Vehicle Detection

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

Feature Extraction

The code for the feature extraction is contained in the file called `train_classifier.py`.

Reading in the data

The training data for the car feature is partly extracted from video. It thus appears almost identical in in a whole series of image, which makes the model tend to overfit easily.

I tried two different approaches to read in the data.

1) reading all directories together and using scikit's `train_test_split` to split up the data in train and validation sets 2) reading all directories together, using the first 90% of the data in each directory for training and the last 10% for testing

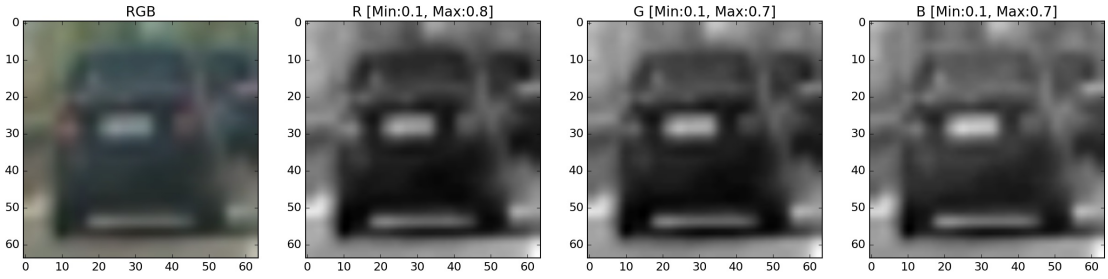
I did not manage to get any significant improvements in the window detection later on with with splitting the data manually, so I decided to drop that approach.

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

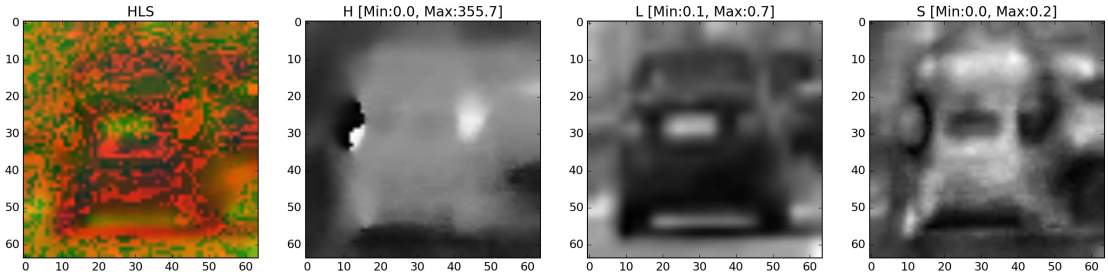
Colorspaces

I investigated several color spaces to see the information each channel would contain when being applied to a car and non-car image. The code for that can be found in `readin_images.py`. See the following for the car examples:

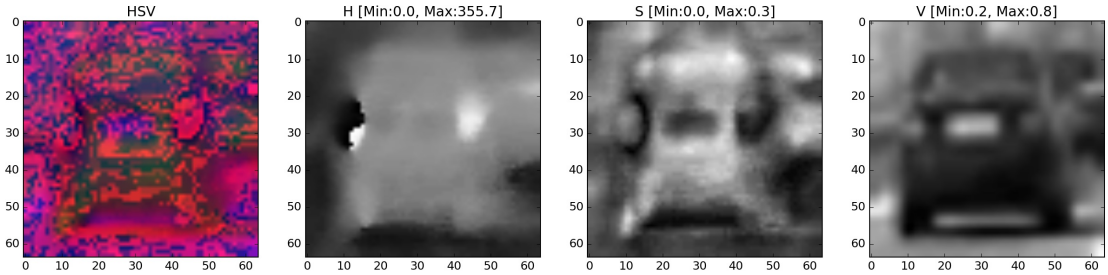
RGB



HLS



HSV



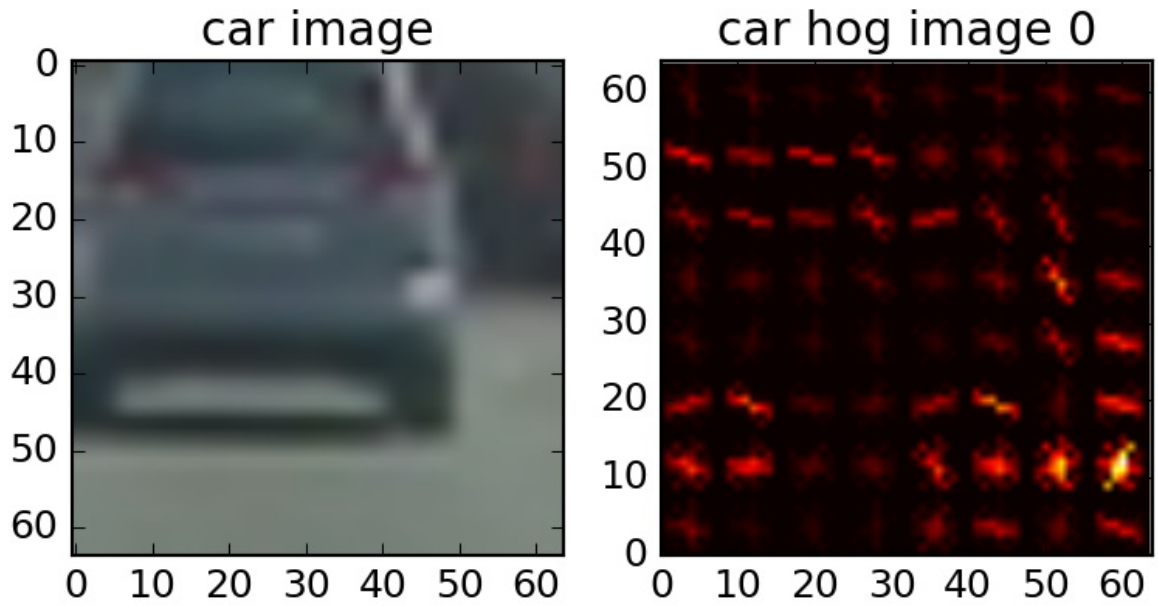
1. Explain how (and identify where in your code) you extracted HOG features from the training images.

