We want to locate and track Vehicles in images of a video.

We will learn what features are the most useful for a supervised learning classifier that can help us detect whether something is a vehicle (or any object) or not.

For example: some characteristics in an image that might be useful to detect cars in an image are Color, Position within the image, shape, and Apparent Size and more!

We will learn about object Detection, and then learn how to classify them.

**Detection:**

Color and Gradient are good differentiators between objects. They are both features that we can use to help classify our image. In all applications, we will use a combination of features that lead to the best result.

So what features align with what characteristics in the objects we want to detect?

A Default images pixel values can give details on the color and shape of an object, while the histogram of pixel values gives us info on only the color. At the same time the gradients of pixel values gives us information about an objects shape.

**Features:**

Simplest Feature you can get from an image is the color. You can compare a known car image with the test region you want to check for that object. You can do this by subtracting pixel values or getting to correlation and threshold the values. This is called **Template Matching**.

The problem with template matching is that you need a Template. If you are thinking about tracking a vehicle from a video, its position relative to the camera changing as it speeds up and slows down would not match the template, meaning it won’t work well for us. (Same Orientation, size, and color)

We need to find a transformation that are robust to changes in appearance. An example of one transform is the **histogram of color values** in an image. It will match things in a region that share the same color distribution. By normalizing the distribution, we can also account for a variation in the objects size. It will also match the object if it is in slightly different aspects orientations can still be matched. Note that this is only matching based on color, which can still match unwanted regions.

Now let’s think about other color spaces. If we thing about HSV images. In some images, cars are usually saturated in color while the background is pale. Maybe Saturation values can help identify cars invariant of colors. Of course this isn’t always the case, but it’s good to try as much as you can.

You should get used to plotting your test images in different color spaces to try and determine which color spaces can help make vehicle detection color invariant.

There are many large datasets available of images used throughout the process of making autonomous vehicles. An example of one is <http://www.cvlibs.net/datasets/kitti/>

We can use some of the frames of video they have collected to see if we can locate clusters of colors that correspond to specific cars, trees, the sky, etc. This way we can get a better idea of what sections we can use to identify vehicles over the environment. This doesn’t need to be perfect but it will be one of many useful features that will be used to help us identify vehicles.

It may be confusing to look at the 3D plots of full images and try and gain information about the vehicles versus the environment, so you should also try to compare the 3D plots of vehicles versus the environment.

**Spatial Binning of Color**

Raw pixel values are quite useful to include in your feature vector in searching for cars.

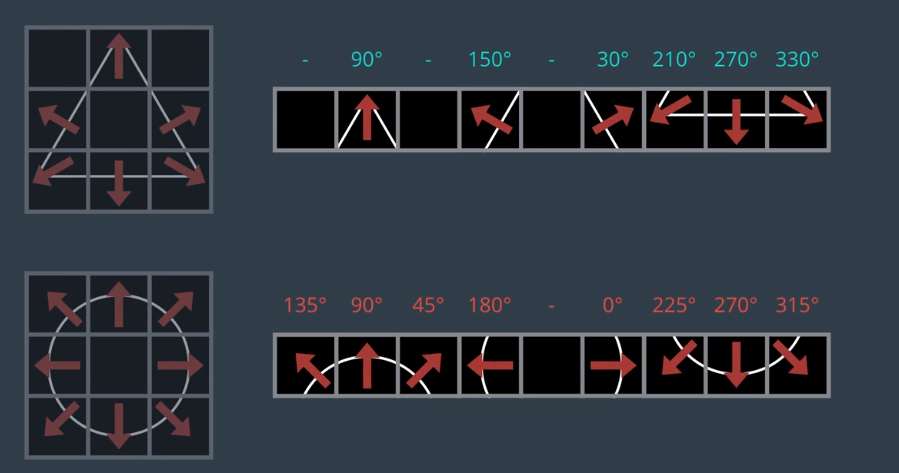


If you want to include raw pixel values, it can be cumbersome to include all 3 color channels of a full resolution image, so we can perform **spatial binning** on an image and still retain enough information to help in finding vehicles.

