Project Vehicle Detection with HOG and SVM

Vehicle Detection Project by Xuandong Xu

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.

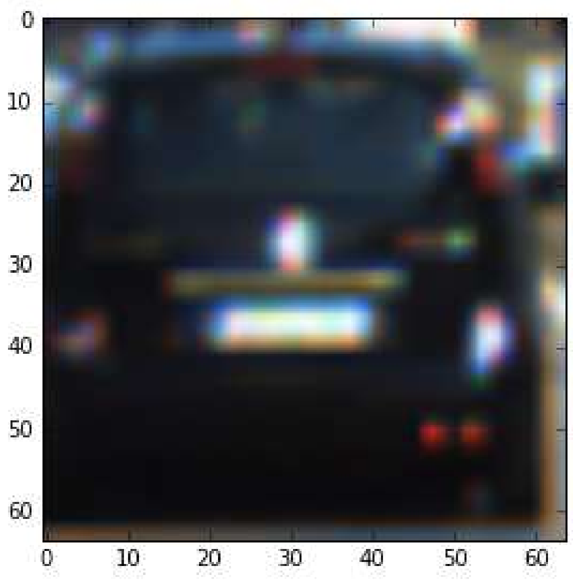
The code for extracting the HOG features is shown here:

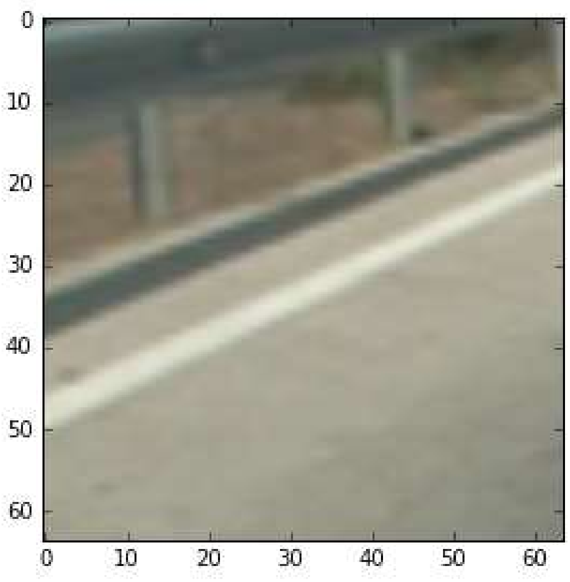
```
from skimage.feature import hog
# Define a function to return HOG features and visualization
def get_hog_features(img, orient, pix_per_cell, cell_per_block,
                        vis=False, feature_vec=True):
    # Call with two outputs if vis==True
    if vis == True:
        features, hog_image = hog(img, orientations=orient,
                                  pixels_per_cell=(pix_per_cell, pix_per_cell),
                                  cells_per_block=(cell_per_block, cell_per_block),
                                  transform_sqrt=True,
                                  visualise=vis, feature_vector=feature_vec)
        return features, hog_image
    # Otherwise call with one output
    else:
        features = hog(img, orientations=orient,
                       pixels_per_cell=(pix_per_cell, pix_per_cell),
                       cells_per_block=(cell_per_block, cell_per_block),
                       transform_sqrt=True,
                       visualise=vis, feature_vector=feature_vec)
        return features
```

This is from the lessons, so I put this function in the file called `fromlesson.py` .

From the lesson I know that the GTI folders have images that are very similar. To avoid training samples leaking into testing samples, I re-ordered and manually splitted my training and testing set. I took 20 percent of the overall GTI images and put them at the very end of my sample list. I then took these 20 percent samples as my test sets.

