PROJECT 5: VEHICLE TRACKING

Mithi Sevilla - March 23, 2017

INTRODUCTION

For this project I have written a software pipeline to detect vehicles in a video.

To achieve this goal, I have done the following:

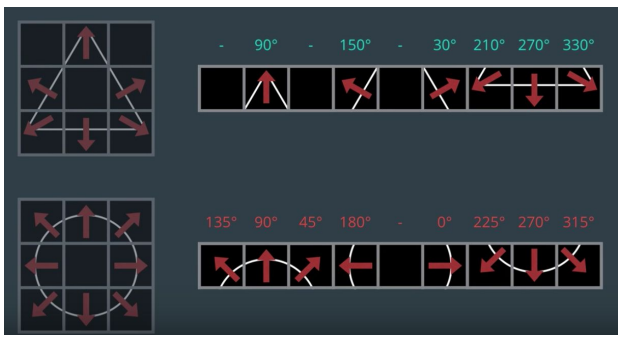
- Performed a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images
- Normalized these features and randomized a selection for training and testing a classifier
- The classifier trained and used for prediction is a Linear SVM classifier
- Implemented a sliding-window algorithm and used my trained classifier to search for vehicles in images
- Ran my pipeline on a video stream
- Created a heatmap of recurring detections frame by frame to reject outliers and follow detected images
- Estimated a bounding box for vehicles detected

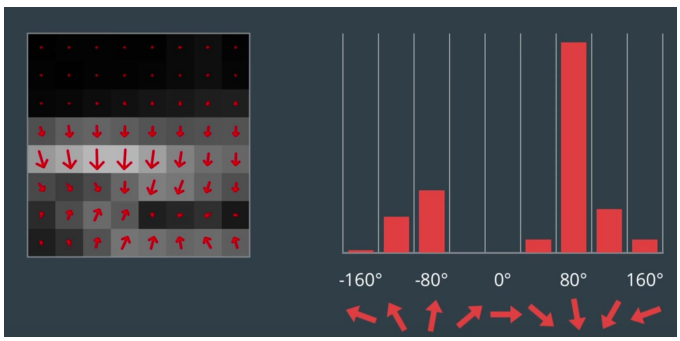
This project includes the following files:

pipeline.ipynb	**featuresourcer.py**	**WRITEUP.pdf**
pipeline_verbose_test.ipynb	**binaryclassifier.py**	**project_video.mp4**
feature_sourcer_test.ipynb	**slider.py**	**project_video_output_conser.mp4**
classifier_training.ipynb	**heatmap.py**	**project_video_output.mp4**
classifier_test.ipynb	**helpers.py**	**svc.pkl / svc2.pkl**
slider_test.ipynb / slider_test2	**test_video.mp4**	**scaler.pkl / scaler2.pkl**
heatmap_test.ipynb	**test_video_verbose_output.mp4**	
test_images/	**test_video_output.mp4**	

FEATURE EXTRACTION

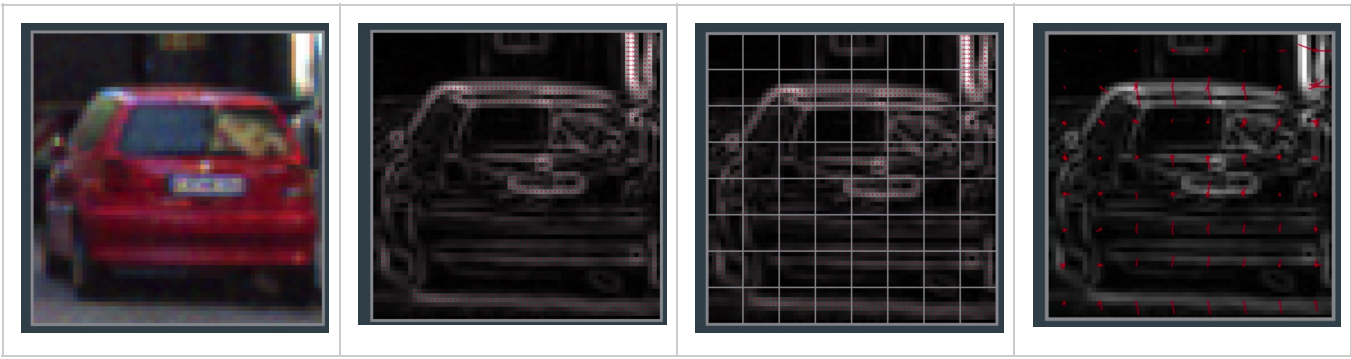
THE IDEA BEHIND HISTOGRAM OF ORIENTED GRADIENTS