Spatial binning means to store information/pixel values into a combined bin in a way to organize it. Down sampling an image is an example of spatial binning. We can see from the image above that even going all the way down to 32x32 pixel resolution, the car is still clearly identifiable by eye, meaning relevant features are still preserved at this resolution.

**Gradient Features:**

When we want to gain information on a class object that can come in a variety of colors, structural queues like gradients and edges can give us a more robust representation. Actually the presence of gradients in specific direction around the center can give us information about its shape.



If we take a shapes gradient image, and separate it into different grid cells, and treat them as a flat 1D array, we can obtain a signature for that image. Different signatures correlate with different shapes. The signature for an image should have enough flexibility to account for small variations in orientation, size, etc. It should also be distinct so we can distinguish between shapes.

The above image of the shapes shows that the looking at the presence of gradients in specific directions around the center and lead to giving us a signature for that specific shape.

Using gradient values directly can lead to making the signature too sensitive.

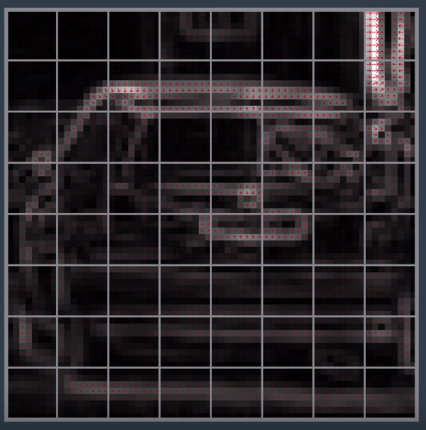
So to generalize the gradient and make the signature for each shape less sensitive, we use a technique called the **Histogram of Oriented Gradients (HOG).**

**Histogram of Oriented Gradients:**

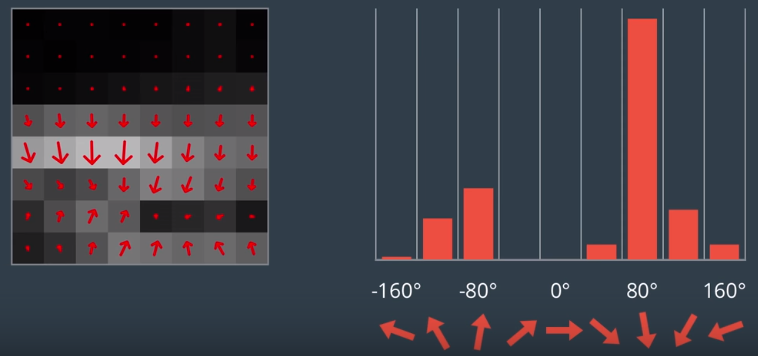
Getting the HOG of an image means to take the gradient of an image so that you can have the magnitude and direction (orientation) of the gradient. Separate your image into small cells. For each cell, you take its pixels gradients orientation and magnitude, and create a histogram of orientations for the whole cell. So in essence you are taking a histogram of the gradients orientations (Hence the name).

This gives us a histogram, meaning binned data, by the orientation like the image above where the magnitudes are summed.

Example:

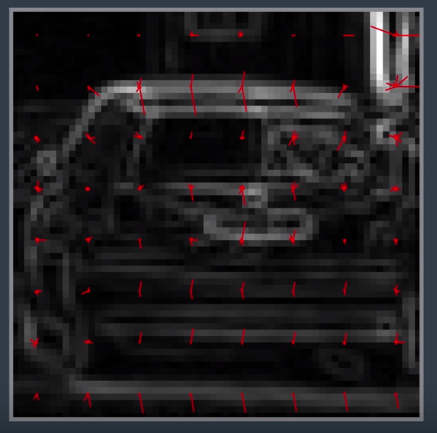
If we look at a 64x64 image of a car (1st image) and take its gradients magnitude and direction at each pixel, and then group them up into small cells of 8x8 each (2nd image). Then looking at each cell, we compute a histogram of the gradient directions (orientations) with each of the 64 pixels inside the 8x8 cell. The gradients samples at this small scale can be summed up into say 9 bins.