Here's a picture for car and a picture for non-car example:

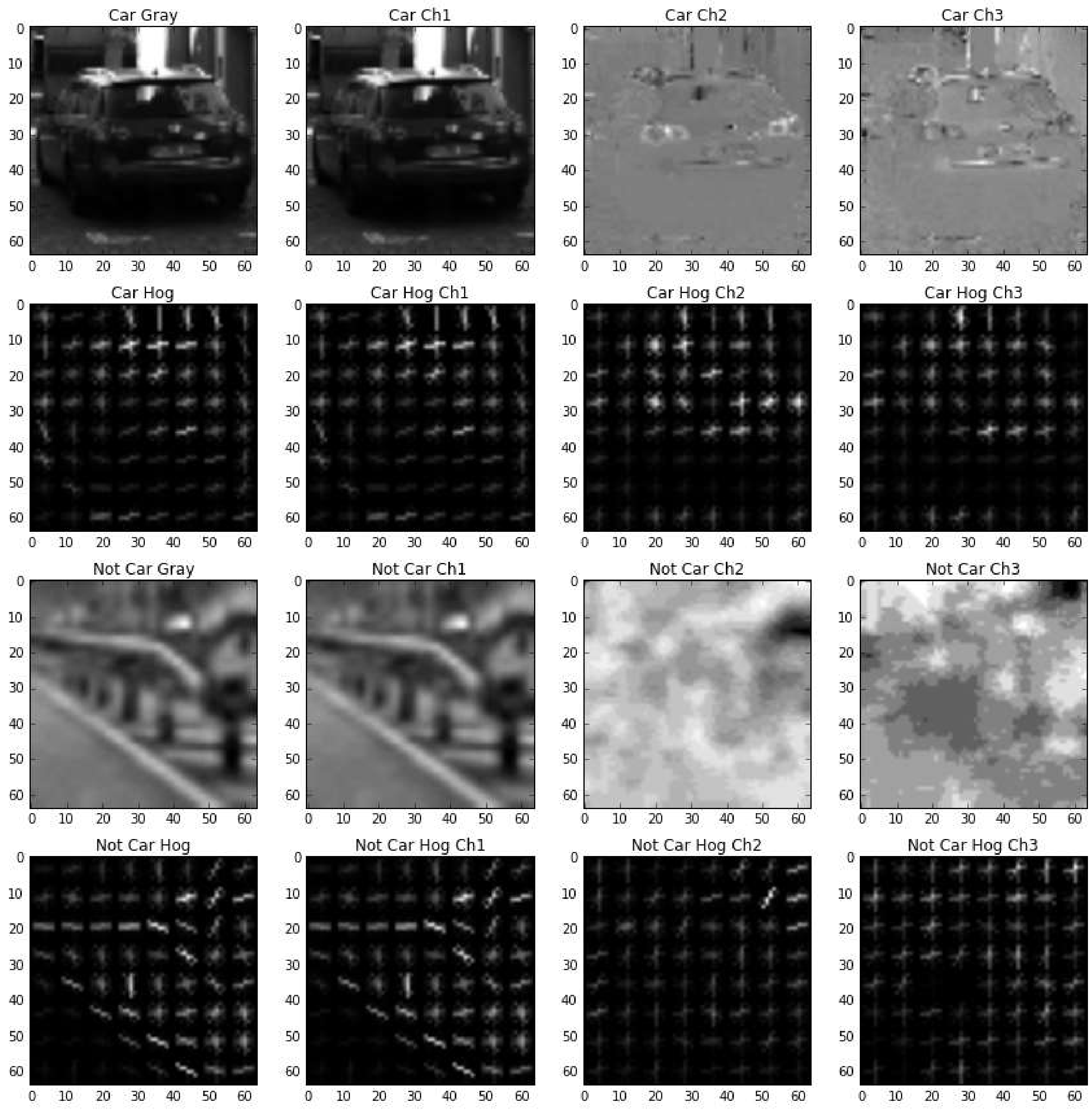




I tested on different color spaces and decide to go with `YCrCb` colorspace, because it has the highest accuracy when using a classifier. Here's my setup for the HOG detection:

```
class trainer:
    def __init__(self):
        self.colorspace = 'YCrCb' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
        self.orient = 9
        self.pix_per_cell = 8
        self.cell_per_block = 2
        self.hog_channel = 'ALL' # Can be 0, 1, 2, or "ALL"
        self.spatial_size = (8,8)
        self.hist_bins = 32
```