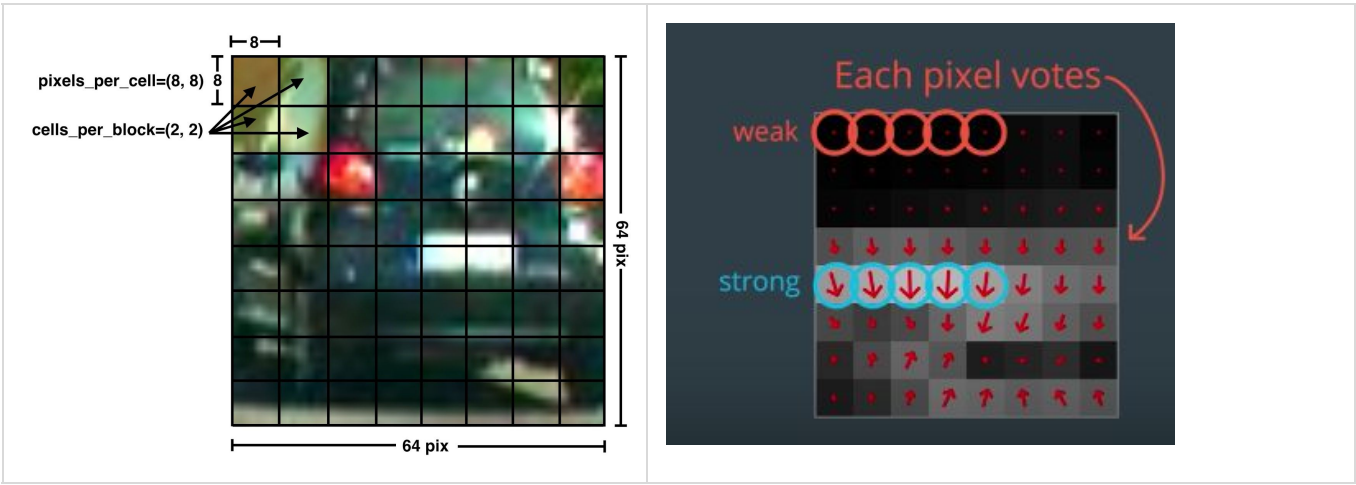
Taken from one of Udacity's lecture videos

A class of objects such as a vehicle that we are training vary so much in color. In contrast. structurals cues like shape give a more robust representation. Gradients/derivative of specific directions captures some notion of shape. To allow for some variability in shape, we'll use the a Histogram of Oriented Gradients (HOG) instead.



Taken from one of Udacity's lecture videos

The idea of HOG is instead of using each individual gradient direction of each individual pixel of an image, we group the pixels into small cells of n x n pixels. For each cell, we compute all the gradient directions and grouped into a number of orientation bins. We sum up the gradient magnitude in each sample. So stronger gradients contribute more weight to their bins, and effects of small random orientations due to noise is reduced. This histogram gives us a picture of the dominant orientation of that cell. Doing this for all cells gives us a representation of the structure of the image. The HOG features keep the representation of an object distinct but also allows for some variations in shape.



Taken from one of Udacity's lecture videos

We can specify the number of **orientations**, **pixels_per_cell**, and **cells_per_block** in computer the hog features of a single channel of an image. The number of orientations is the number of orientation bins that the gradients of the pixels of each cell will be split up in the histogram. The **pixels_per_cells** is the number of pixels of each row and column per cell over each gradient the histogram is computed. The **cells_per_block** specifies the local area over which the histogram counts in a given cell will be normalized. Having this parameter is said to generally lead to a more robust feature set. We can also use the normalization scheme called **transform_sqrt** which is said to help reduce the effects of shadows and illumination variations.

