Udacity Vehicle detection project

Car detection and tracking

Project aims at detecting vehicles in a video from non vehicles. It also aims at detecting lane lines in the video.

Training data is obtained from Udacity site namely vehicles and non vehicles data set which helps identify between vehicles and background.

Project is divided into two steps:

1. Feature extraction
2. Classifier selection

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

Feature extraction:

Classifier used in the case is SVM. But before the use of classifier we need to extract features. This step is performed by the HOG feature extraction technique. HOG is Histogram Oriented Gradients. It helps to find the gradient of every region. Every object has a unique motif or pattern, so HOG helps to identify this. This can also be called as features.

We also use binned color and color histogram features. Color and shape are provided by binned color features and colors alone as features is obtained from color histogram features.

Parameters used are as below:

color_space = 'RGB'

orient = 8

pix_per_cell = 16

cell_per_block = 1

hog_channel = 'ALL'

spatial_size = (16, 16)

hist_bins = 16

spatial_feat = False

hist_feat = False

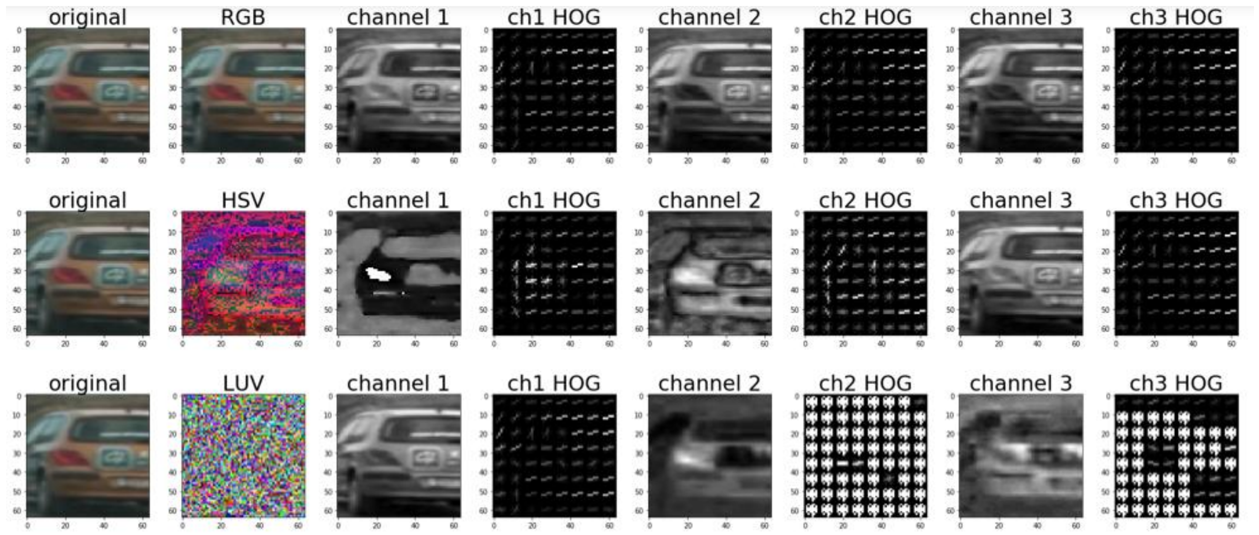
hog_feat = True

HOG detailed:

HOG is performed on each of the color channels. In RGB images, there are 3 channels. So there will be HOG performed on 3 channels.

I also tried HOG on three different color scheme/ spaces. Color spaces I tried was RGB, HSV and LUV.

Below are images of HOG applied on three layers for each of the color schemes:



I could have used the get_hog_features() function described in the Udacity class but I used the command ‘hog’ directly to obtain HOG of the required images.