This histogram is not the sum of the number of pixels that fit into that orientation, but the sum of the magnitudes of those pixels.

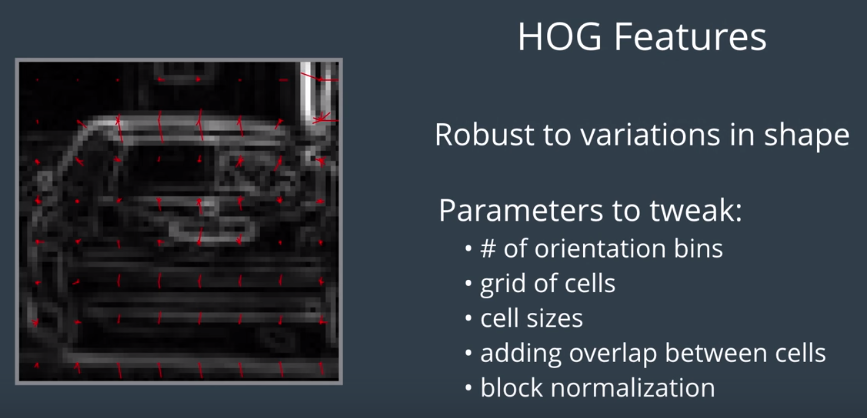


Each cell can be thought of an addition of the contributions from samples from each bin of the histogram to get a star of different lengths. The max length is the dominant gradient direction in this cell. Because the histogram is a sum of the magnitude values, the stronger gradient magnitudes contribute more weight to their orientation bin, so the effect of small random gradients due to noise is reduced.

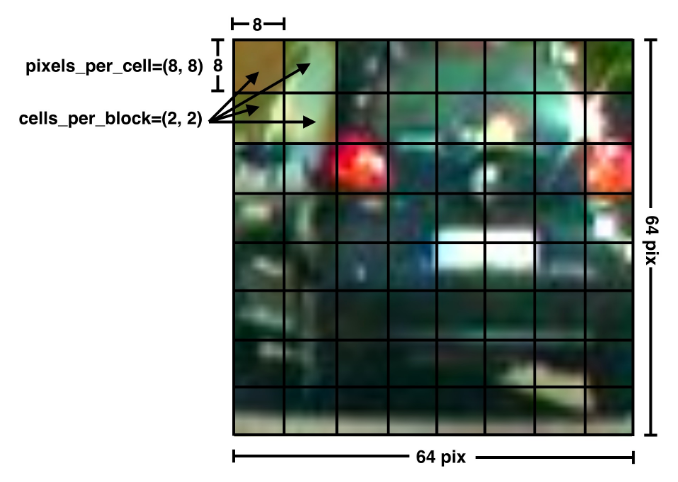


When you do this for all the cells, you can see a representation of the original structure emerge, which can become a key signature for a given shape.

This representation of the shape is called the Histogram of Orientated Gradients (HOG). It is then flattened into a vector and is used as a feature descriptor.



These parameters can be tweaked to change how accommodating/sensitive the feature is. In practice the intensity in a small block of cells is normalized to give a more unique but robust signature.



The scikit-image package has a build in function that extracts the Histogram of Oriented Gradients feature vector. The function can take the parameters shown in the image above, as well as another optional power law or “gamma” normalization scheme (set by the flag transform\_sqrt) that may help reduce the effects of shadows or other illumination variation, but will cause errors if your image has negative values.

So if you think about the number of elements in your HOG features vector for the whole image, instinct is to say that each cell has 9 bins, and there are 8x8 cells, so then the HOG features vector is 9x8x8= 576.