Histogram of Oriented Gradients (HOG)

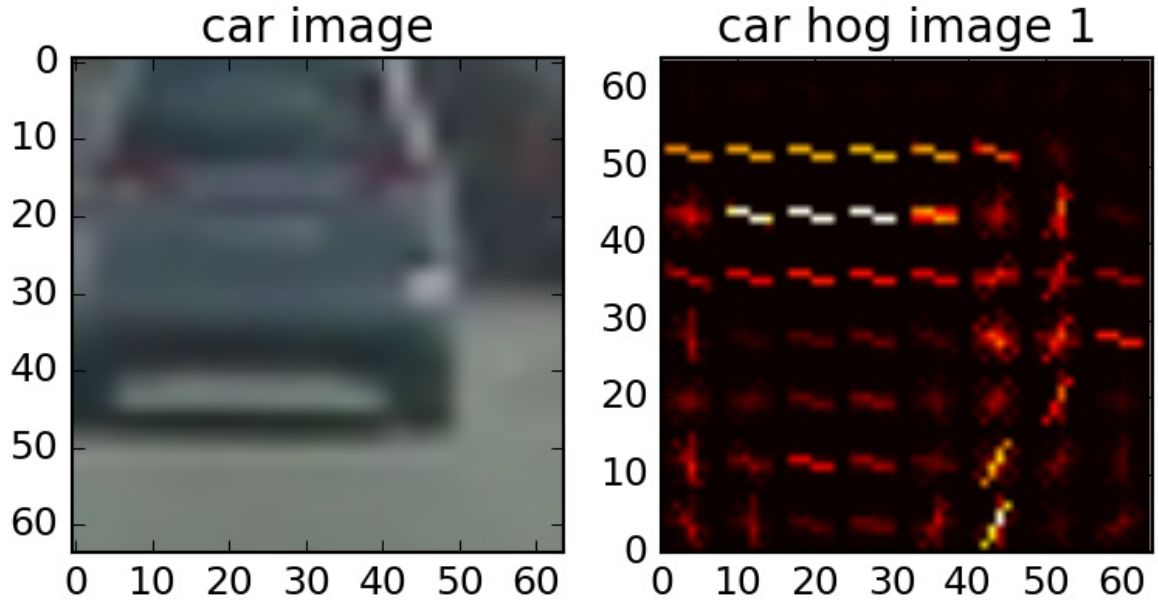
I then explored different color spaces and different `skimage.hog()` parameters (`orientations`, `pixels_per_cell`, and `cells_per_block`). I grabbed random images from each of the two classes and displayed them to get a feel for what the `skimage.hog()` output looks like.

Here is an example using the `HSL` color space and HOG parameters of `orientations=9`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)` for the car class:

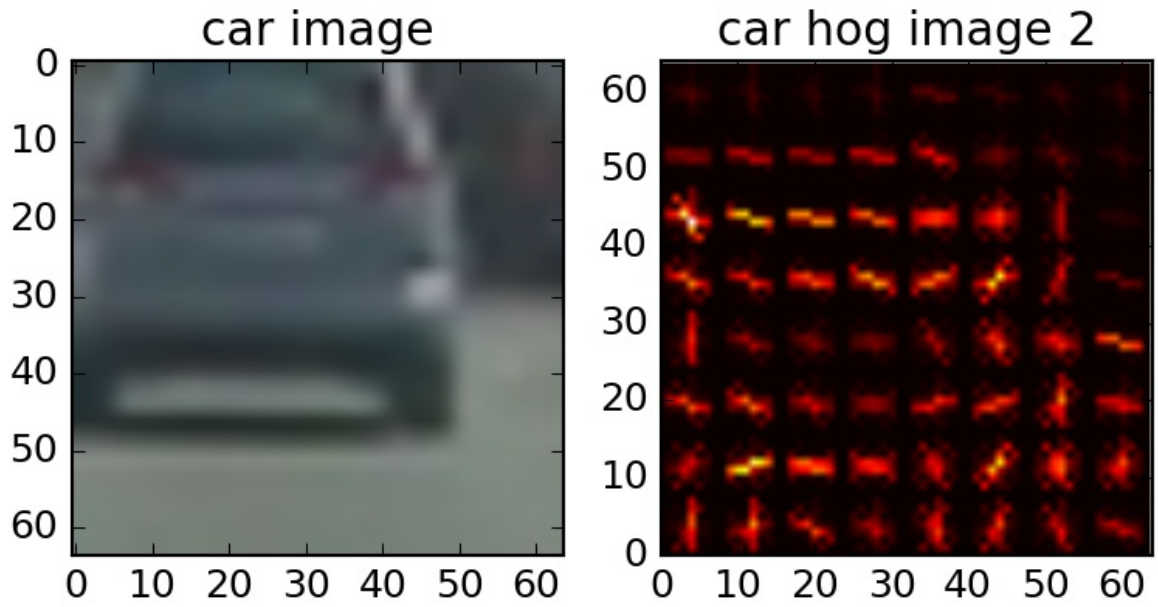
Channel 1



Channel 2

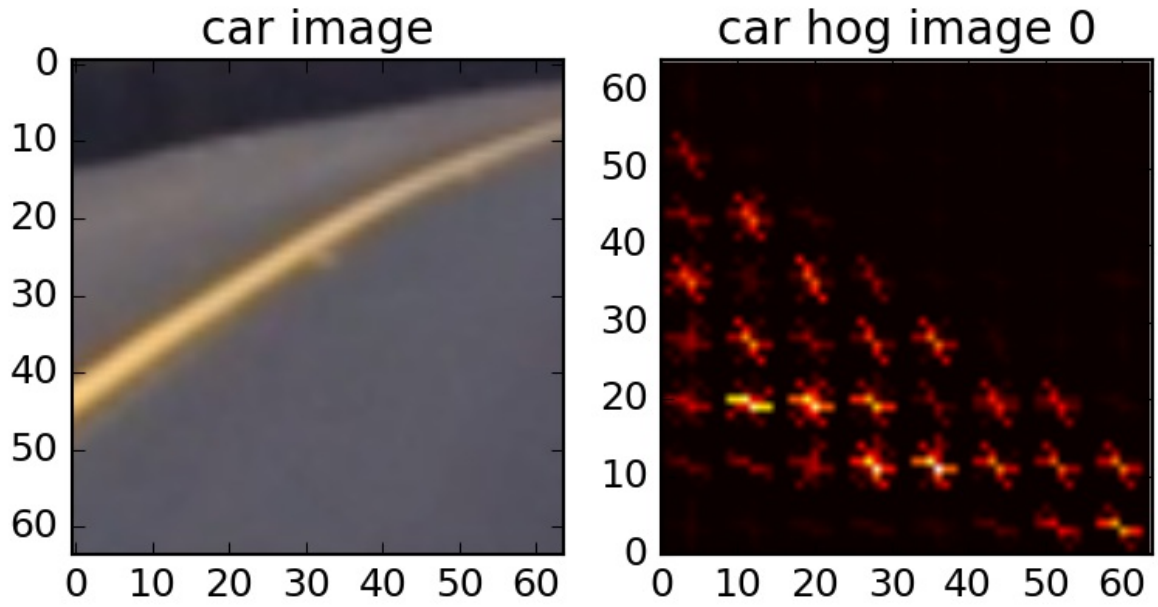


Channel 3

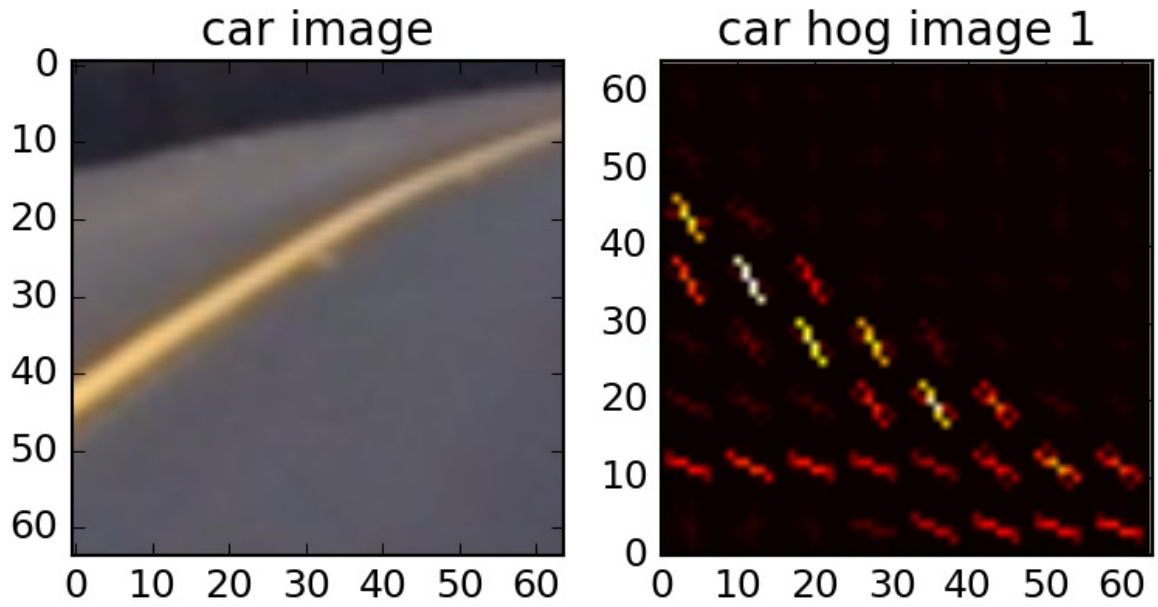


And here is an example using the `HSL` color space and HOG parameters of `orientations=9`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)` for the car class:

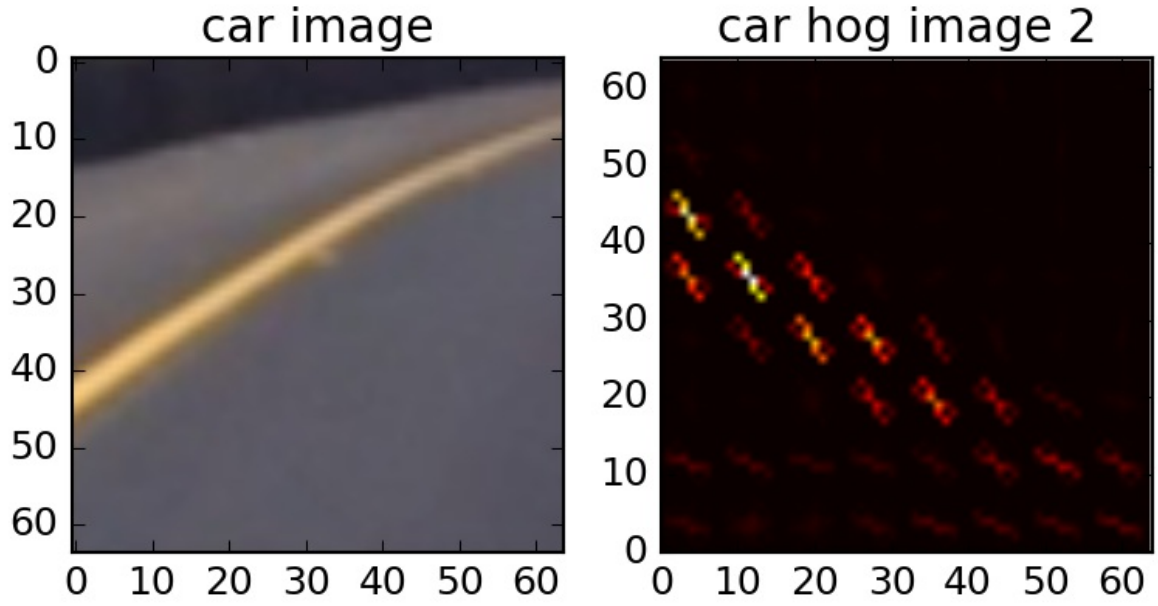
Channel 1



Channel 2



Channel 3

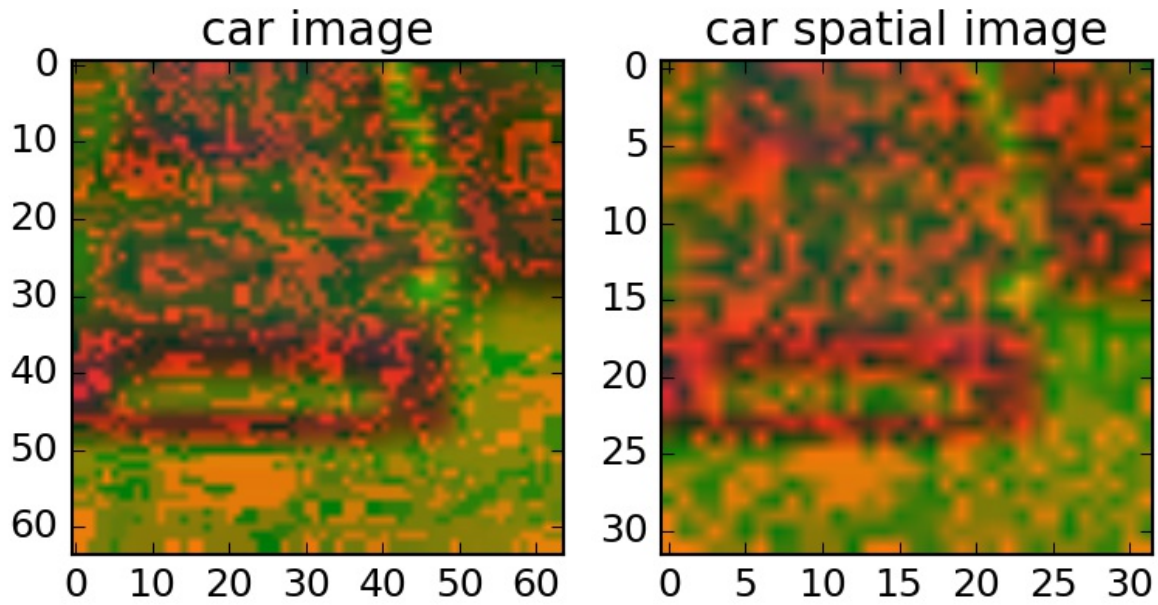


The hog feature image is flipped horizontally, but that does not impact the classification as long is it is consistent with both training and prediction.

Spatial Binning

The image gets downsampled to 32 by 32 pixels. Downsampling the car

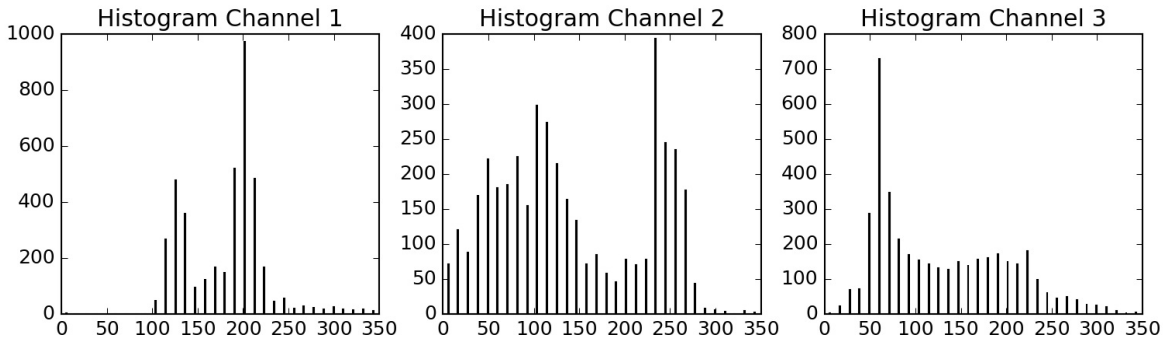
looks like the following:



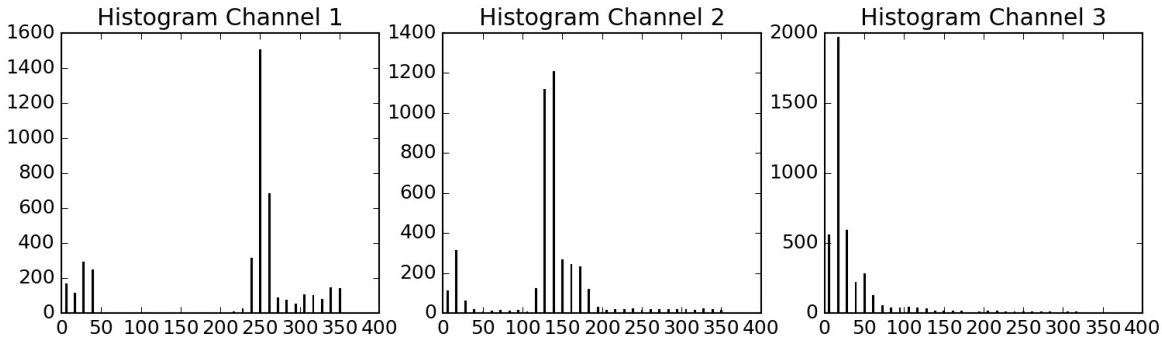
Color Histogram

For each color channel, a histogram is calculatated.

Car



Not-Car



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

I tried various combinations of the following parameters:

- orientations
- pixels_per_cell
- cells_per_block

I did not experience any improvements above orientation = 9. For the other two parameters, a tradeoff has to be chosen between computation time and accuracy. I went with num_pix_per_cell = 8 and num_cell_per_block = 2.

Setting the spatial binning parameter spatial_size was pretty easy, as the input image is already (64x64). I tried (32x32) and (16x16) and finally went with (32x32) to improve accuracy, as the feature vector does not get that much longer.

For the histogram binnings, I also decided to use 32bins, as it proved to show signficantly differences when aplied to car and non-car images.

