BOTHELL

Vehicle Detect

Adam Ali, Taylor Brady

CSS 487 A

December 5, 2018

Final Project

Table of Contents

Acknowledgements

Thank you to Dr. Clark Olson for teaching CSS 487 A: Computer Vision at the University of Washington Bothell and inspiring us to pursue specialized applications of computer science.

Overview

Project Goal

Vehicle Detect is a project that explores object recognition techniques to locate vehicles in images, such as those from traffic and dash cams. Drawing a simple bounding box around each vehicle in an image is the objective.

Development Environment

Vehicle Detect is written purely in C++ using OpenCV. In fact, both computer vision and machine learning concepts are applied here.

Dataset

We start with about [9K labeled images from Udacity]. We want to build a simple model that has a basic understanding of what a *"car"* and *"non-car"* typically look like. It turns out that the included labels are already *"car"* and *"non-car"*. Each training image is 64×64 pixels.

[4]

Figure 0: Examples from each class of the Udacity dataset.

Pre-processing

In order for a computer to obtain this understanding, we need some quantitative representation of the images that can be more easily comparable. We achieve this using *Histogram of Oriented Gradients*. When we obtain this numerical profile of *"cars"* and *"non-cars"*, we need to find a way to plot these representations and use classification techniques to label new test images based on similarity to already labeled data. We achieve this using *Support Vector Machine*.

Histogram of Oriented Gradients (HOG)

Feature Descriptors

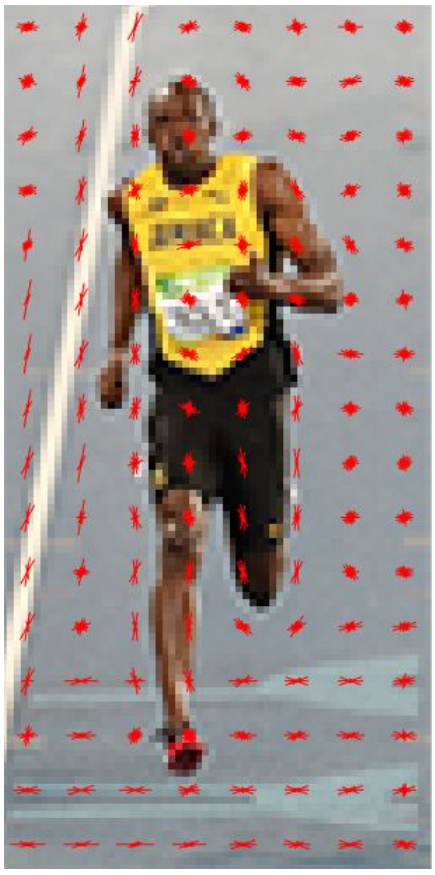
The goal of our pre-processing step is to read the pre-labeled training images and extract meaningful features from them. We hope that these meaningful features will yield the "general aspect" of the images. That is, we hope to obtain *Feature Descriptors*. A *Feature Descriptor* is a simplified representation of an image that reveals only useful information about the nature of the image and discards extraneous information. It is a minimal set of data that reveals the essence of what qualifies the object to be what it is. For example, given an image of a banana we might care more about the curvature, smooth texture and predominately yellow hue rather than the bowl it might be sitting in or the background. It is ideal to have training images with the least extraneous information as possible.

Gradient Magnitude and Orientation

HOG measures the distribution (histogram) of possible *Gradient Orientations*. By default, images have an *Intensity* at each pixel. Typically this is a vector in color images (RGB) or a single value in gray-level images. A *Gradient* is the change in *Intensity*, or the derivative. In our case, we can measure the *Gradient* in x and y to reveal two important bits of information for any patch of an image: how much the *Intensity* is changing overall (*Gradient Magnitude)* and in which direction (*Gradient Orientation)*.

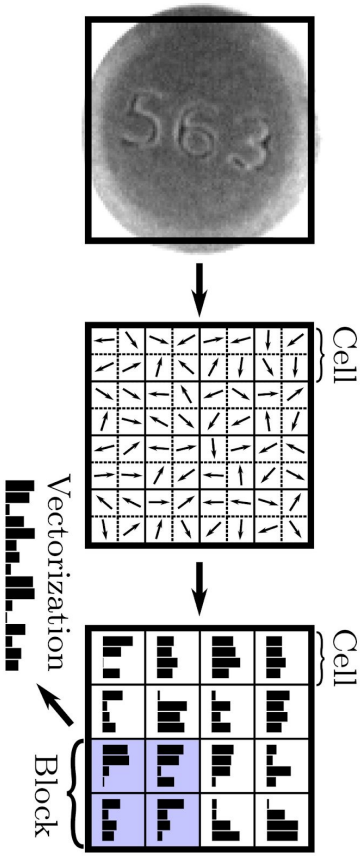
Edges have significant *Gradient Magnitudes* because they have sharp changes in *Intensity* relative to their surrounding pixels. A color image would have three *HOGs*, one for each channel (RGB). A gray-level image only needs one. Images should be converted to grayscale unless the color is important to the objection recognition. In our case, the color is NOT important to determining what is or isn't a car.

Note that in a three-channel *HOG*, there would be three *Gradient Magnitudes* and *Gradient Orientations* per pixel. In practice, we would use the greatest *Gradient Magnitude* per pixel and its corresponding *Gradient Orientation*. The histogram measures the total *Gradient Magnitudes* for each *Gradient Orientation.* In other words, it measures shape characteristics (from vertical to horizontal lines) by weighting them. This yields an approximate numerical representation of the shape of the object. Realize that the *Gradient Magnitude* uses the *Gradient* in x and y to find a *Gradient* for both axes, as we shall see in a moment. *HOG* was originally devised for pedestrian detection [as demonstrated by Dalal and Triggs](). The idea was that pedestrians will have similar *HOGs* because we typically see a certain ratio of vertical *Gradient Orientations* around the legs/arms and horizontal *Gradient Orientation* around the torso.