CHOOSING FEATURES AND PARAMETERS

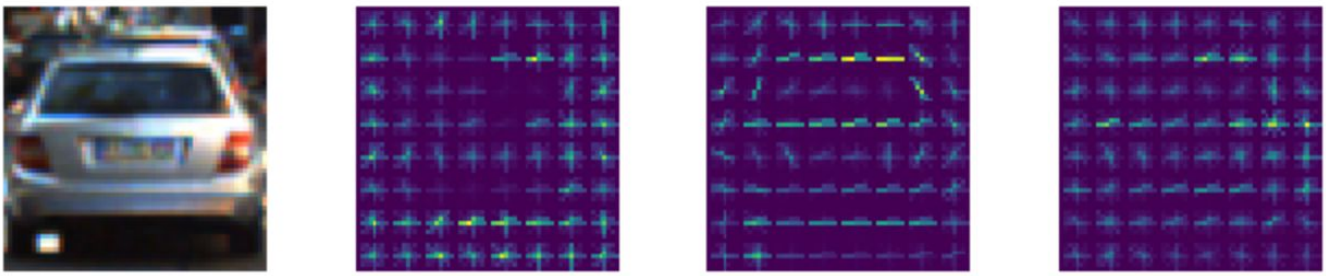
I chose to perform HOG to all channels of the image in HSL format. The *Hue* value is its perceived color number representation based on combinations of red, green and blue, the *Saturation* value is the measure of how colorful or or how dull it is, and *Lightness* how closer to white the color is. My intuition is that shape of the change in all these measurements provide a good representation of the shape of a vehicle. I chose an **8 pixel by 8 pixel cell** and **2 cell by 2 cell block** as inspired by the examples from the lectures. This also seems like reasonable parameters as our training set is composed of 64 pixel by 64 pixel images. I chose **12 orientation bins** as it was typical to choose a number between 6 and 12 for this parameter. I also chose to enable the tranfrom_sqrt normalization scheme to help reduce the effect of shadows and illumination variations.

```
I have made a FeatureSourcer class for this.  Check featuresourcer.py for
implementation details, as well as feature_source_test.ipynb for sample usage.
```

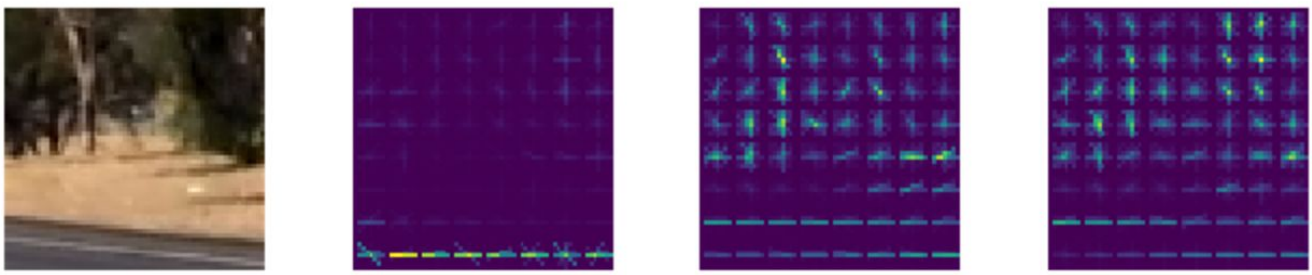
The actual number of features is the total number of block positions times the number of cells per block, times the number of orientations. Having an 8 pixel by 8 pixel cell with a 2 cell x 2 cell block at a 64 by 64 single channel image will have a total of 7 x 7 block positions, so the number of features per channel is **7 x 7 x 2 x 2 x 12 = 2,352.** Having three channels gives us a total of **7,056 features**.

```
sourcer_params = {
  'color_model': 'hls',               # HLS, HSV
  'bounding_box_size': 64,            #
  'number_of_orientations': 12,       # 6 - 12
  'pixels_per_cell': 8,               # 8, 16
  'cells_per_block': 2,               # 1, 2
  'do_transform_sqrt': True
}

start_frame = imread("vehicles/KITTI_extracted/5364.png")
sourcer = FeatureSourcer(sourcer_params, start_frame)

f = sourcer.features(start_frame)
rgb_img, h_hog_img, s_hog_img, l_hog_img = sourcer.visualize()

new_frame = imread("non-vehicles/Extras/extra4846.png")
f = sourcer.features(new_frame)
rgb_img, h_hog_img, s_hog_img, l_hog_img = sourcer.visualize()
```

Vehicle: visualization of the HOG features for Hue, Saturation, and Lightness respectively



Non-Vehicle: visualization of the HOG features for Hue, Lightness, and Saturation respectively