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. The code can be found in `train_classifier.py`. One important step is the scaling of all features using the same scaler. That's done in the following code:

```
X = np.vstack((cars_features, not_car_features)).astype(np.float

# Fit a per-column scaler
```

```
    X_scaler = StandardScaler().fit(X)

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

I then create the classifier and perform the classiciation.

FInnally, I save both settings and classifier as a pickle to seperate classification and prediction in the video stream.

Sliding Window Search

The vehicle detection is performed in `detect_vehicles.py`.

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?

I started out with implementing a sliding window technique:

```
def generate_windows_list(img,
                          x_start_stop=[None, None],
                          y_start_stop=[None, None],
                          xy_window=(64, 64),
                          xy_overlap=(0.5, 0.5)):
```

It creates a list of windows wich are then applied to a given image to extract features for the classification.

Certainly, only one window size does not prove to be sufficient, so I played around with both windows size and the overlap. Also, I cutted off

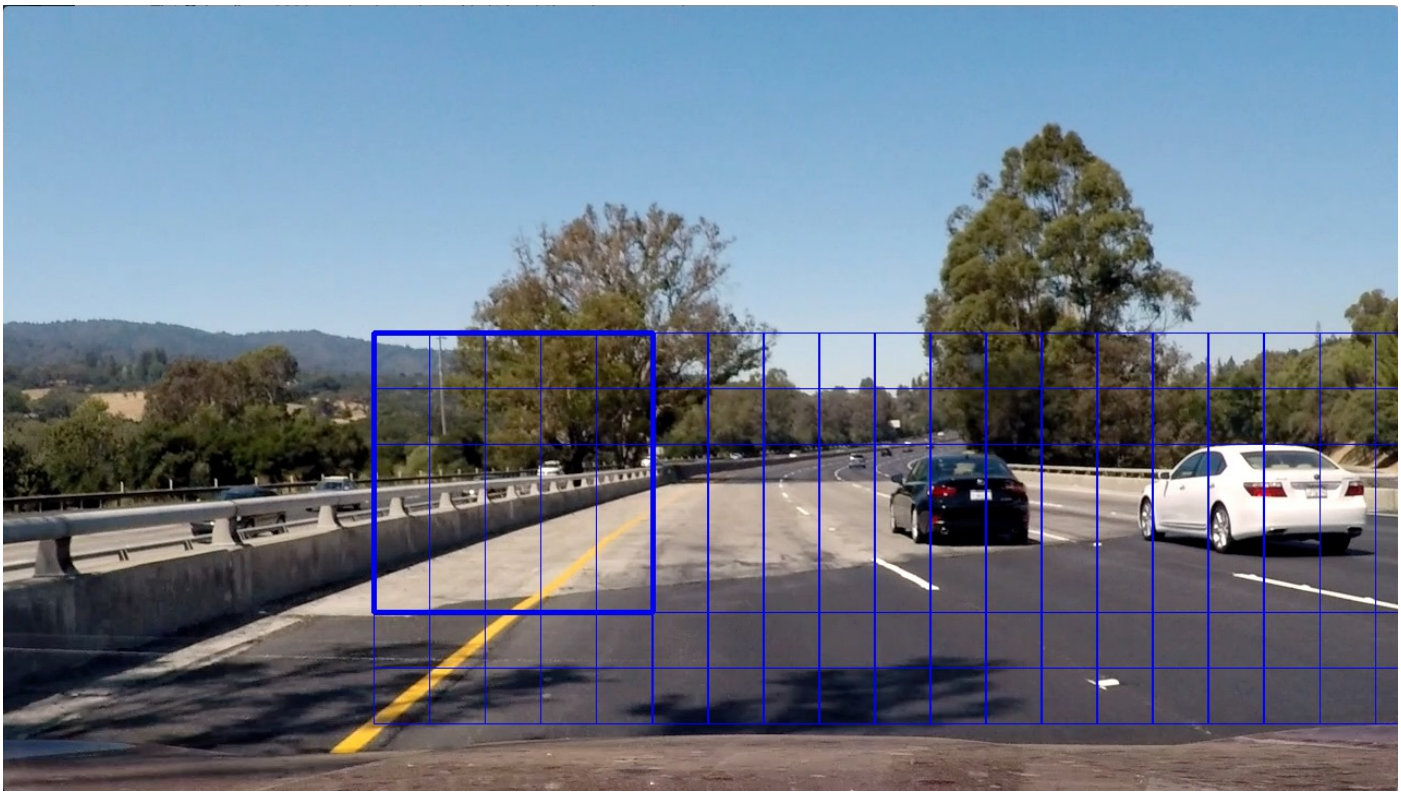
regions that where not of interest and would only cause additional computation time or worse: False positives.

In the following, I am showing the area that was searched by the sliding windows. In a bolder frame, the size of the window can be seen (which, due to the overlap, might be not that obvious)

Sliding Windows Search 64x64, 80% Overlap

Sliding Windows Search 128x128, 80% Overlap

Sliding Windows Search 256x256, 80% Overlap



Extracting features from image only once

The sliding window technique worked decently but turned out to be too slow. As proposed in the Udacity Tips and Tricks section, I went for the

other way: Apply the feature extraction to the whole image and then chunk it into slices. The code to this can be found in the function `find_cars()`.