[8]

Figure 1: HOG for pedestrian detection. Each bundle of red lines is a plot for the gradients in its respective patch. Lines are angled according to *Gradient Orientation*, with their size according to *Gradient Magnitude*. Notice the



[0]

Figure 2: A *HOG* visualization for a medicinal tablet. The same principles apply despite the subject in question.

predominantly vertical lines around the legs.

In general, a *HOG* will take the form of a $9D$ vector (array with 9 bins). Each dimension/bin accounts for some angle or *theta* that a *Gradient Orientation* can take. In theory, this can be anywhere from 0 to 360. In practice, it turns out that we can consider opposing angles to be equivalent for our purposes as the lines drawn at these angles have the same slope. To represent 0 to 180 in our 9-dimensional vector, we can distribute the bins as $\{ 0, 20, 40, 60, 80, 100, 120, 140, 160 \}$. Degree 0 is considered equivalent 180. You might be wondering how we can use this since many *Gradient Orientations* will NOT fall precisely on nice even angles like this, e.g. 64 instead of 60. There is a technique for this in a later step.

Gradient Magnitude (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)	*Gradient Magnitude* (Total)
0	20	40	60	80	100	120	140	160

Gradient Orientation θ *(theta)*

Figure 3: Data structure for *Histogram of Oriented Gradients*.

Kernel Convolution

To get *Gradient* information from an image, we must use *kernel convolution*. A image is essentially a matrix of values. Similarly, a *kernel* takes the same form but is intended for use as a filter. The *kernel* is overlaid on each each pixel in the reference image, and its value is modified using information from its neighbors and the values in the *kernel*. We use these to accomplish linear transformations, blurring, sharpening, etc.

Every pixel in the reference image is replaced with the sum of products of overlapping elements between the kernel and the reference image, when the kernel is flipped (rows and columns). For example, where the first matrix is the kernel and the second is the reference image, where there is a perfect overlap to find the new value for the center pixel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

Figure 4: Kernel convolution, where first matrix is the kernel and the second is the image, such that the result is stored in the image at [2,2] when the two matrices are overlaid perfectly.

Imagine that one of our images, a *"car"* or *"non-car"* is given. We will use two kernels to determine the *Gradients* in x and y:

Gradient with respect to x*:*

-1	0	1

Gradient with respect to y*:*

-1
0
1

Figure 5: The gradient kernels.

Notice that these kernels replace the pixel they center on with the delta between its neighbors (left/right for x and above/below for y). For potentially better results, we can use the *Sobel operator* to apply smoothing/blurring while calculating *Gradients*. This can help with noise from the camera or lighting.

The Gradient Unit Circle

If we convolve our reference image matrix with each kernel separately, we will have two new matrices for *Gradient* in x and y. For each set of corresponding elements in these matrices, we can derive two more new matrices for *Gradient Magnitude* and *Gradient Orientation*. For each element $[r, c]$:

- *Gradient* in x and y: G_x and G_y
- *Gradient Magnitude*: $Gmag$
- *Gradient Orientation:* θ

It helps to use the unit circle as a conceptual representation of what *Gradient* means. Imagine that every element $[r, c]$ has the following unit circle:

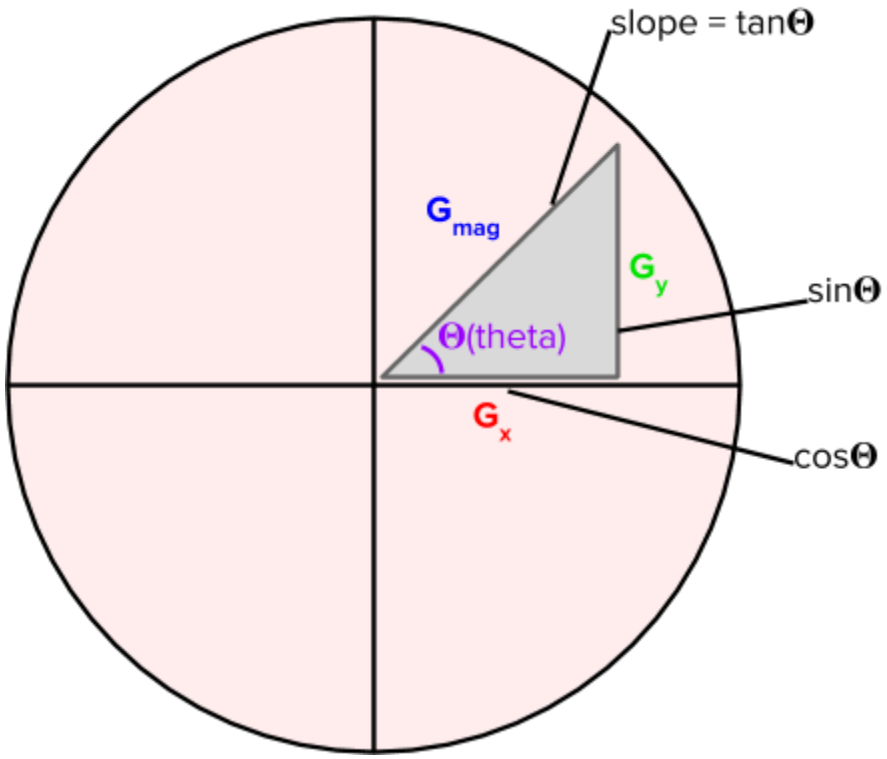


Figure 6: The Gradient Unit Circle.

With the *Gradients* forming two legs of the triangle, we can use the Pythagorean theorem to find the hypotenuse of this triangle, which solves half of our problem by giving *Gradient Magnitude* (accounts for both x and y for a general value). The other half is *Gradient Orientation*.

$$Gmag = \sqrt{G_x^2 + G_y^2}$$

Figure 7: Calculating the *Gradient Magnitude*.