Find the code below:

```
def extract_features (image, color_space='RGB', spatial_size=spatial_size,
                    hog_channel='ALL'):
    file_features = []
    feature_image = convert_color(image, color_space)
    spatial_features = cv2.resize(feature_image, spatial_size).ravel()
    file_features.append(spatial_features)
    if hog_channel == 'ALL':
        hog_features = []
        for channel in range(feature_image.shape[2]):
            hog_features.append(hog(feature_image[:,:,channel], orientations
= orient, pixels_per_cell = (pix_per_cell, pix_per_cell),
cells_per_block=(cell_per_block, cell_per_block), transform_sqrt=True,
                                visualise= False, feature_vector=True))
        hog_features = np.ravel(hog_features)
    else:
        hog_features = hog(feature_image[:,:,hog_channel], orientations =
orient, pixels_per_cell = (pix_per_cell, pix_per_cell),
                    cells_per_block=(cell_per_block, cell_per_block),
transform_sqrt=True,
                    visualise= False, feature_vector=True)
    file_features.append(hog_features)
    assert (file_features != [])
    return np.concatenate(file_features)
```

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

I tried a mix of various parameters and settled on what I required in the end.

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 linear SVM as the classifier for my HOG features. So kernel is linear for the SVM classifier.

Classifier selection:

After features are extracted we need to select the appropriate classifier. The options that comes to our mind is decision tree, neural networks and SVM. But we finally choose SVM due to the ease of implementation.

SVM stand for support vector machine and it helps to find a hyperplane that seperates between classes.

Please find the code below for SVM classifier:

Features are needed to train a classifier and predict test set

Sliding Window :

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 wrote a function that will create a list of windows to search the cars in the image. I generated windows based on HOG features. If I tried to create windows of random size then there will be many windows and that will be unproductive. Objects nearer to my car will be bigger than the objects farther from my car. Thus I might have to use windows of all different sizes, which is inefficient.

Thus in my HOG feature based window, the HOG is calculated for the image just once. One main benefit is that HOG feature based window do not vary in size but the images get varied in size during feature extraction steps. Model works well over 50% of overlap.

Sliding window is implanted form the code in Udacity class. Sliding window searches for the images of the car in the whole of the image. Size of the window has to be tuned appropriately to match with the required image to be identified.

Please find the codes below:

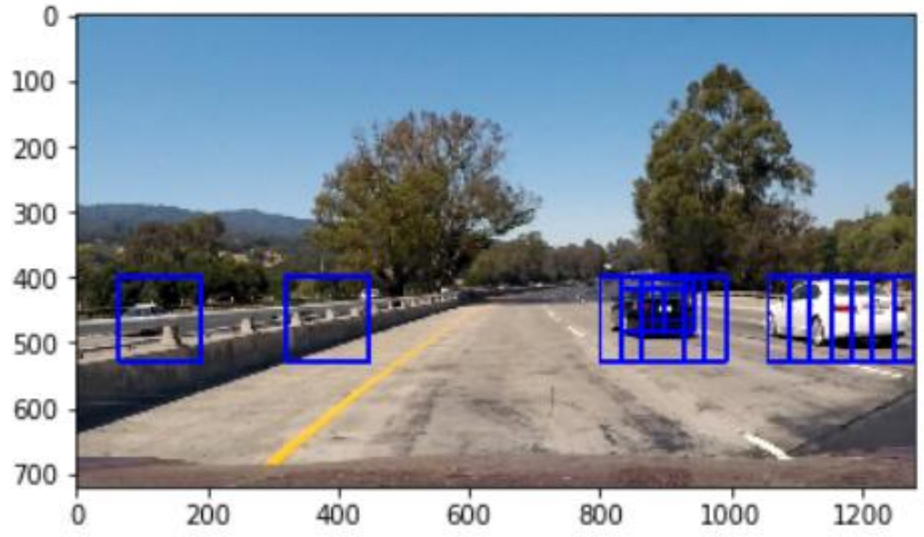
```
class Model(object):
    def __init__(self, model_file=None):
        if model_file is None:
            self.model = LinearSVC()
            self.scalar = None
        else:
            with open(model_file, 'rb') as f:
                self.model, self.scalar = pickle.load(f)
    def save_model(self):
        with open('model.pkl', 'wb') as f:
            pickle.dump((self.model,self.scalar), f)


    def preprocess(self, images):
        features = []
        for i in images:
            features.append(extract_features(i, color_space=color_space,
hog_channel='ALL'))
        print("feature length", len(features[0]), "feature shape",
features[0].shape)
        return np.array(features).astype(np.float64)


    def train(self, X, y, test_size=0.3):
        features = self.preprocess(X)
        self.scalar = StandardScaler().fit(features)
        scaled_X = self.scalar.transform(features)
        stat = np.random.randint(0, 100)
        scaled_X, y = shuffle(scaled_X, y)
        X_train, X_test, y_train, y_test = train_test_split(scaled_X, y,
                                                            test_size=test_size,
                                                            random_state=stat)
        self.model.fit(X_train, y_train)
        score = self.model.score(X_test, y_test)
        log("Testing score: {}".format(score))
    def predict(self, feature):
        return self.model.predict(feature)
```