That is not true with block normalization, which calculates at each block, and the step across and down.

Meaning that in this image you can look and see that the blocks would be 4x4, but with normalization, the number of blocks becomes 7x7 due to the addition of one step down per block and one step across.  
Therefore, the total size of the HOG features vector is 7x7x2x2x9 = 1764. (2x2 cells per block)

This will serve as one of the features that you will use to train your SVM Classifier. This means that there will be a database used with images labeled as vehicles, and non-vehicles.

Images used from the project are comprised of images taken from: <http://www.gti.ssr.upm.es/data/Vehicle_database.html>

<http://www.cvlibs.net/datasets/kitti/>

**Combining Features:**

We can combine color based and shape based features due to the fact that they complement each other in detailing a given object in an image. Having more features to recognize a given object can make our detector more robust. However we need to be careful how we use it.

We would normally combine feature vectors from each feature into one combined vector, however due to the fact that they can represent different quantities, the magnitude of some values can dominate the combined vector over others. A good idea is to try and normalize the magnitude of the combined feature vector. It helps to ensure that one features data doesn’t overwhelm the others.

Sklearns package has a method called StandardScaler() that can normalize our vector.

There may also might be more elements of one feature than the other in the combined feature vector. If the quantity is very large it may or may not be an issue. This could be solved by trying to find and remove redundancies in the combined feature vector. This can be done by using a decision tree to try and analyze the relative importance of features and then drop the ones that aren’t contributing much.

Now we can use this combined feature vector to build a classifier.

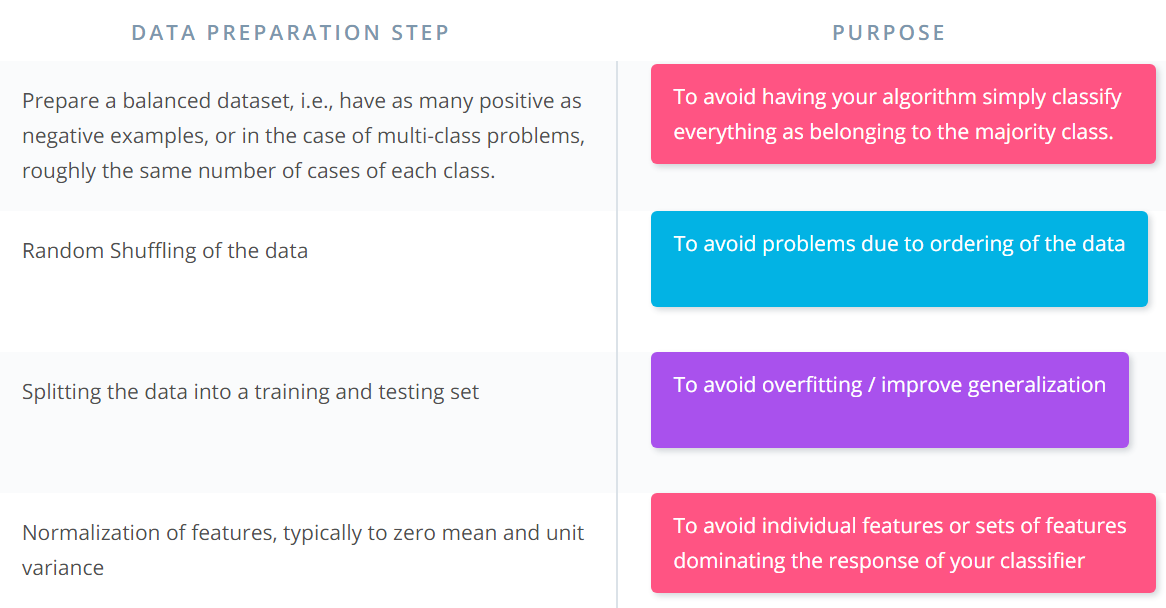
**Building a Classifier:**

Now that we have features, we can design a classifier that can differentiate between a car image, and a non-car image. Then we can run this classifier over an entire frame sampling small patches along the way. The patches that are classified as cars, are the desired detections.