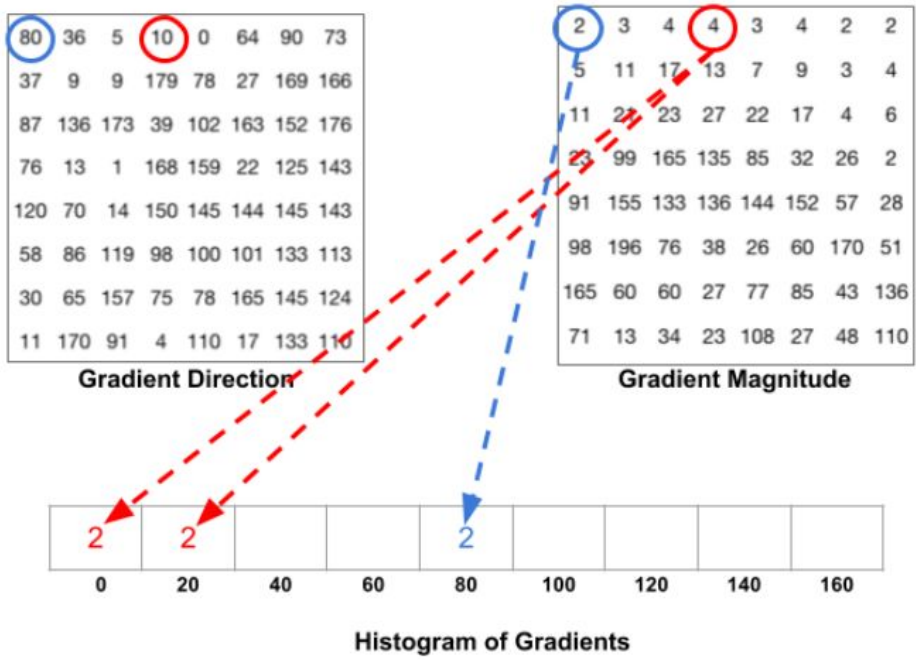
Because the $tan(\theta)$ function yields the slope of the hypotenuse given *Gradient Orientation*, we can use the inverse to find θ when we know the slope of the hypotenuse. The slope is found easily with the ratio of G_y to G_x:

$$\theta = arctan(\frac{G_y}{G_x})$$

Figure 8: Calculating the *Gradient Orientation*.

Populating the Histogram

Recall that our $9D$ vector has a bin for every *Gradient Orientation* between 0 and 160, split by 20 degree increments. Naturally, θ will almost certainly produce *Gradient Orientations* that do NOT fall nicely into one of these bins (i.e. end in a 0). We can remedy this by using a weighting technique:



[8]

Figure 9: Visualizing the weighting technique.

If a *Gradient Orientation* happens to match precisely with one of our bins, then the corresponding *Gradient Magnitude* for that particular element can be added to the sum. Otherwise, we find the bin closest to it and measure the absolute distance. This distance is divided by the distance from the closest bin to the bin on "the other side" (which is always 20). This gives a weight which is applied to the corresponding *Gradient Magnitude* to provide the amount to add to that bin. The remainder is added to the other bin.

For example, say that at some element we have a *Gradient Orientation* of 124 and a corresponding *Gradient Magnitude* of 50.

- The bin closest to 124 is 120.
- This yields a distance of 4.
- The bin on the other side is 140.
- The distance between 120 and 140 is 20.
- $\frac{4}{20}$ is 20%.
- The value added to the bin at 120 is 20% of the *Gradient Magnitude* or 10.
- The value added to the bin at 140 is the remainder or 40.

Gradient Magnitude (Total)	*Gradient Magnitude (Total)*	*Gradient Magnitude (Total)*	*Gradient Magnitude (Total)*	*Gradient Magnitude (Total)*	*Gradient Magnitude (Total)*	**10**	**40**	*Gradient Magnitude (Total)*

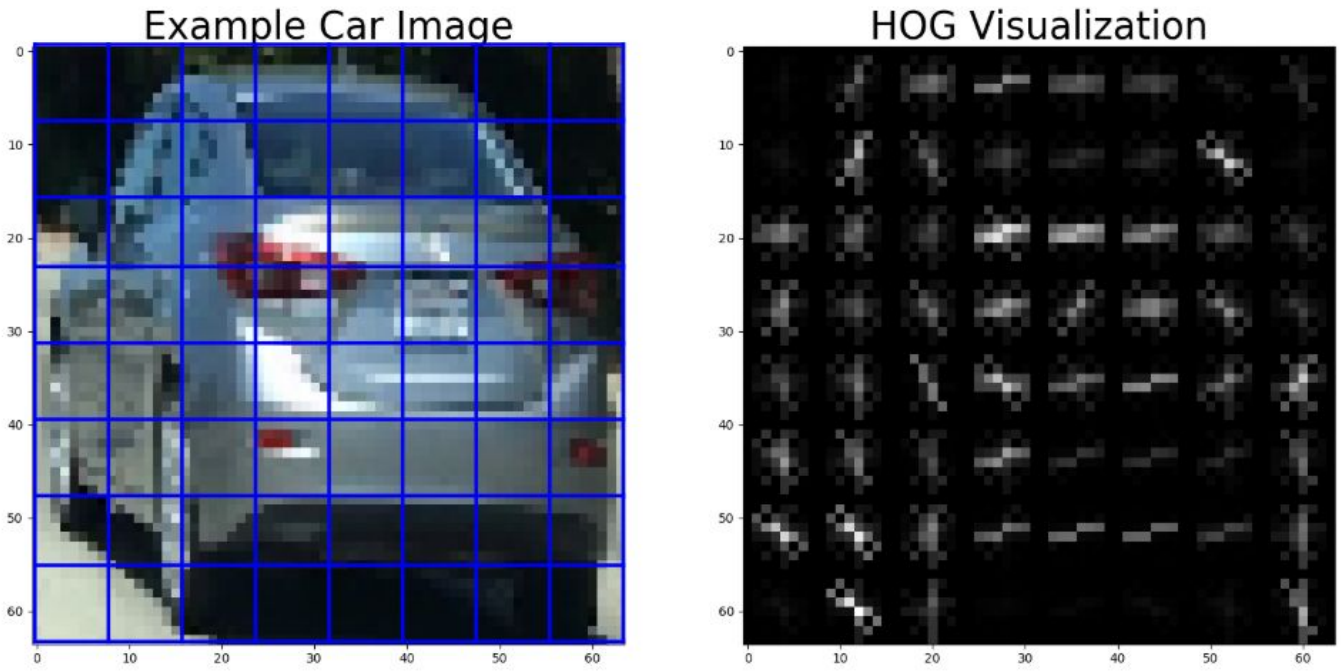
0 20 40 60 80 100 120 140 160

Gradient Orientation θ *(theta)*

Figure 10: Example application of weight/bias for *Gradient Magnitude*.