Here's a sample image I generated to showcase my setup for the HOG detection:



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

I started by poking around these parameter to see which one would improve the accuracy of my classifier. Eventually I see that the `spatial_size` would have negative effect on the result, and using all channels in `hog_channel` would boost my result. So I decreased the default `spatial_size` to 8 and set my `hog_channel` to be all.

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).

Due to similiar images in the `GTI*` folders, the training samples may get leaked into the testing samples. So I decided to manully split the data by putting the last 20 percent images in the `GTI` folder at the very end of my sample list. Here's a piece of code:

```
def r8020(self,*mylists):
    # returns 80% and 20% lists
    return80 = lambda mylist:mylist[:int(len(mylist)*0.8)]
    return20 = lambda mylist:mylist[int(len(mylist)*0.8):]
    return1 = return80(mylists[0])
    return2 = return20(mylists[0])
    for mlist in mylists[1:]:
        return1 += return80(mlist)
        return2 += return20(mlist)
    # int(len(mylists[-1])*0.8)
    return return1,return2

def prepare_data(self):
    returnindex = lambda string,temp:[i for i,item in enumerate(temp) if string in item]
    # extract images from each folder, car samples first
    gti_far = glob.glob('./files/vehicles/GTI_Far/*png')
    gti_left = glob.glob('./files/vehicles/GTI_left/*png')
    gti_middle = glob.glob('./files/vehicles/GTI_MiddleClose/*png')
    gti_right = glob.glob('./files/vehicles/GTI_Right/*png')
    gti_non= glob.glob('./files/non-vehicles/GTI/*png')
    # not-car samples
    kitti = glob.glob('./files/vehicles/KITTI_extracted/*png')
    extra = glob.glob('./files/non-vehicles/Extras/*png')

    #split by 80 percent
    gti80,gti20 = self.r8020(gti_far,gti_left,gti_middle,gti_right,gti_non)
    extras80,extras20 = self.r8020(kitti,extra)
    # merge, put 20 percent GTI samples to the last
    newlist = gti80+extras80+gti20+extras20
    nonvehicleindex = returnindex('non-vehicles',newlist)
    labels = np.ones(len(newlist))
    labels[nonvehicleindex] = 0
    self.newlist = newlist
    self.labels = labels
```

I then trained my SVM classifier with spatial features, color features and HOG features. I extracted the features and scaled them because SVC needs to have zero-mean to work well. Also different features need to be normalized when combining them together otherwise the learning will be biased. Here's a piece of code showing how I did it:

```
def train_classify(self):
    features = extract_features(self.newlist,
                                          color_space=self.colorspace, spatial_size=self.spatial_size,
                                          hist_bins=self.hist_bins, orient=self.orient,
                                          pix_per_cell=self.pix_per_cell, cell_per_block=self.cell_per_b
                                          spatial_feat=True, hist_feat=True, hog_feat=True)
        X = np.array(features).astype(np.float64)
        # Fit a per-column scaler
        X_scaler = StandardScaler().fit(X)
        # Apply the scaler to X
        scaled_X = X_scaler.transform(X)
        # Define the labels vector
        y = self.labels
        # retrieve length of samples
        totalsize = len(self.newlist)
        # manual split
        X_train, y_train = shuffle(scaled_X[:int(totalsize*0.8)],y[:int(totalsize*0.8)])
        X_test, y_test = shuffle(scaled_X[int(totalsize*0.8):],y[int(totalsize*0.8):])

        # Use a linear SVC
        svc = LinearSVC(C = 0.2)
        #svc = SVC(probability = True)
        # Check the training time for the SVC
        t=time.time()
        svc.fit(X_train, y_train)
        t2 = time.time()
        print(round(t2-t, 2), 'Seconds to train SVC...')
        # Check the score of the SVC
        print('Test Accuracy of SVC = ', round(svc.score(X_test, y_test), 4))
        # Check the prediction time for a single sample
        t=time.time()
        self.svc = svc
        self.X_scaler = X_scaler
```