CLASSIFIER TRAINING

I have used a total of 8,792 samples of vehicle images and 8,968 samples of non-images in my data set. This data set is preselected by Udacity that come from the GTI vehicle image database and the KITTI vision benchmark suite (http://www.gti.ssr.upm.es/data/Vehicle_database.html, http://www.cvlibs.net/datasets/kitti/) These images are scaled down to 64 pixels by 64 pixels each as mentioned before. After loading all images in memory, I have used my class **featureSourcer** to extract the features of all images at the data set. I have used the **StandardScaler** from SKlearn to make a scaler based on the mean and variance of all the features in the data set. The StandardScaler normalizes features by removing the mean and scaling it to unit variance. We use this scaler to transform the featured the raw features from our **FeatureSourcer** before feeding the scaled feature to the our classifier for training or predicting. We do this as a safety measure because it is a common requirement for machine learning estimators, as they might behave badly if an individual feature do not look like the standard normally distributed data (see http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html).

```
unscaled_x = np.vstack((vehicles_features, nonvehicles_features)).astype(np.float64)
scaler = StandardScaler().fit(unscaled_x)
x = scaler.transform(unscaled_x)

y = np.hstack((np.ones(total_vehicles), np.zeros(total_nonvehicles)))
```

I used 80% of the data set to train the classifier. The remaining 20% is used to determine the accuracy of the classifier. To randomize the splitting of the data we used the built-in **train_test_split** function from **sklearn** and fed it a random number between one and one hundred.

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2,
                                                    random_state = rand.randint(1, 100))
```

The classifier algorithm we used is **Linear Support Vector Machine** as recommended by Udacity for this project. It has advantages including being effective in high dimensional spaces even when the number of dimensions is almost as large or even larger than the number of sample such as this case. It is also said to be memory efficient and versatile (http://scikit-learn.org/stable/modules/svm.html)

```
svc = LinearSVC()
svc.fit(x_train, y_train)
accuracy = svc.score(x_test, y_test)
```

Here are some stats from training this classifier:

```
Accuracy: 0.9859
Loading images to memory: Time Taken: 9.9
Extracting features Time Taken: 1781.44
Scaling features: 4.15
Training classifier: 30.93
```

Please check **classifier_training.ipynb** for more details about how I trained this the classifier. To make it cleaner I have wrapped our **svm** model and **scaler** to a class called **BinaryClassifier** to predict vehicle images from non-vehicle images given a vector of unscaled features which can be found in **binaryclassifier.py**. Check **classifier_test.ipynb** for sample usage.

```
class BinaryClassifier:

  def __init__(self, svc, scaler):
    self.svc = svc
    self.scaler = scaler

  def predict(self, f):
    f = self.scaler.transform([f])
    r = self.svc.predict(f)
    return np.int(r[0])
```

SLIDING WINDOWS

I have to implement a method of searching for vehicles in an image. We can get a subregion of an image and run that classifier in that region to see if that patch contains a vehicle. Firstly, we have to consider that getting the HOG features is extremely time consuming, so instead of getting the HOG image for each patch region with have many overlaps with each other, we extract hog features of the whole frame of an image and then we subsample that extraction for each sub window of that image.

We create a class called **slider** for this. What the slider needs from us is that we feed a **frame**, a **window size (ws)** in pixels and the **starting vertical position (y)** (y axis in pixels where we want to search for the a vehicle. It then outputs a list of a locations of where the vehicles are found. The locations are represented by the top corner of the subregion and the length of the sides in pixels. We initialize a **slider** instance with a **featureSourcer** and **classifier** and the number of pixels horizontally to increment a sliding window (**increment**) horizontally on the x axis.

```
slider = Slider(sourcer = src, classifier = cls, increment = 8)

bounding_boxes = slider.locate(frame = this_frame, window_size = ws, window_position = wp)
img = put_boxes(this_frame, bounding_boxes)
```