Using Patches

By default, this $9D$ vector could be based on the entire image as a whole by applying the above process to each pixel in the image. However, the *HOG* can be made more specific with regard to feature extraction by generating a $9D$ vector for patches in the image as opposed to the entire image. This was shown previously in `Figure 1` with pedestrian detection. Similarly for a car sample:



[3]

Figure 11: Visualizing HOG across patches.

Each corresponding patch is a plot of the $9D$ *HOG* for the patch itself. There are 9 lines, one for each dimension, whose angles and sizes correspond to the *Gradient Orientations* and *Gradient Magnitudes* respectively. Usually the result is that areas with predominantly vertical or horizontal features will display as predominantly vertically or horizontally biased plots.

Using a well-sized patch, the *HOGs* will be more sensitive to precise features in the image by squeezing out more information in general (by virtue of having multiple vectors for each patch instead of one master vector).

It has been empirically shown that 8×8 patches work well for this application (Dalal)[9], based on the general scale of the features we are interested in. These include details like curvatures of rooflines, lamps and wheels. For any given image, this would give us a *HOG* for every 8×8 patch.

This is fine-tuned approach is what really leverages the power of *HOG*. If we were to take each of these vectors and stitch them together into a massive $1 \times 8n$ vector, where n is the number of 8×8 patches, we would have a strong numerical representation of the “general aspect” of our sample.

Normalization

In a vacuum, all of the techniques discussed so far would appear to be sufficient. In the real world, images are subject to variations in lighting. *Gradients* are affected by the overall brightness or darkness of the image, which in turn means that *Gradient Magnitudes* will scale with the lighting. This means that the *HOGs* will scale as well, making two *HOGs* for the same image but with sufficiently different lighting a non-match. To mitigate this, we use a normalization technique as follows:

The L^2 norm of a vector renders all the elements to values relative to the vector length. It can be thought of as measuring the legs of a triangle as values relative to the hypotenuse, by taking the leg value divided by the hypotenuse value. More generally, dividing all elements by the vector length (square root of the sum of squares of the vector). For all elements in vector V :

$$L^2(V_i) = V_i \div \sqrt{\sum_{i=0}^{|V|-1} V_i^2}$$

Figure 12: The L^2 norm of a vector.

There are a number of ways to apply this normalization. We can choose to normalize each 8×8 patch (one $9D$ vector at a time), or use some batch normalization process. It helps to have more elements to work with when calculating our normalizer (the vector length). In this case, we normalize 16×16 patches, or four 8×8 patches, which works out to normalizing one 1×36 vector at a time. Keep in mind that unlike the 8×8 patches, these 16×16 patches overlap in that they are offset by only 8 pixels (see `Figure 11`). All training images are in a standardized 64×64 size. This means that, after all of this pre-processing, we will have a 1×1764 normalized *HOG* for each training image. This is according to the following facts:

- We have a 64×64 image.
- We generate a *HOG* for every 8×8 patch.
- Each *HOG* is a 1×9 vector.
- When normalizing, we use overlapping 16×16 patches that concatenate the four 8×8 patches inside.
- Each normalized 16×16 patch has a 1×36 normalized vector.
- The 16×16 patch is offset by 8 pixels at a time, meaning that in our 64×64 image there are 7 vertical and horizontal positions for this patch size. This gives us 49 16×16 patches in the 64×64 image (see next figure).
- 49 1×36 vectors concatenated equates to a 1×1764 vector.

This massive vector serves as a singular datapoint for the training image. The next step is to plot these vectors to build a statistical model of *“cars”* and *“non-cars”*, such that new data can be classified as one or the other.

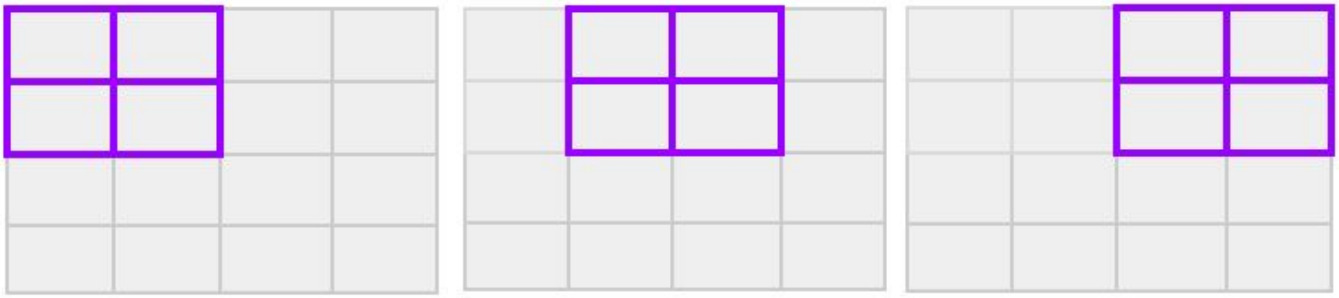
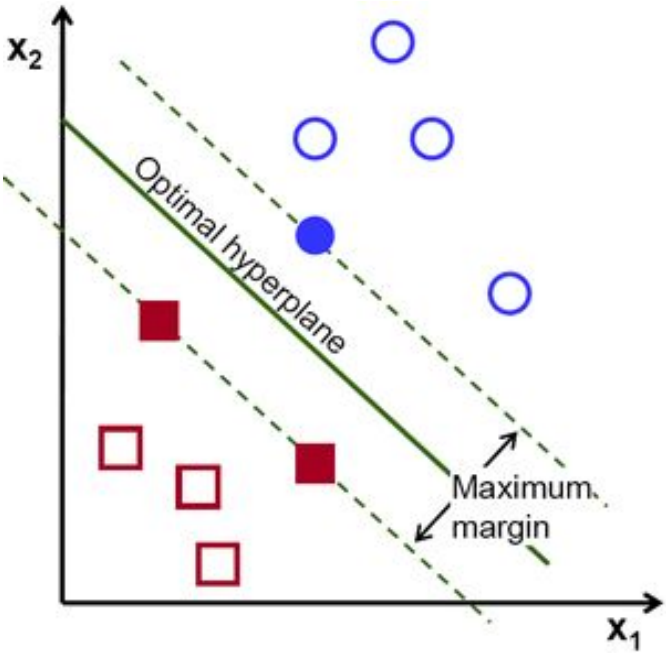


Figure 13: Sequencing of overlapping patches.