Sliding Window Search

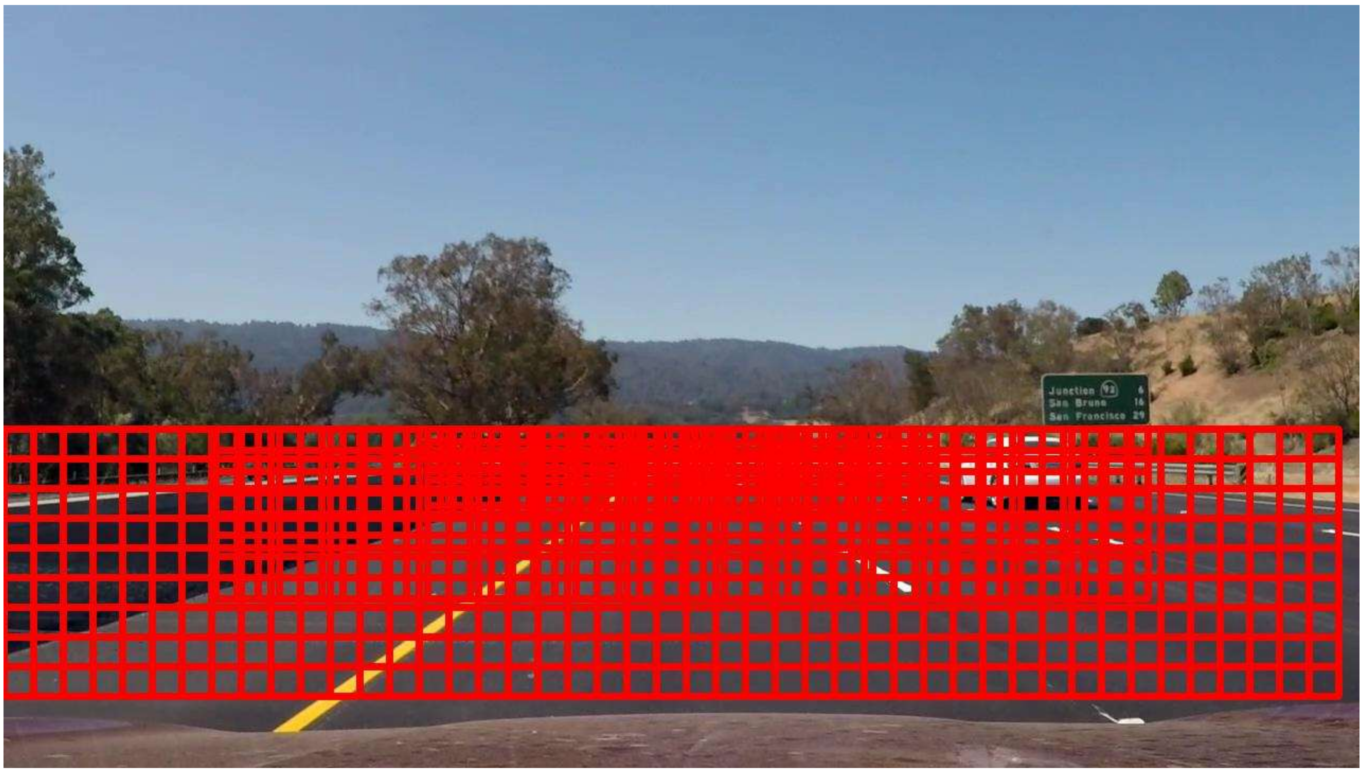
1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