```
ws = 180, 100, 120, 140, 180, 210 # window sizes (pixels window height)
wp = 360, 390, 390, 390, 390, 390 # window positions (y_start)
```

Sample 1

Sample 2

Sample 3

How it does this is that, for each frame, it slices a **strip** of the frame which is the subregion of interest based on the given starting vertical position (**y**) and window size (**ws**). It resizes this strip so that the window size is 64 pixels in height which is the size of the images on our data set that we feed to our classifier when trained. We slide the window over that strip at a given increment. We then get the HOG features of each of the window and check with our classifier if a vehicle is there. I decided that an overlap of 75% or 8 pixels increment of 64 pixels was good to search for vehicles without being too much of an overkill. Window sizes were squares of 80, 120, 160, 180 and 210 pixels per side and were decided eyeing the sizes of the vehicles on the video. The strip of window where the search was performed was also by eyeing the vehicles on the video and taking into consideration that we should only search below the horizon as we don't expect to see vehicles on the sky.

Check **slider.py** for implementation details, as well as **slider_test2.ipynb** and **slider_test2.ipynb** for sample usage.

```
def locate(self, frame, window_size, window_position):

  y, ws, boxes = window_position, window_size, []
  scaler, strip = self.prepare(frame, y, ws)

  self.sourcer.new_frame(strip)
  x_end = (strip.shape[1] // self.h - 1) * self.h

  for resized_x in range(0, x_end, self.i):

    features = self.sourcer.slice(resized_x, 0, self.h, self.h)

    if self.classifier.predict(features):
      x = np.int(scaler * resized_x)
      boxes.append((x, y, ws))

  return boxes
```

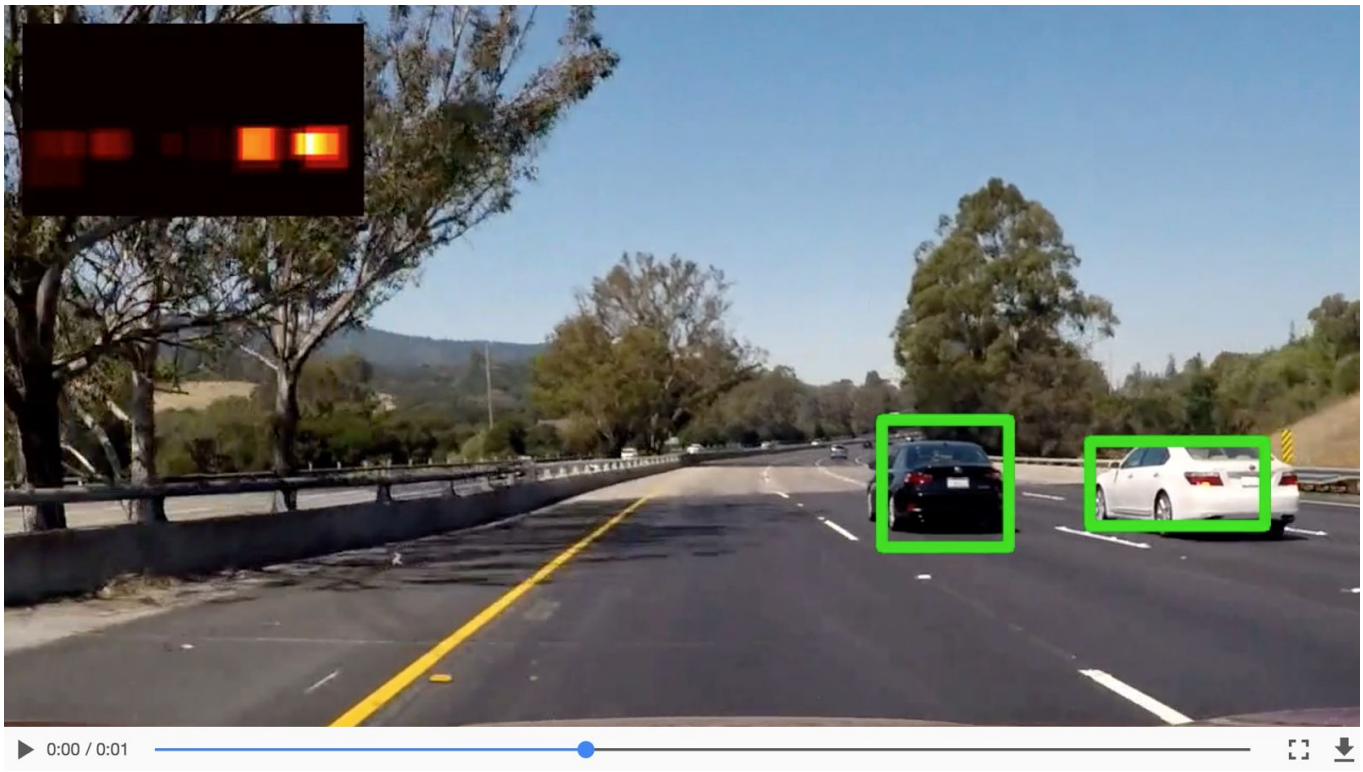
```
def prepare(self, frame, y, ws):
```