Support Vector Machine (SVM)

Definition



[5]

Figure 14: A *Support Vector Machine* and its *Optimal Hyperplane*.

As defined by OpenCV documentation:

> A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (*supervised learning*), the algorithm outputs an optimal hyperplane which categorizes new examples.

Consider the above plot. The objective of the *SVM* is to plot the data and calculate a linear boundary between the two classes of data, assuming that the data is linearly separable. The techniques described in the previous section, if implemented appropriately, should meet this precondition. This linear boundary, also known as a decision boundary or *Optimal Hyperplane*, should be a line that maximizes the minimum distance to any datapoint. We don't want the line to have a near-miss with any particular point as it will be sensitive to noise and increase our chances of incorrect classifications. The greater the margin of empty space around the *Optimal Hyperplane*, the better.

Note that the *SVM* can have dimensions greater than two. It becomes difficult to imagine an *Optimal Hyperplane* beyond a $3D$ *SVM*, but the algorithm can still generate a decision boundary. This is implemented behind the scenes in OpenCV. The procedure to find *Optimal Hyperplanes* is rather involved.

Notice that in the above example there are two dotted parallel lines flanking the *Optimal Hyperplane*. These define the margin between it and the nearest data points on either side. The flanking margin lines are bounded to these data points. They are referred to as *Support Vectors* and are generally the most extreme data points on the interiors of each class. If we can define a pair of margin lines, a center line will automatically yield the *Optimal Hyperplane*.

There are two core functions of the SVM: train and predict.

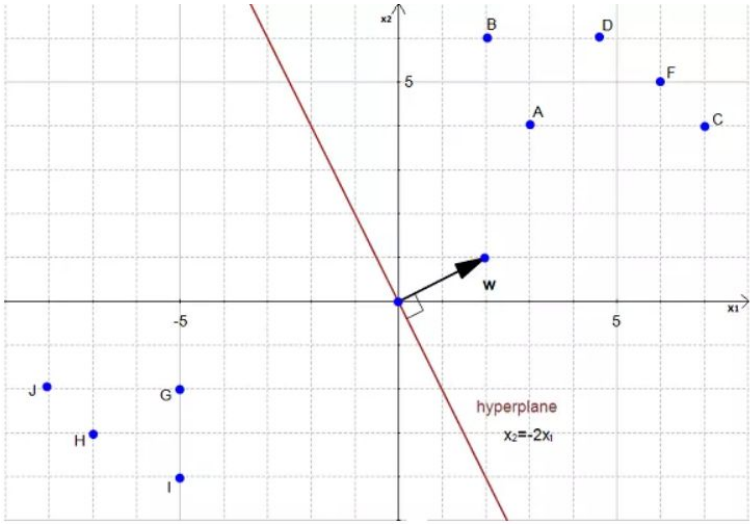
Finding the Optimal Hyperplane (Training)

During training, the *SVM* is populated with the data points yielded after pre-processing. Then an *Optimal Hyperplane* is calculated for the linearly separated data.

We know that in $2D$, a line is expressed as $y = mx + b$. What we need is an alternative notation that is easily scalable to dimensions greater than 2. We can replace scalars in the line equation with vectors to achieve this:

- $y = mx + b$ is equivalent to $y - mx - b = 0$. Moving everything to one side makes it possible to group scalars into vectors.
- Allocate vector $\boldsymbol{w}\,[-a, \quad 1]$ and vector $\boldsymbol{x}\,[x, \quad y]$. This plots the line in a parameter space, and is expandable to any dimension. Just as we can use x and y in $2D$ to find where a point is relative to the line, we can use vector ***x*** to find where a datapoint is relative to the *Optimal Hyperplane*.
- If we transpose ***w*** and take the dot product with ***x***, minus the *y-intercept*, we should get the original equation for the line:
  - $\boldsymbol{w}^T\boldsymbol{x} - b = -a \times x + 1 \times y = y - ax - b = 0$

Interestingly, when plotting plotting ***w*** in the same space as our data, the hyperplane itself is actually always orthogonal to ***w***. We can use this fact to use ***w*** as a geometric guide to define where the hyperplane should be relative to surrounding data points (the previously mentioned *Support Vectors*). For example:



[6]

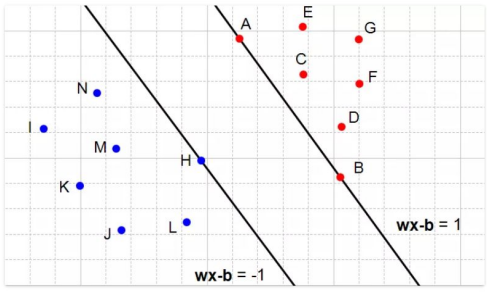
Figure 15: A *Hyperplane* and its orthogonal ***w*** vector.

In an *SVM*, data points are paired with class values. For any data point x_i, there is a corresponding value y_i that is either positive or negative to denote which class it belongs to. In our case, that would be *"car"* or *"non-car"*. Now we need to define some constraints for $\boldsymbol{w}^T\boldsymbol{x} - b$ such that it meets the criteria of the *Optimal Hyperplane*. Recall that we technically have three *Hyperplanes*, one on each extremity of each class and one in between that is the *Optimal Hyperplane*.