By looking at the project video, one could tell that the further the car is away from you, the smaller the size it will be, and the position is near the top more. So I decided to pick different scales based on the how many times I want to scan. I set the scales to be uniform distributed between 1.75 to 0.75. The interval between this gap is decided by the number I want to search. Also, one could tell that the region that need to be scanned is different. So a smaller scale would correspond to a smaller rectangle close to the top. Therefore, the `ystop` criteria for each search is different. And the width of rectangle is changed for every search. Here's a piece of code:

```
def multisearch(self,img,param,times=3):
    # suppose scale is from 1 to 2
    scales = [1.75-i*(1./(times-1)) for i in range(times)]
    ystops = [660-i*(260./times) for i in range(times)]
```

```
        xoffsets = [i*(600./times) for i in range(times)]
        #xoffsets = [0,0,0]
        bboxs = []
        draw = np.copy(img)
        for i in range(times):
            draw,bbox = self.find_window(img,draw,param,scales[i],ystops[i],xoffsets[i])
            bboxs.append(bbox)

        bboxs = [i for i in bboxs if len(i)>0]
        return draw,bboxs
```

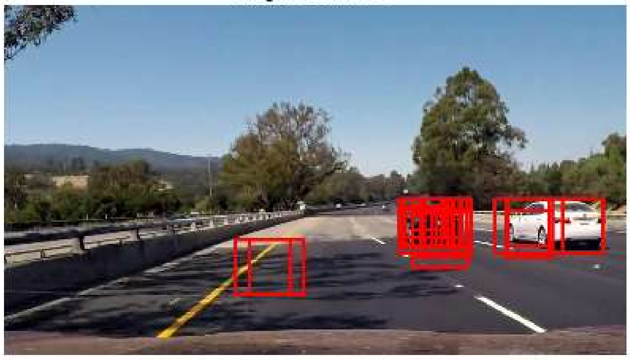
Here's my graph representation of it:



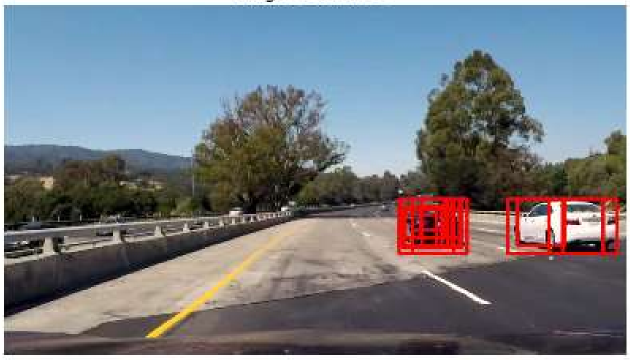
2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

In the end I choose to search on 5 scales, combining spatial feature, color histogram feature and HOG feature. I also played around with the `C` parameter in the `LinearSVC` class. I set it to be smaller than 1 because the accuracy was so high I suspect it might overfit. Here are some example images:

Img 1


Img 1 searched

Img 2


Img 2 searched

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.)