```
    scaler = ws / self. h
    y_end = y + ws
    w = np.int(frame.shape[1] / scaler)

    strip = frame[y: y_end, :, :]
    strip = cv2.resize(strip, (w, self.h))
    self.current_strip = strip

    return scaler, strip
```

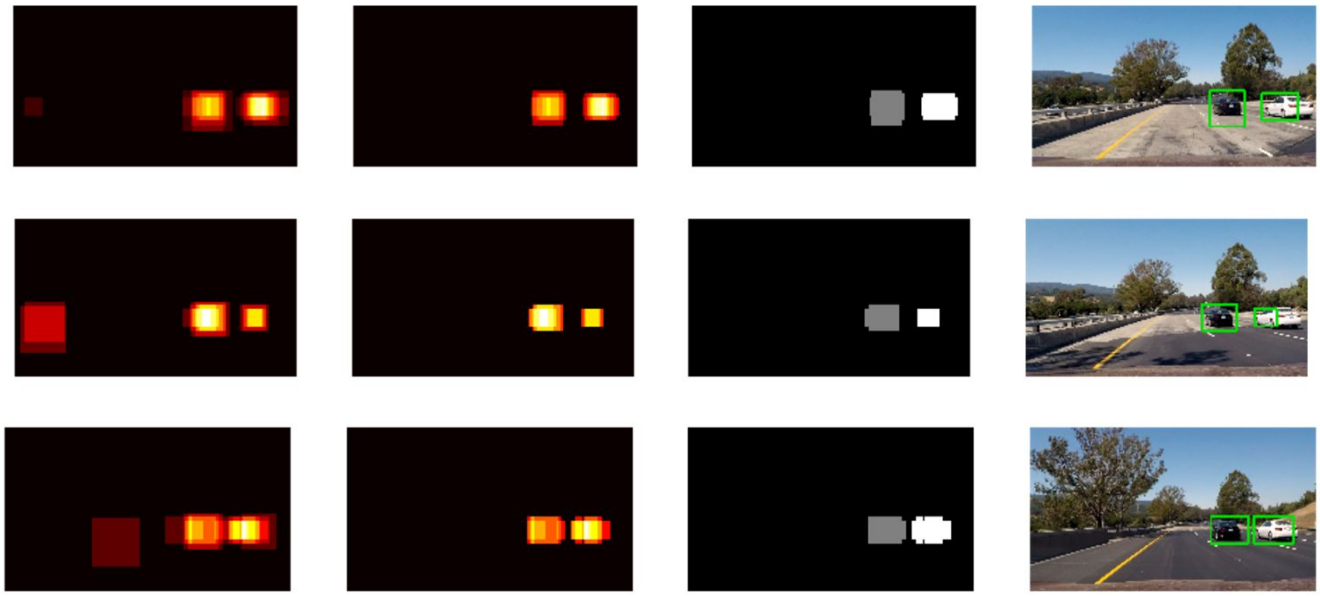
HEATMAPS

Check **heatmap.py** for implementation details, as well as **heatmap_test.ipynb** for sample usage.

Given a few consecutive frames, it can be noticed that there are overlapping detections and false positive detections are spaced out. We can build a heatmap to combine overlapping detections and remove false positives. To make a heat map we start with a blank grid and "add heat" (+1) for all pixels within windows where positive detections are reported by the classifier. The "hotter" the parts, the more likely it is a true positive, and we can impose a threshold to reject areas affected by the false positives. We have integrated a heat map over several frames of video. Areas with multiple detections get "hot" while transient false positives stay "cool". We have made a **HeatMap** class to implement this. We initialize with a size of the **HeatMap** by feeding a sample **frame**, the **threshold**, and its "**memory** size" or how many frames it will keep before rejecting the oldest frame.