```
def find_cars(img,
              clf,
              scaler,
              color_space='RGB',
              spatial_size=(32, 32),
              hist_bins=32,
              scale=1,
              cells_per_step=2,
              x_start_stop=[None, None],
              y_start_stop=[None, None],
              orient=9,
              pix_per_cell=8,
              cell_per_block=2):
```

Specifying a window size is done indirectly. Why so? The reason is that features are extracted from a 64-64 window. In order not to change that, the input image is just scaled to achieve a different searching window size. This is done via the `scale` parameter. Cells per step can be used to specify how many cells the searching window should be moved. This correlates with the overlap-parameter in the sliding window approach.

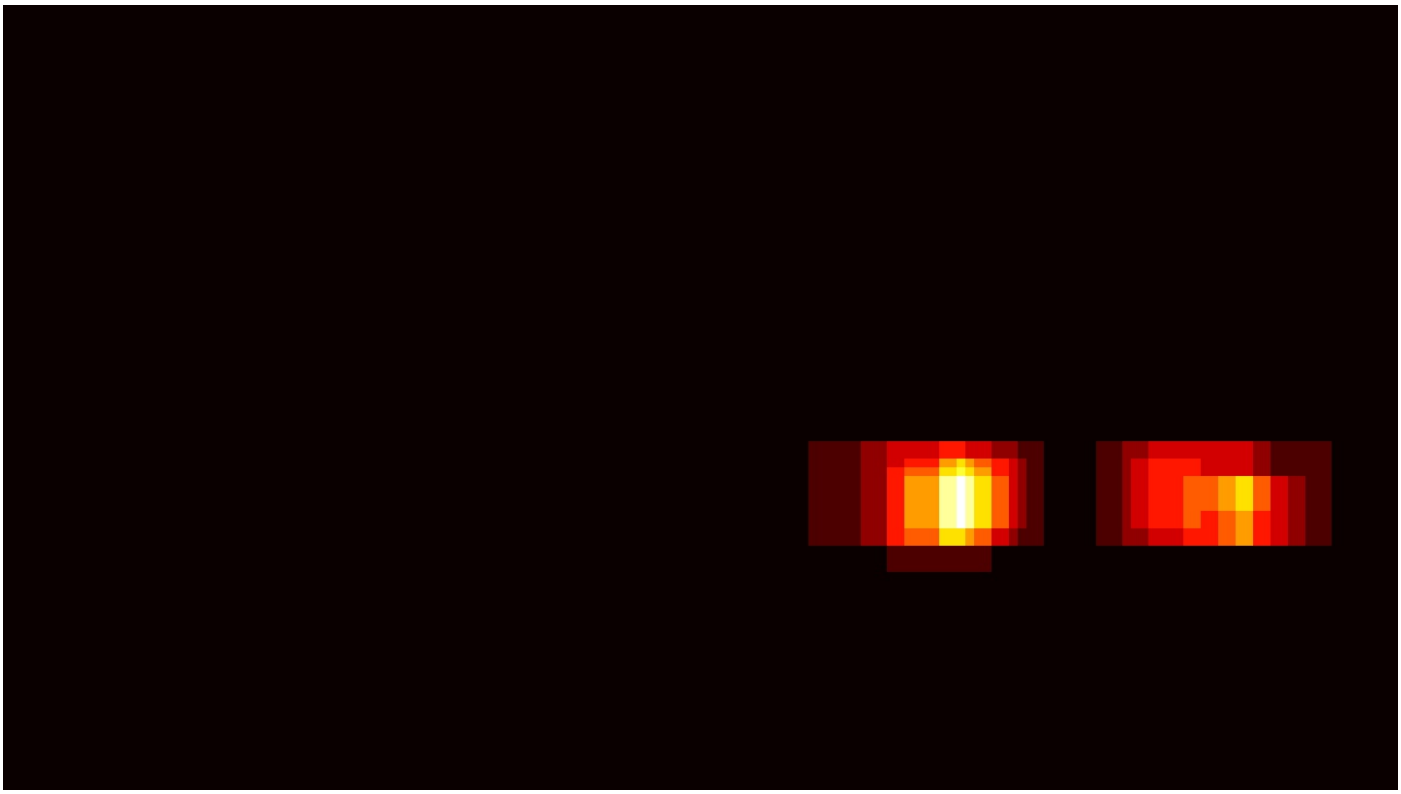
The `finding_cars` function adds up all windows in which a positive classification occured. That leads to a heatmap, where the hot parts are the areas in which a car is assumed to be.

Positive Windows from 64x64 search

Positive Windows from 128x128 search

![alt text][image_pos_windows128]

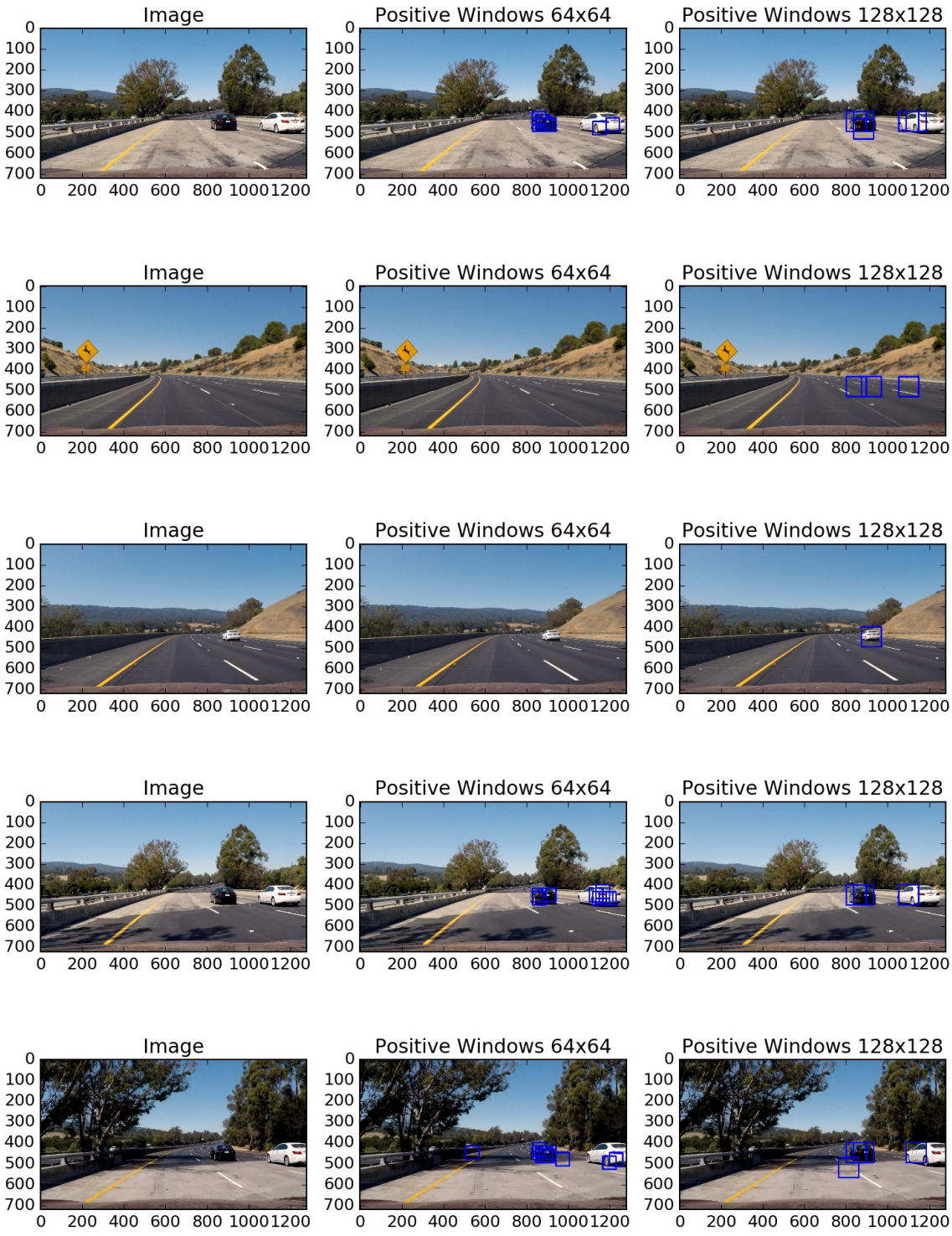
Heatmap from combined positives windows



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 all three scales using HLS color space. I used HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. Here are some example images. It shows the results of positive windows detection for both window sizes, which are then combined via adding up the heatmap.



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 am adding up the heatmaps from previous time frames using weightings to make sure recent detections have a greater impact and not to be too late with the detection. This is done in the `Tracing_algorithm` class.