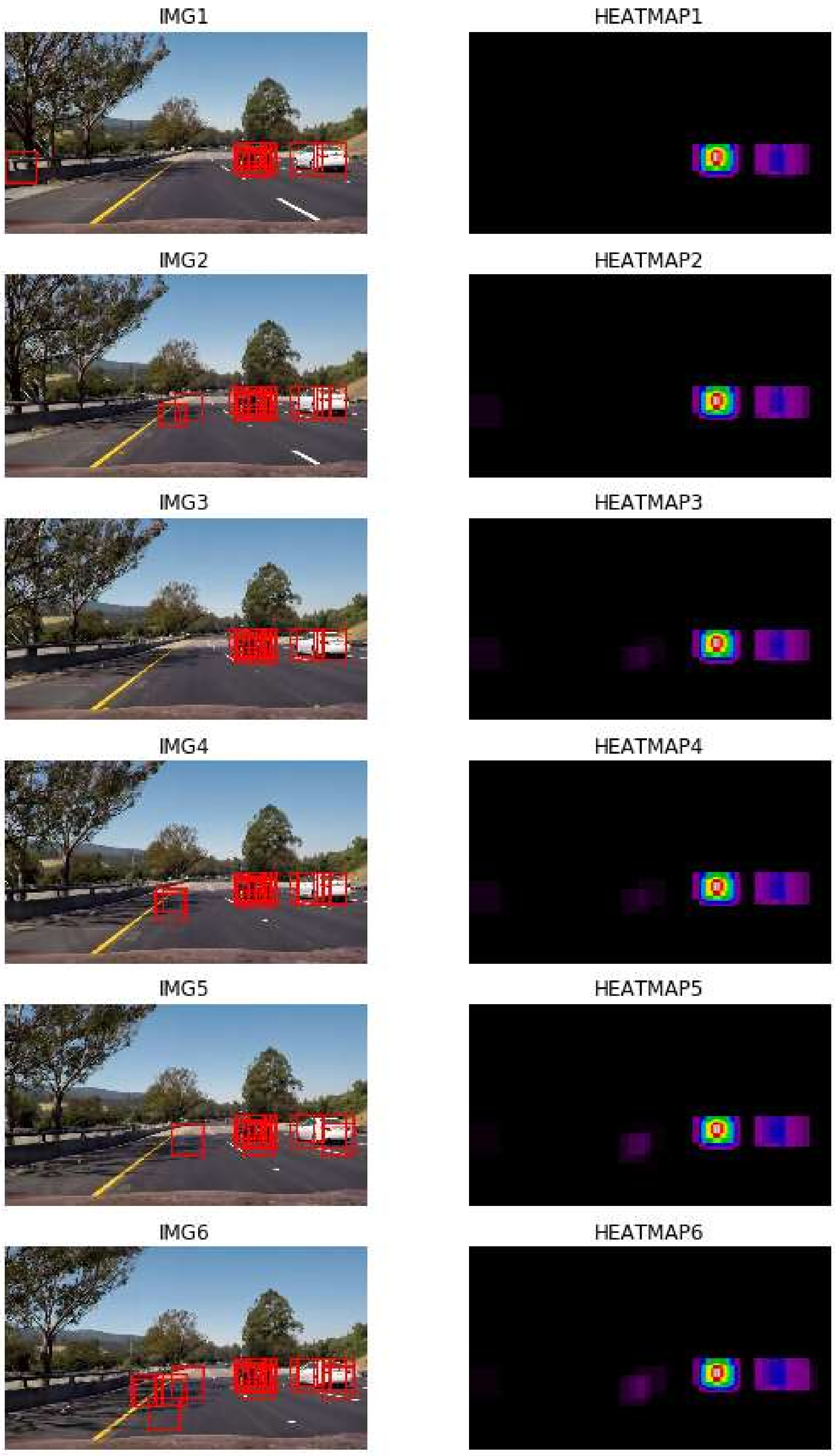
I included the video file under `./files/output/project_video_rendered.mp4` .

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.