So again, we train our classifier using labeled data of car and non-car images. So our dataset includes labeled data of car images and non-car images.

If we only had a full frame available of a video, we would need to crop out specific car and not car regions and scale them all to a fixed size. Ideally, our dataset would be balanced meaning that the number of car and non-car images are equal, otherwise we run the risk of having class imbalance. We can mitigate some of these issues if they arise by duplicating images in the smaller set to make up for the difference. Or if we do not have enough non-car images, we can always go back to the full frames and just extract more images.

Now we need to split our data into Training Set and Test Set. We need to shuffle the data randomly before splitting the sets. We will using training set to train our classifier, and test its accuracy with the test set. Therefore the initial dataset we start with should ideally be balanced.



Training: Your training set individually has its features extracted, and then those features are matched with those images labels. The error between the models prediction and the correct label is used to adjust the model for the next test image.

Testing: After training is performed with the whole set, previously unseen images to the model in the test set are used to determine the models accuracy.

Typically the error in the test set is larger than the training set, and both errors decrease the more you train your set.

If we keep training beyond a certain point with the training set, the testing sets error will increase. This is due to the fact that your model is over-fitting to the training data, and is becoming less generalized to new unseen images.

The choice of what classifier to use can take experimentation to find out what one works best for a specific problem. We will start with what is said to work well with HOG features, which is Support Vector Machines (SVM), but we can use Decision Trees, a Neural Network, or even a combination of multiple classifiers.

**Parameter Tuning:**

Because we are starting with SVM, there are 2 parameters that we can tune. Those are the Gamma and the C parameters. We also have a choice of the kernel we want to use to find a decision boundary that will minimize our prediction error. (If we choose a linear kernel, we can only tune C, not Gamma)

We can also use one of scikit-learns parameter tuning algorithms to tune them for us.

There are 2 algorithms that can do this called:

GridSearchCV: Exhaustively works through multiple parameter combinations, cross-validating as it goes.

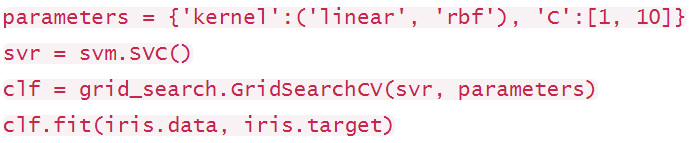
If I input C as [0.1, 1, 10] and Gamma as the same, it would go through every possible combination.

RandomizedSearchCV: This works the same as the above but takes a random sample of parameter combinations. This is faster than GridSearchCV, due to only using a subset of the possible parameter combinations.

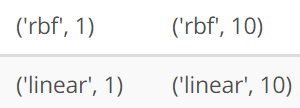
Cross-validation with GridSearchCV:

3-fold cross validation is used to determine the best parameter set, meaning that the training set is taken and split into 3 equal partitions. The algorithm trains with 2 partitions and uses the third to validate. Then once that is completed, it chooses another as the validation set and trains on the other 2 and so on. The accuracy is averaged over each partition, and then the parameter combination that led to the best accuracy score is chosen.

Example:



Chooses all possible parameter combinations.



We can access what parameters are the most optimal from the classifier using clf.best\_params\_

**Sliding Windows:**

Now that we have extracted features and trained a Classifier, we can start thinking about how to apply our classifier that can decide between car images and non-car images on a full image. What we need to do is run our classifier on select windows in a full image and determine whether that window contains a car or not.

We will implement a sliding window technique that steps across an image in a grid pattern and runs our classifier on each window. If the window contains a car, we will keep those window positions and draw a box.

There are a couple things that need to be decided.

What size window do we want to search with? Where in the image do you want to start and stop the search? How much do we want each window to overlap?

