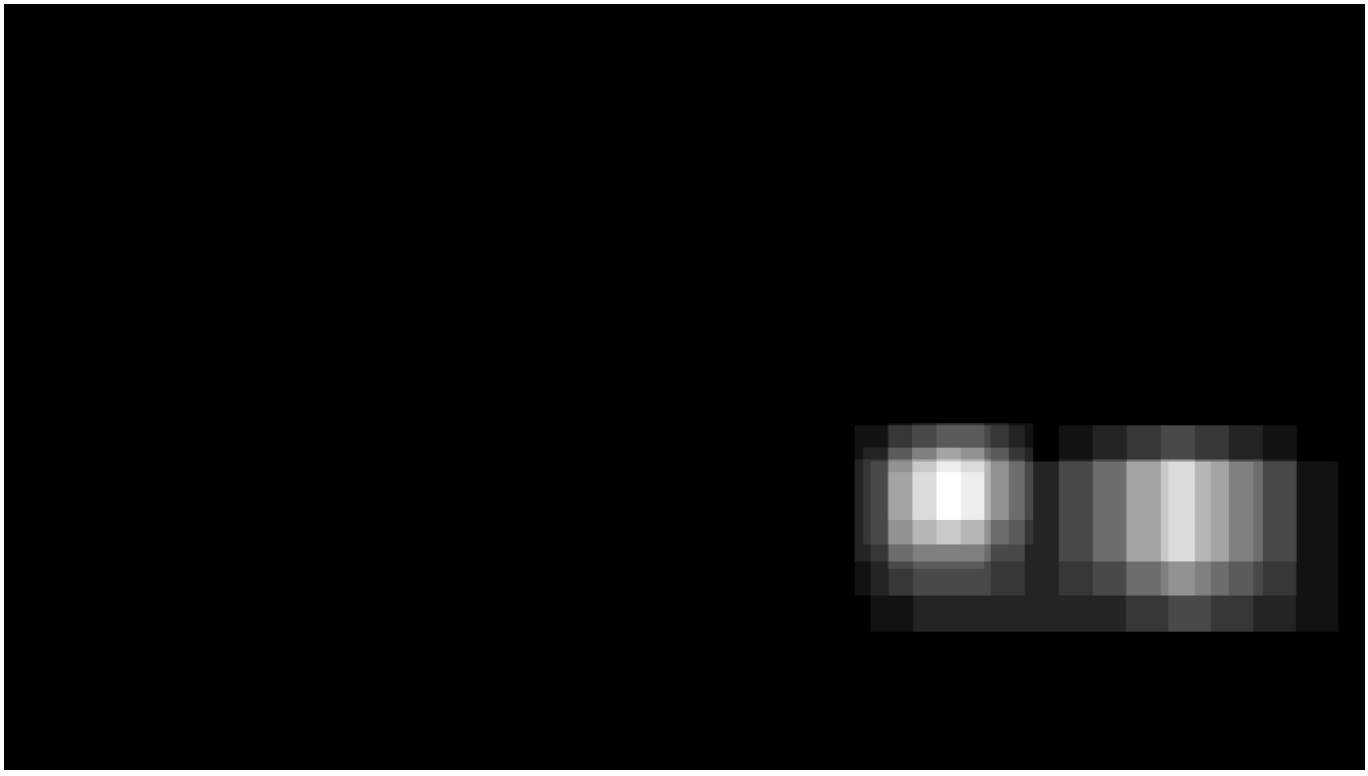
This filtering is implemented in function `cars.processing.is_detection_sensible()` on line 216.

Here are outputs from each processing stage for a single image:

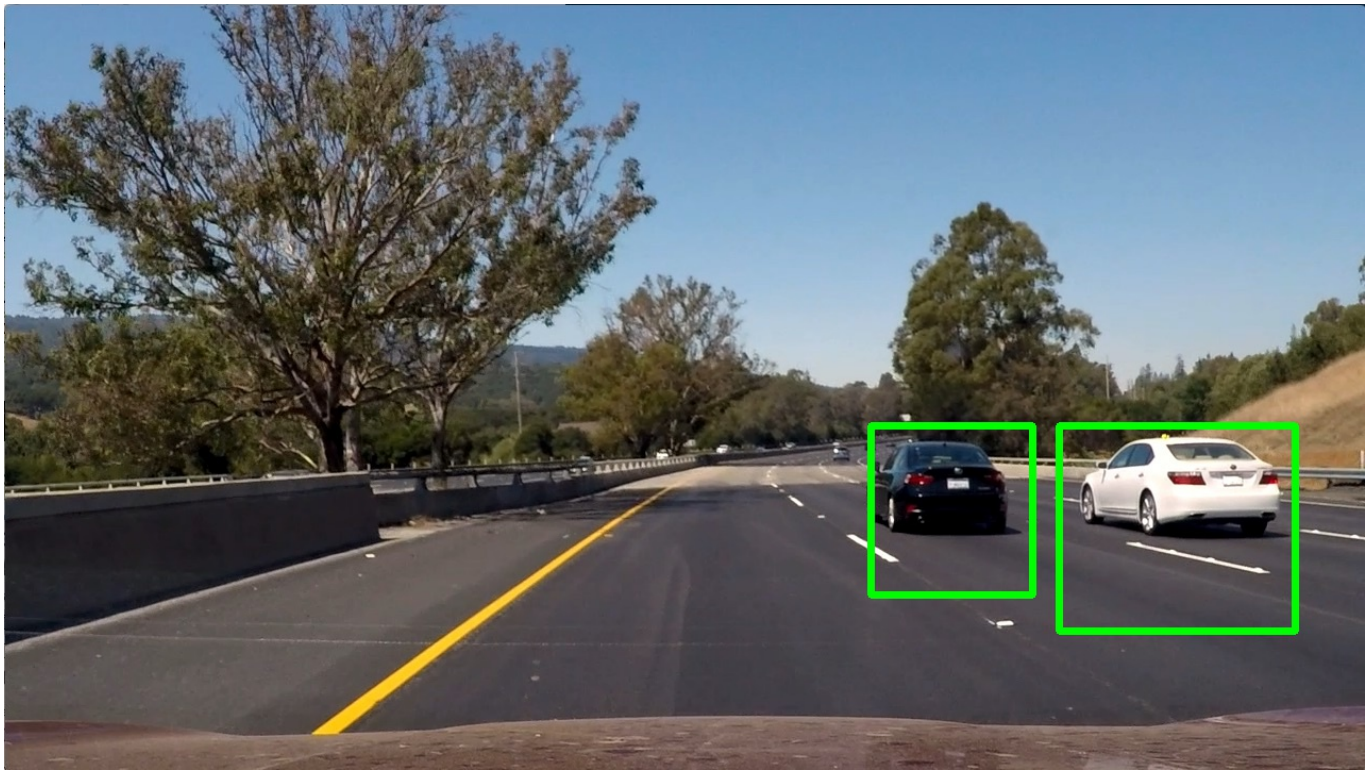
- Raw detections



- Heatmap



- Final detections for a few frames



The final stage of my processing pipeline, removing detections with unlikely sizes and shapes, is only useful once every dozen frames and as such it is difficult to grab a frame for which its influece shows. Its benefit is more obvious in videos.

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.

Part of postprocessing pipeline common to both still images and videos (heatmaps, merging multiple detections, rejecting detections that are too small or have strange size) is already addressed in previous section "Show some examples of test images to demonstrate how your pipeline is working".

For videos I added also postprocessing that tracks consecutive detections. This helps to stabilize detections and accomodate for both occasional misses of valid detections and occasional appearances of false positives.

At each frame detections are added to tracked detections list, along with a score representing in how many recent frames was detection found.

Detections from each new frame are cross checked with tracked detections using intersection over union. If it is high, detection from current frame is counted as a reoccurance of tracked detection, bumping up its "tracking count". If a previously tracked detection wasn't matched with any detection in current frame, its tracking count is decreased. Only objects with sufficiently large tracking count are displayed. Therefore each detection must appear in a few consecutive frames before it is displayed (hence very few false detections in video) and an occasional miss of a valid object in video won't remove the bounding box as long as object is detected again within the next few frames.

Code achieving above is included in `cars.processing.AdvancedVideoProcessor.process()` function on line 294.

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?

By far the main problem was computational time. My final pipeline takes 9 minutes to process 50 seconds of video at 12 fps. This would be of course a no-go in deployment (though multithreaded implementation in low lever language could probably solve that), but it also was limiting iteration speed during development.

As for likely failure points - ultimately the system is only as good as its underlying classifier. Classifier's accuracy is just under 99%, but that's because of "easy" test data. Non-vehicles data used for training and testing classifier is made up of random crops of road surface, road shoulders and trees. Empirically one can see that when sliding over whole image, trained classifier sometimes fails on man-made object with regular shape and angles, especially metal ones - so in addition to cars it sometimes detects road signs, barriers, etc. A training data that includes such data in negative class could help make the classifier better.

There is also a problem with positive images - they only include views of vehicles rears (although from varying angles). It's entirely possible that a car driving headway towards the camera might not be detected, since no forward facing car was pretent in positive dataset.

Another important point of failure is shape of scanning windows - a square is roughly ok for regular cars, but not necessary so for trucks and buses (which aren't present in training data anyway).

Finally current simple method of merging results using a heatmap is a rather quick and dirty solution. It requires using arbitrary numbers for detection cutoffs and tends to produce loose bounding boxes. This means a car driving perefectly fine might be detected as edging into adjacent lane, possibly triggering an emergency brake. A more sophisticated method for merging multiple that produces tighter bounding boxes would be necessary in production.