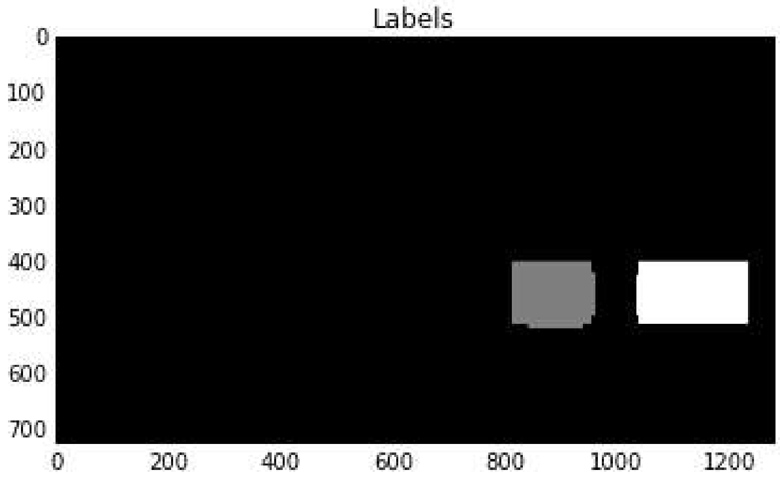
For the false positives and overlapping, I created a function to record current list of boxes found using slide window search. The rest of the code is similiar to the lesson on Udacity. It is basically stacking a heatmap and setting up a threshold for removing weak "heated" area. Here's how I implemented my function for recording current boxes:

```
class heatmap:

    def __init__(self):
        self.previousboxes = []
        self.allboxes = []

    def recordboxes(self,bbox_lists):
        # it will record up to 14 list of boxes from past images
        # past that, it will delete the old one and insert new one,
        # just like a queue
        if len(self.previousboxes)<14:
            self.previousboxes.append(bbox_lists)
        else:
            self.previousboxes.remove(self.previousboxes[0])
            self.previousboxes.append(bbox_lists)

        # refresh allboxes
        self.allboxes = []
        for boxlist in self.previousboxes:
            self.allboxes += boxlist
        #print('allboxes length',len(self.allboxes))



    def add_heat(self, heatmap):
        # Iterate through list of bboxes
        for box_list in self.allboxes:
            for box in box_list:
                # Add += 1 for all pixels inside each bbox
                # Assuming each "box" takes the form ((x1, y1), (x2, y2))
                heatmap[box[0][1]:box[1][1], box[0][0]:box[1][0]] += 1

        # Return updated heatmap
        return heatmap
```

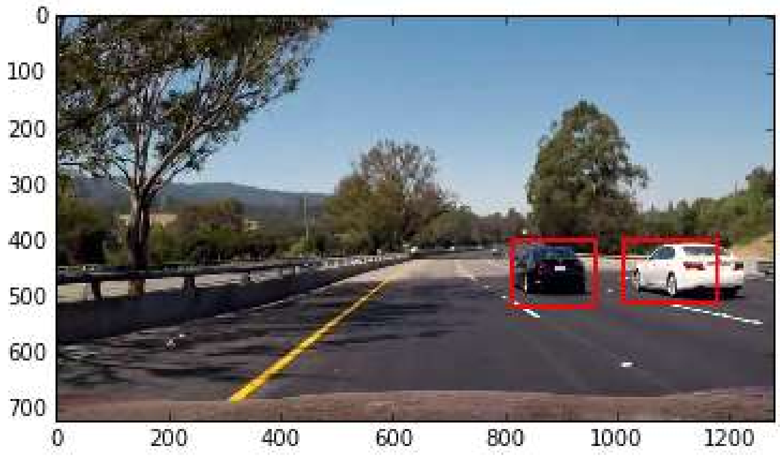
Here are six frames and their corresponding heatmaps:



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?

For this project, many parameters were tuned. I tried many combinations of scale factors, overlapping pixel sizes and etc. I once came up a good classifier with `SVC(probability = True)` , however it doesn't scale very well with big data set. The time it took to train and process images in a video were unacceptable. Due to that issue I switched back to `LinearSVC` as my classifier because of the speed.

The pipeline will likely to fail when seen shadows. I think the reason is due to overfitting. I did however lower the `C` term in the `LinearSVC` classifier. Maybe it wasn't enough, maybe two many features were extracted from a single image that it is causing noise underneath.

To make it more robust, my best bet is to implement a deep learning classifier using `Tensorflow` or

`Keras`. I've seen some people already successfully implemented it and it worked great. Also training data could be split and processed in a smarter way. Surely that would help make the pipeline more robust.