```
class Tracing_algorithm():
    def __init__(self, queuelength):
        self.queuelength = queuelength

        self.current_heatmap = None
        self.recent_heatmaps = []
        self.avg_heatmap = None

        self.frame = 0

        self.buffer = 0

        self.coord_of_found_cars = []

        self.labels = None
```

```
        self.number_of_found_cars = 0

        self.binary_map = None

        self.list_of_cars = []
```

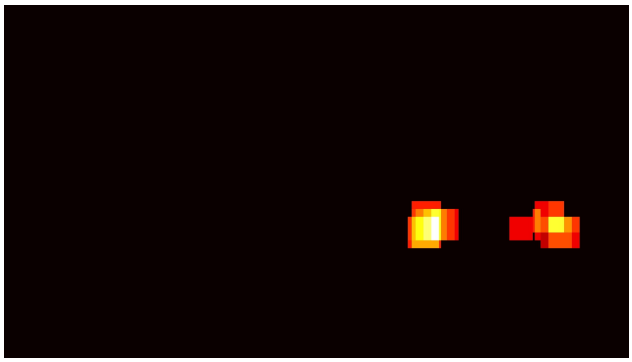
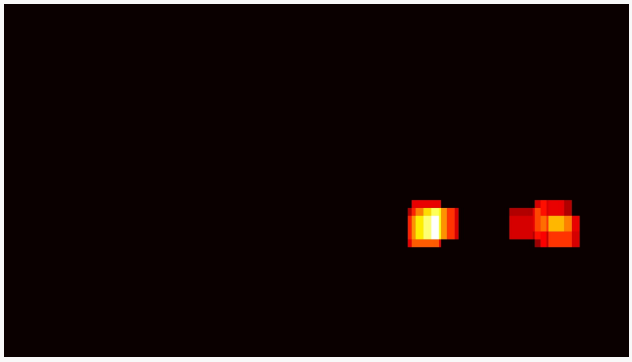
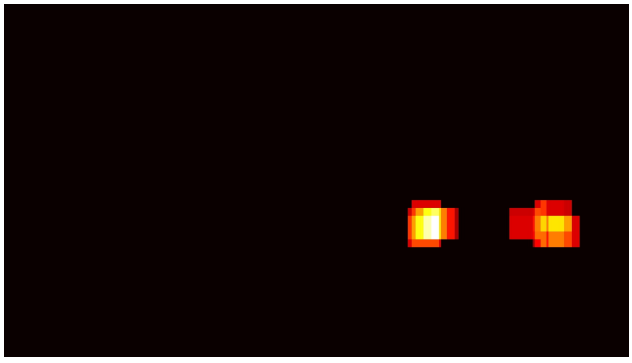
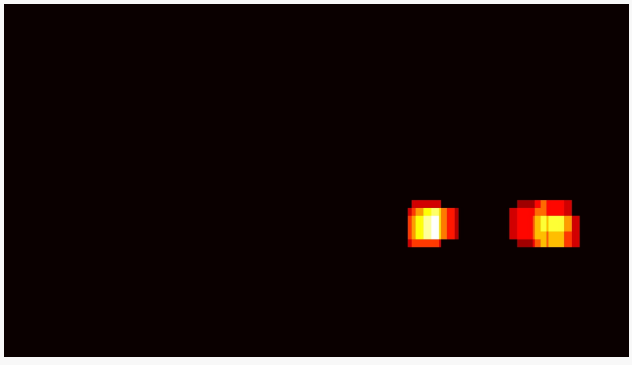
`coord_of_found_cars` contains the center of the detected car and also width and height. It is extracted after applying a threshold to the heatmap (with the positive detections) and extracting boxes using `scipy.ndimage.measurements.label`.I then assumed each blob corresponded to be a possible vehicle. Those coordinates are then used to create car instances for each detected car. This seperation allows for some plausibility checks before creating a new car which was not detected before. All this is done in the `update_car_list()`-method of the `Tracing_algorithm`-class. In there, the coordinates are searched through to find out wether they match a previously known car. If so, the new coordinates are passed to that car instance. If not, the following checks are applied:

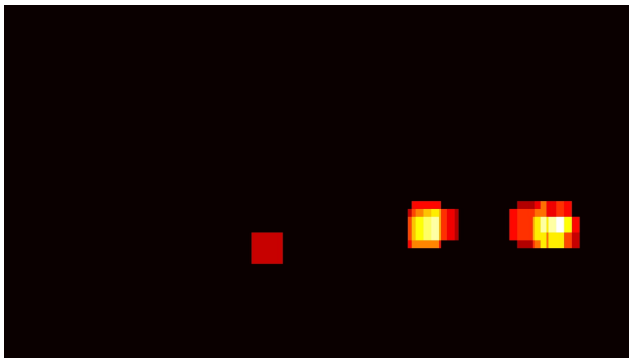
- Does the detected shape meaning ratio between height and width makes sense?
- Does the center of the newly detected car lies within any boxes of known cars?