If the whole image is searched:

Ex. If we have a 1280x960 image with 64x64 windows and 50% overlap. We will have (1280-64\*0.5)/(64\*(1-0.5)) = 39 vertical and (960-64\*0.5)/(64\*(1-0.5)) = 29 horizontal. Totalling 39\*29 = 1131 windows are searched in total.

In general, we don’t know the size of the objects in our image, we should always have multiple-scales of windows. So we should establish a maximum and minimum scale that we want to search and then search those and some intervals in between. It is important to understand that the number of windows we are searching can increase rapidly, so doing things like only searching the bottom portion of our image where cars will actually appear can save a lot of processing time.

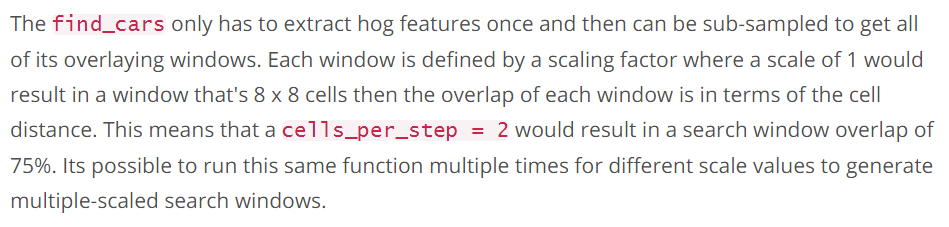
We also know that cars that should be detected by smaller windows are closer to the horizon of our image, so we can limit the region we search for vehicles with a window of that scale, and vice versa with large scale windows.

**More Efficient Sliding Window Approach:**

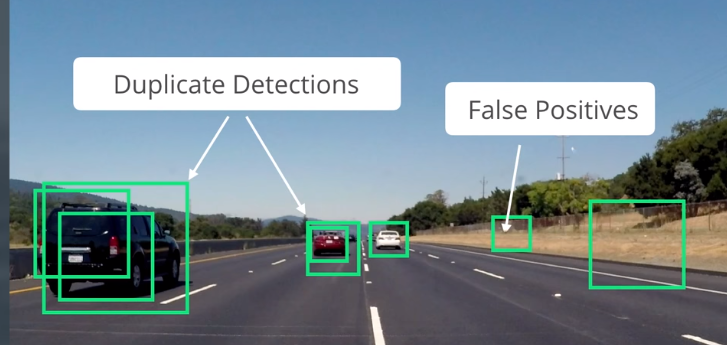
**HOG Sub-Sampling Window Search**

This method lets us only have to extract the HOG Features once. It combines the ability to extract features and make predictions. It extracts HOG features once and then sub-sampled to get all of its overlaying windows.

Each window can be defined by a scaling factor where a scale of 1 would make your search windows the same size as the HOG cell (8x8).



**Filtering Results:**



Once we complete our sliding window search, we can have a couple things happen. We can have **false positives**, meaning our classifier decided a certain window we searched should be classified as a car. We can also have multiple search windows recognizing the same vehicle called **Duplicate Detections**.

Correctly eliminating/filtering all the false positives while combining all the search windows from duplicate detections is important in determining where vehicles are and equally importantly not in an image.

When relying on accurate vehicle detection, things like false positives can lead to an automated vehicle to respond with emergency breaking for example.

Having out tight bounding box for one vehicle will help us determine the size and position of each vehicle.

The size and position of other vehicles will be used in path planning and control algorithms to steer us clear of other vehicles.

**Solution:**

Duplicate Detections can handled by finding the centroid of each of the windows that detect the same vehicle and drawing one box with that centroid.

False positives are solved through a video pipeline. They can be filtered by determining which windows are detected in one frame and not the next.

With high confidence detections we can accurately calculate the size and the position of each vehicle in a video from frame to frame. We can also track how that vehicles centroid is moving from frame to frame, and use this data to estimate the position of each vehicle in subsequent frames.