Let hyperplanes H_0, H_1, and H_2 exist such that:

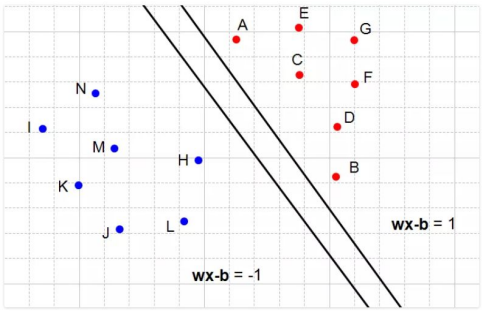
- H_0 is the *Optimal Hyperplane*: $H_0 = \boldsymbol{w}^T\boldsymbol{x} - b = 0$.
- H_1 is the *positive extremity Hyperplane*: $H_1 = \boldsymbol{w}^T\boldsymbol{x} - b = 1$. This one is by the reds in `Figure 14`.
- H_2 is the *negative extremity Hyperplane*: $H_2 = \boldsymbol{w}^T\boldsymbol{x} - b = -1$. This one is by the blues in `Figure 14`.

For every data point (vector) in class 1, x_i, we plug it into the corresponding *Hyperplane* H_1 and check that the constraint is satisfied. If any data point violates the constraint, then there are data points on the wrong side of that *Hyperplane*. The same is done for class -1. The correct *Hyperplane* is found by solving each inequality. This often leads to multiple solutions:



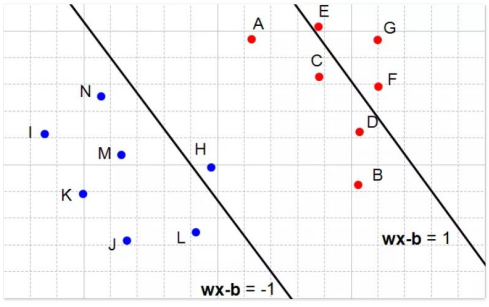
[7]

`Figure 16: `*`Hyperplanes`*` that satisfy the inequalities.`



[7]

`Figure 17: `*`Hyperplanes`*` that satisfy the inequalities.`



[7]

`Figure 18: `*`Hyperplanes`*` that do NOT satisfy the inequalities.`

The last step is to maximize the distance between H_1 and H_2, then use them to find H_0. This is a complex optimization problem that exceeds the scope of this document, but details are available in the references.

Classification

A foreign data point is plotted onto the same SVM after pre-processing the test input, and will fall on either side of the *Optimal Hyperplane*. The side it falls on is the predicted class.

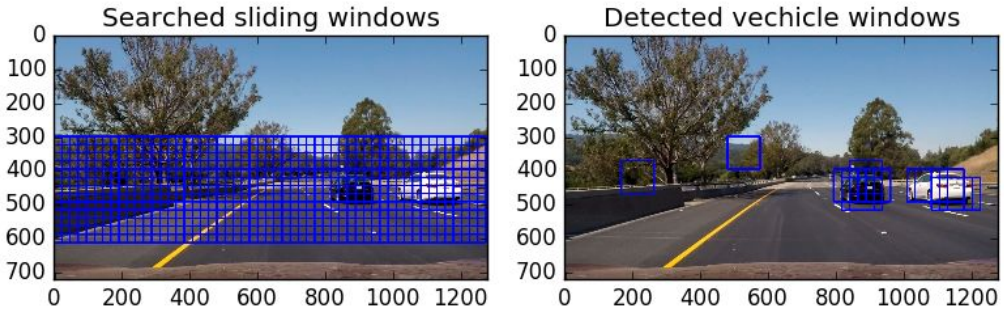
We use the same concepts from training by evaluating the inequality for each class using the new data point (vector) x_i. Using the *Optimal Hyperplane* equation $\boldsymbol{w}^T\boldsymbol{x} - b = 0$, the satisfied equality is the predicted class of x_i:

- If the inequality $\boldsymbol{w}^T x_i - b = 1$ is satisfied, x_i is of class 1.
- If the inequality $\boldsymbol{w}^T x_i - b = -1$ is satisfied, x_i is of class -1.

Obtaining Foreign Data

Sliding Window Search

Our training data is conveniently sized to standardized 64×64 labeled images. When it comes time to test, however, we will generally have any size image (typically from a dash or traffic cam) that includes multiple vehicles in a scene. To get bounding boxes around the vehicles, we need to convert our test image into a series of small images that can be imported into our *SVM* and classified as *"car"* or *"non-car"* using the same feature extraction techniques discussed here. In other words, we use *Sliding Window Search* to analyze and build a *HOG* for 64×64 patches of the image at a time. Patches that come back positive are drawn with a colored border overlaid on the original image.

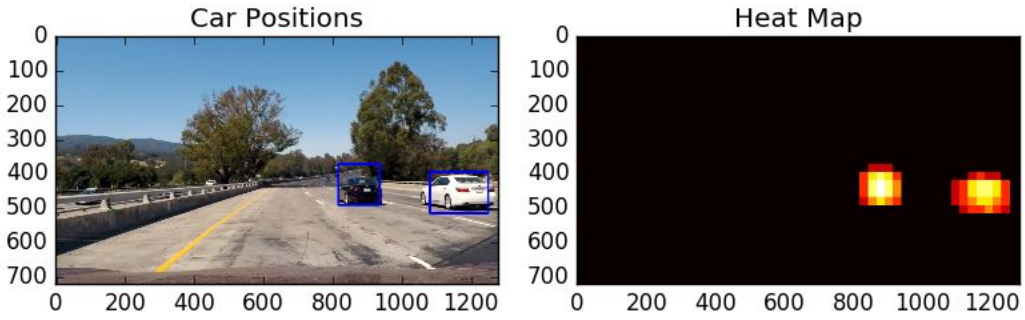


[1]

Figure 19: *Sliding Window Search and false positives.*

In the above example, the search is limited the lower/middle area based on the assumption cars will not be resent in the sky. As the window slides across the image in raster order (left to right/top to bottom), its treated like a training image and compared with the *Optimal Hyperplane* in the *SVM*. Note that the sliding windows overlap. That is, they slide by a delta smaller than the size of the window itself. This is similar to the way the 16×16 patches were handled in the *Normalization* step. We do this because many positive windows will cluster around actual cars in the image. Naturally there are some false positives, but they generally have at most a couple of windows clustered around them. We can use this to our advantage to suppress false positives.