If none of those is true, a new car instance is created. Only after a series of positive detections of this same car instance, it is displayed in the image.

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:

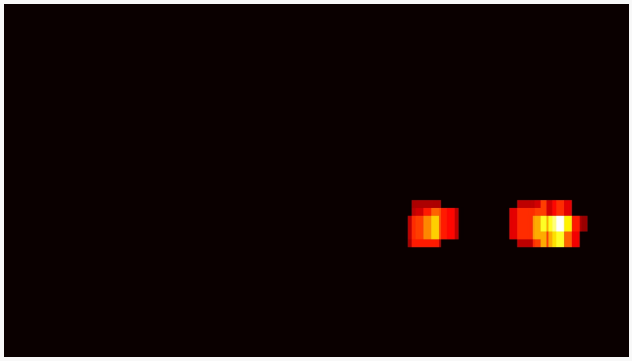
Here are six frames and their corresponding heatmaps:

Positive Windows with 64x64 search	Averaged Heatmap
Frame 9 	Frame 9 
Frame 10 	Frame 10 
Frame 11 	Frame 11 
Frame 12 	Frame 12 
Frame 13	Frame 13

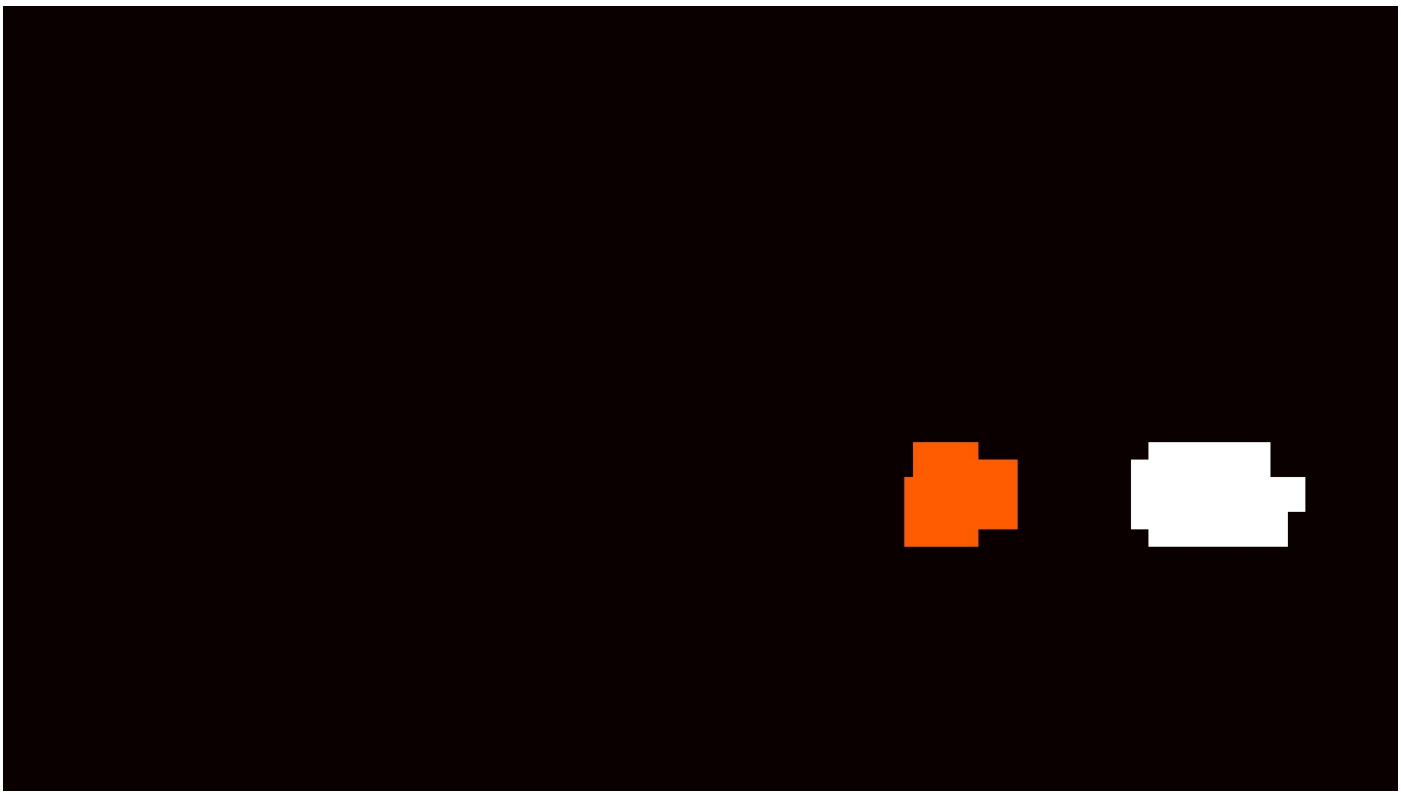


Frame
14

Frame 14



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



Here the resulting bounding boxes are drawn onto the last frame in the series:

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 classification still produces too many false positives. There are various options to do so:

1) One possible solution would be to go with another classifier. Convolutional Neural Networks might be a promising solution for that, as it works very well when searching for patterns in an image. 2) Improvement of training data. Instead of time-series data, clearly different input data could be used. The udacity training seems to be

promising. 3) Thresholding the decision function, meaning that if the classifier is not quite sure (but is going to decide for one or the other class), this classfication result could be neglected.

Classification goes hand in hand with the window selection. If the classification would be more robust, a less amount of windows would be needed, which would result in a much faster pipleline.