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 two scales using YCrCb 3-channel HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. Here are some example images:



Because of the HOG feature based sub sampling technique, from each window, features are extracted and the SVM based classifier will predict if the window contains a car or not.

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 have attached video with the attachments.

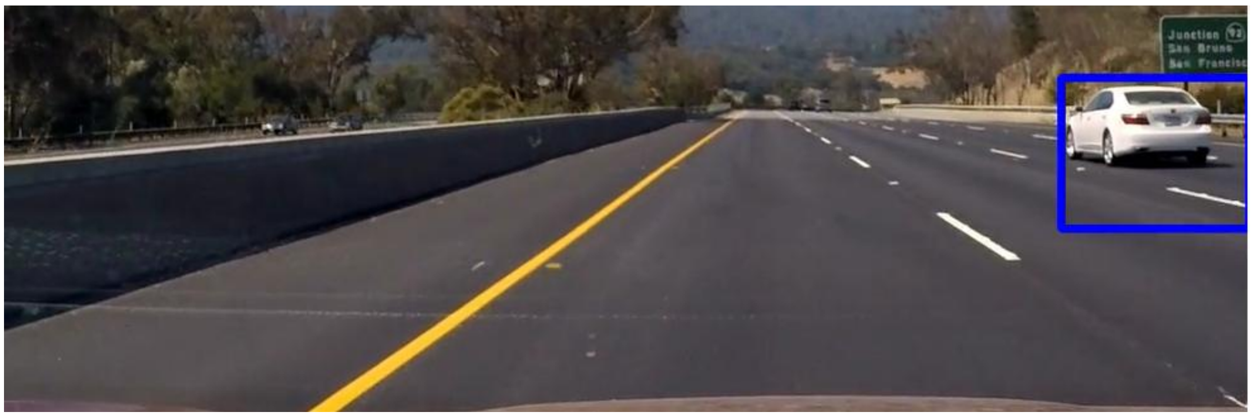
2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

Classifier does not detect third image and also there are multiple false positives. This can be removed by the following techniques:

1. Heat map

I have written a function to combine boxes to prevent false positives. I search the number of cars in every frame and then include the required boundary boxes.
Then the major step of heap map comes to picture. I set a small number of car image count. When the number of cars detected becomes equal to the car image count, then a combined heat map is generated. I put a quota of 4 for the heatmap value. Thus for multiple false positives, only one bounding box is generated using the use of heat maps.
The code is given below:

Please find the final image below after removing all false positives:



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?

Pipeline is able to detect cars properly on the roads.

We can use the code from Udacity project 4 for our application, mainly for lane detection. I have tried that too but it did not add any improvement on our model. Thus it was removed from the final work. But we were also faced with a few problems.

Light conditions affect the way we detect the vehicles. This can be solved by the use of more training data of varying light intensities. Also memory constraints are a big issue in personal laptops. We can use AWS services to improve training speed.

We could use powerful classifier for our implementation to get better results. Current classifier has a few problems with overlapping cases. This can be improved by the following methods. We could use decision trees and also feed forward neural networks to get superior performance.

Pipeline is not real time. We had combined 3 to 5 frames for our application. In practice it doesn’t affect the performance and helps to remove noise. But in cases of fast moving cars it is better to reduce the number of frames to 1 or 2 to get superior performance.

Overlapping can cause a great bit of trouble to our model. A better advanced classifier can fix these issues.