Heatmap



[1]

Figure 20: Cleaning up the boxes.

For every cluster, the total number of overlapping boxes is recorded. Box counts that exceed some threshold (typically 4-5) are kept as positives and the rest are discarded. The cluster is replaced with a single average box.

Software Design

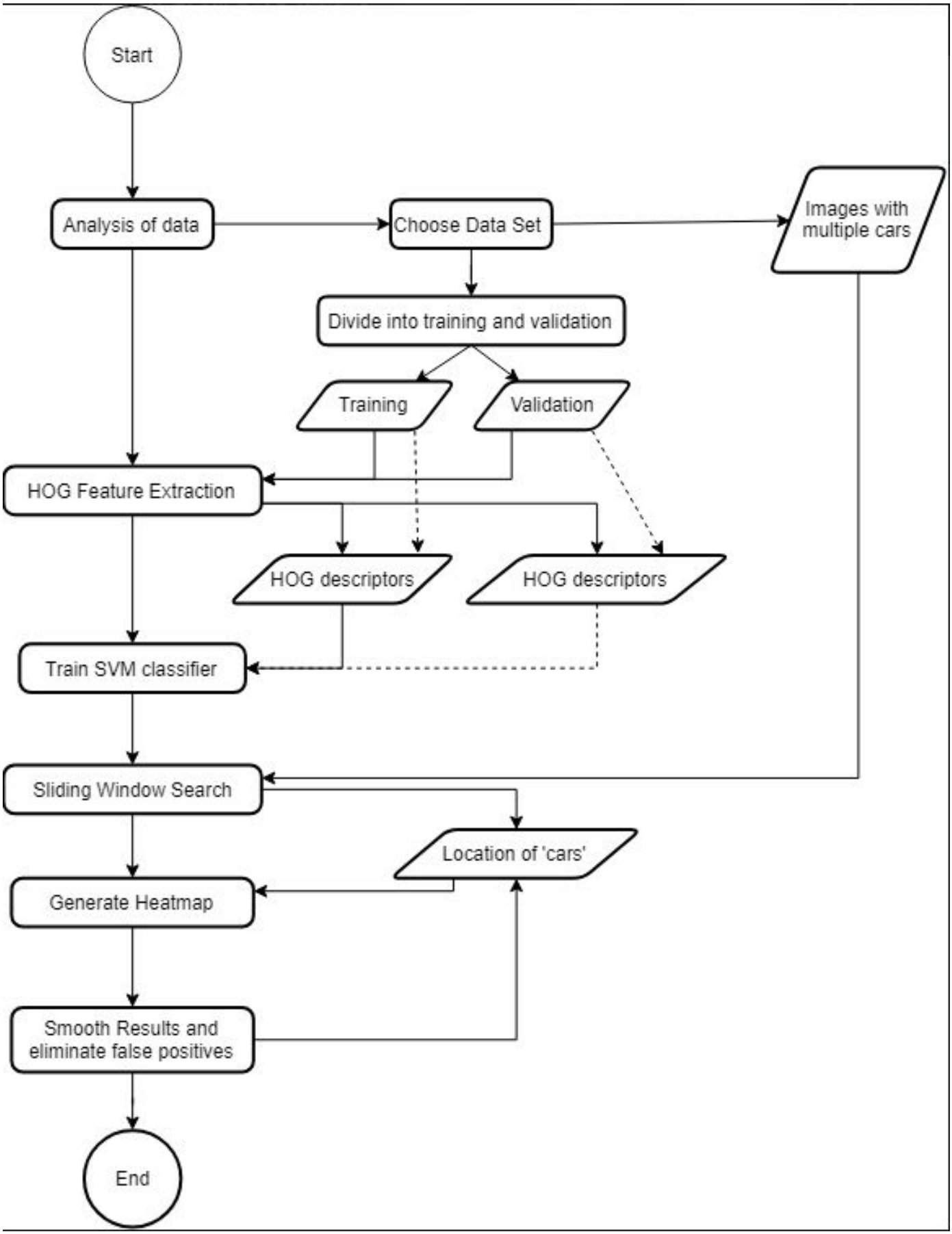


Figure 21: The flowchart.

Results

Figure 22: A fairly decent result.

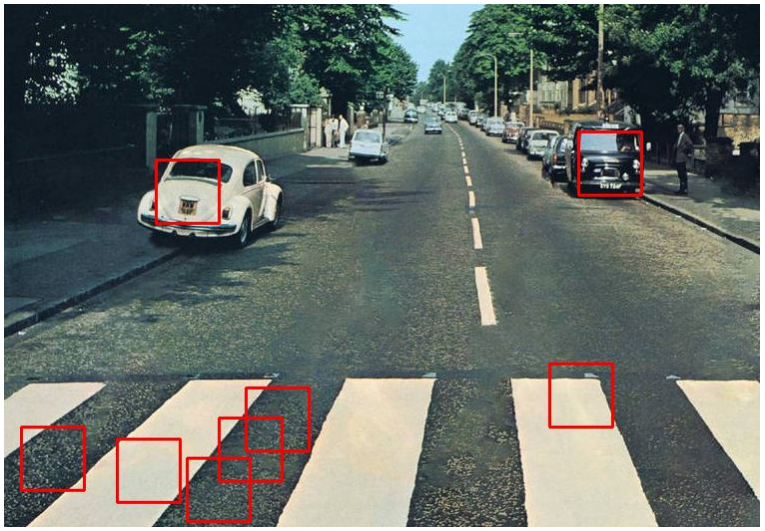


Figure 23: A less decent result.

```
+---------------------------------+
|     Adam Ali, Taylor Lippert    |
|         CSS 487 A Olson         |
|        December 5, 2018         |
|          VEHICLE DETECT         |
+---------------------------------+
| Cars     at dataset\car.......Y |
| Non-cars at dataset\non-car...Y |
| SVM      at dataset\svm.yml...Y |
| Test img at dataset\test.png..Y |
+---------------------------------+

SVM has already been trained!

Displaying results...
[ INFO:0] Initialize OpenCL runtime...
```