```
ws = 80, 120, 160, 180, 210  # window sizes
wp = 410, 390, 380, 380, 400 # window positions

this_heatmap = HeatMap(frame = frame1, thresh = 3, memory = 12)
this_heatmap.reset()

for sz, pos in zip(ws, wp):
  bounding_boxes = slider.locate(frame = this_frame, window_size = sz, window_position = pos)
  this_heatmap.update(bounding_boxes)

mp, tmp, lmp = heatmap.get()
labeled_img = heatmap.draw(this_frame)
```



I used **scipy.ndimage.measurements.label()** to identify individual blobs in the heatmap, each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected.

```
def update(self, boxes):

  if len(self.history) == self.memory:
    self.remove(self.history[0])
    self.history = self.history[1:]

  self.add(boxes)
  self.history.append(boxes)

def add(self, boxes):
  for box in boxes:
    x1, y1, x2, y2 = box_boundaries(box)
    self.map[y1: y2, x1: x2] += 1

def remove(self, boxes):
  for box in boxes:
    x1, y1, x2, y2 = box_boundaries(box)
    self.map[y1: y2, x1: x2] -= 1

def get(self):
  self.do_threshold()
  self.label()
  return self.map, self.thresholded_map, self.labeled_map

def do_threshold(self):
  self.thresholded_map = np.copy(self.map)
  self.thresholded_map[self.map < self.thresh] = 0

def label(self):
  labeled = label(self.thresholded_map)
  self.samples_found = labeled[1]
  self.labeled_map = labeled[0]
```

PIPELINE

For my pipeline, first I had to define my features which was HOG features of the three HSL channels. I defined the parameters for this hog features. I instantiated a **FeatureSourcer** which will be used to extract the features of a given image. I loaded a linear SVC classifier (with scaler) that I had trained earlier with my selected features which will be used predict if the features of an image is a vehicle. Loading the **FeatureSourcer** and **BinaryClassifier** I instantiated a **Slider** which will be used to locate sub regions within a particular strip of the frame of a sub region's particular size n where the classifier predicts that a vehicle is located. We slide this slider within different slices/strips of the frame and different sizes of subregions. We accumulate these subregions that are positive accumulated over a series of consecutive frames and feed it to a **HeatMap.** Areas with multiple detections get "hot" while transient false positives stay "cool" as we impose a threshold to this **HeatMap**. The remaining "Hot" regions and draw a bounding box for each "hot region" which implies a vehicle.

Check **pipeline.ipynb**

```
sourcer_params = {
  'color_model': 'hls',                # HSL, HSV
  'bounding_box_size': 64,             #
  'number_of_orientations': 12,        # 6 - 12
  'pixels_per_cell': 8,                # 8, 16
  'cells_per_block': 2,                # 1, 2
  'do_transform_sqrt': True
}

svc = joblib.load('svc.pkl')
scaler = joblib.load('scaler.pkl')
temp_frame = imread("test1.jpg")
temp_img = imread("vehicles/KITTI_extracted/5364.png")

src = FeatureSourcer(sourcer_params, temp_img)
cls = BinaryClassifier(svc, scaler)
slider = Slider(sourcer = src, classifier = cls, increment = 8)
heatmap = HeatMap(frame = temp_frame, thresh = 20, memory = 60)

ws = 180, 100, 120, 140, 180, 210 # window sizes (pixels window height)
wp = 360, 390, 390, 390, 390, 390 # window positions (y_start)

def pipeline(this_frame):

  for sz, pos in zip(ws, wp):
    bounding_boxes = slider.locate(frame = this_frame, window_size = sz, window_position = pos)
    heatmap.update(bounding_boxes)

  labeled_img = heatmap.draw(this_frame)
  return labeled_img
```

PROBLEMS ENCOUNTERED

I had three major headaches in this project. The first is that It was hard to estimate the position of the “strip” **(y_start)** of the window I will perform the sliding-window technique as well as the size of the window. A large window like 240 pixels by 240 pixels would perform consistent false positives. The 2nd major problem was tweaking the “memory” (buffer of frames) and “threshold” of the **HeatMap** in order to reject false positives but not reject true positives. The biggest pain was that extracting the HOG features, it was really time consuming, so it took me a long time to see the results of tweaking the parameters. Even up to now, it can be tweaked further for improvements. My pipeline would fail in a lot of conditions particularly when the camera of the car is mounted at a different position since I only used the sliding window technique in a few particular strips of one video frame and only a few window sizes. It would also fail at a low frame per second video since this video keeps a memory of around 10 frames for second given 7 window size- window positions.