Figure 24: SVM already calculated.

```
+---------------------------------+
|     Adam Ali, Taylor Lippert    |
|         CSS 487 A Olson         |
|        December 5, 2018         |
|          VEHICLE DETECT         |
+---------------------------------+
| Cars     at dataset\car.......Y |
| Non-cars at dataset\non-car...Y |
| SVM      at dataset\svm.yml...N |
| Test img at dataset\test.png..N |
+---------------------------------+

SVM has not been trained. Training (this might take a while)...

LOAD CARS DATASET
READING IMAGES 8791/8792 DATASET LOADED.

LOAD NONCARS DATASET
READING IMAGES 8967/8968 DATASET LOADED.

PROCESS CARS DATASET
BUILDING HOG 8791/8792 HOGS COMPLETE.

PROCESS NONCARS DATASET
BUILDING HOG 8967/8968 HOGS COMPLETE.

BUILD SVM
Configuring SVM defaults... Done.
Adjusting data for hyperplane calculation (transpose)... Done.
Calculating support vectors... Done.
SVM saved to dataset\svm.yml

Displaying results...
[ INFO:0] Initialize OpenCL runtime...
```

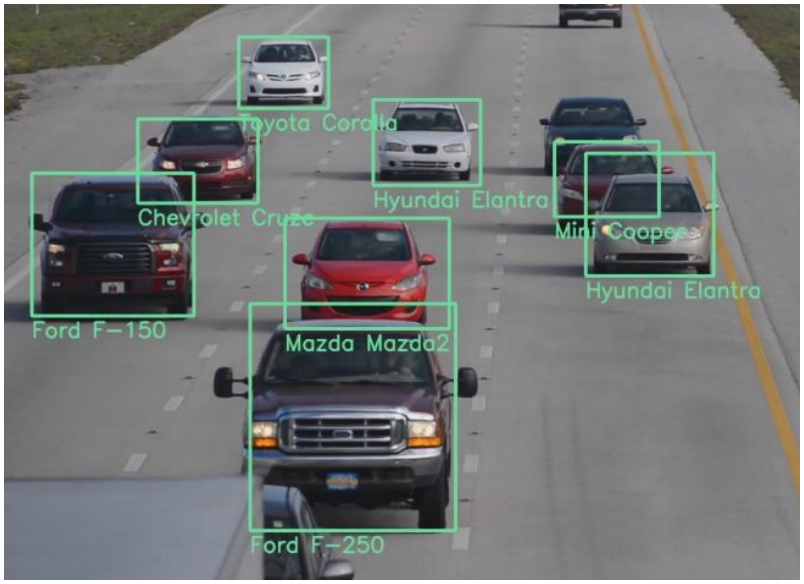
Figure 25: SVM not found.

For a first time attempt, we are reasonably happy with the results. In testing, we noticed that in lower contrast images results tend to suffer. Further, crosswalks are regulated identified as vehicles where there is a clear distinction between asphalt and a painted line. The corner action is likely fooling the SVM into thinking it is a vehicle roofline or some similar feature.

Going Forward

As is, this sort of technology is very practical for autonomous driving and traffic management applications. It is also probably useful for driver assistance features like forward collision warning. It is not unreasonable to think that we can look for even more information in the same sorts of data to introduce more use cases.

It would be interesting to get fine detail about each individual vehicle with regards to brand. It may be possible to classify individual features like lamps, grilles, bumpers and rooflines as part of one or another brand's design language. Something like this paired with color detection would prove useful for law enforcement when APBs are posted. A traffic cam at some location may detect a vehicle matching the APB description passing through and notify central command. If executed properly, this could even theoretically serve as a crude way of approximating a vehicle's position throughout the city.



[2]

Figure 21: An example of make and model recognition using advanced machine learning techniques paired with techniques similar to those showcased in this paper.

References

[0] Bratanič, Blaž, et al. “Real-Time Rotation Estimation Using Histograms of Oriented Gradients.” *PLOS ONE*, Public Library of Science, www.journals.plos.org/plosone/article?id=10.1371/journal.pone.0092137.

[1] Claudiu. “Vehicle Detection Using OpenCV and SVM Classifier.” *Cold Vision*, 23 Mar. 2017, www.coldvision.io/2017/03/23/vehicle-detection-using-opencv-svm-classifier/.

[2] Dehghan, et al. “View Independent Vehicle Make, Model and Color Recognition Using Convolutional Neural Network.” *Computer Vision Lab*, Sighthound Inc., 6 Feb. 2017, https://arxiv.org/pdf/1702.01721.pdf.

[3] Fu, Junsheng. “Vehicle Detection for Autonomous Driving.” *Github*, 28 Mar. 2018, junshengfu.github.io/vehicle-detection/.

[4] Gunzi, Arnaldo. “Vehicle Detection and Tracking Using Computer Vision.” *Chatbots Life*, 7 Mar. 2017, www.chatbotslife.com/vehicle-detection-and-tracking-using-computer-vision-baea4df65906.

[5] “Introduction to Support Vector Machines.” *OpenCV 2.4.13.7 Documentation*, 2 Dec. 2018, www.docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html.

[6] Kowalczyk, Alexandre. “SVM - Understanding the Math - Part 2.” *SVM Tutorial*, 14 Nov. 2018, www.svm-tutorial.com/2014/11/svm-understanding-math-part-2/.

[7] Kowalczyk, Alexandre. “SVM - Understanding the Math - Part 3.” *SVM Tutorial*, 14 Nov. 2018, www.svm-tutorial.com/2014/11/svm-understanding-math-part-3/.

[8] Mallick, Satya. “Histogram of Oriented Gradients.” Learn OpenCV, 6 Dec. 2016, www.learnopencv.com/histogram-of-oriented-gradients/.

[9] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, San Diego, CA, USA, 2005, pp. 886-893 vol. 1.