Computer Vision with OpenCV 3 and Python

Dr Kien Nguyen
Email: k.nguyenthanh@qut.edu.au
Queensland University of Technology, Australia

Warning

Installation

- Ubuntu 18 + Python 2.7/3.6.5 + OpenCV 3/4
  - https://www.begueradj.com/how-to-install-opencv4.0-for-python3.6.5-ubuntu18.04.html

- MacOS + Python 2.7 + OpenCV 4

Outline

- Part 1: Face/Human Detection using Haar Cascades
- Part 2: Object Classification
- Part 3: Object Detection
- Part 4: Object Tracking

Further reading

- Online course:

https://www.udemy.com/master-computer-vision-with-opencv-in-python/

- Book:

Learning OpenCV3 for Computer Vision with Python

Part 1. Face/human Detection using Haar Cascades

- Back to 2001, when face detection was more of a research topic and many methods were proposed which didn't work very well. Paul Viola and Michael Jones came up with their seminal paper which not only detected faces *robustly*, but did so in *real-time*. It is one of the most cited papers in Computer Vision.

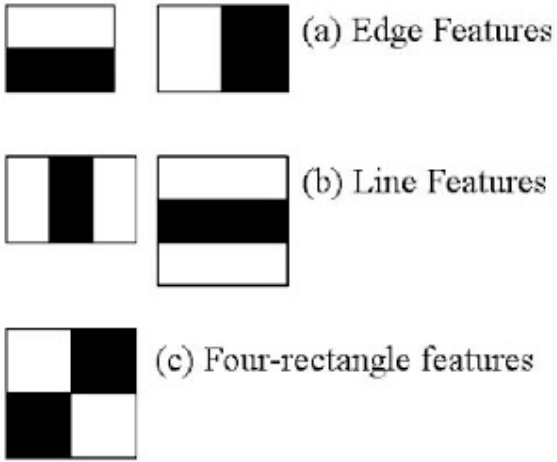
- In Part 1, we will learn how to use haar cascade based face, smile and human detectors in OpenCV.

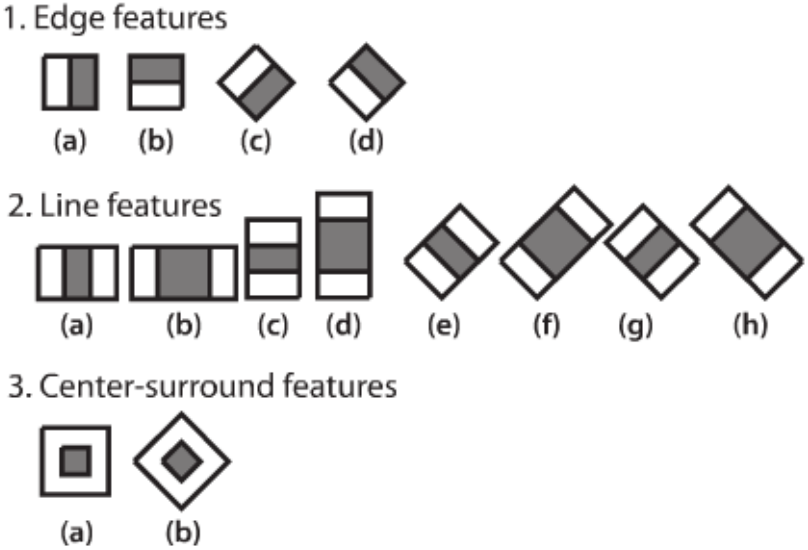
Theory of face detection classifiers

- A computer program that decides whether an image is a positive image (face image) or negative image (non-face image) is called a **classifier**. A classifier is trained on hundreds of thousands of face and non-face images to learn how to classify a new image correctly.
- OpenCV provides us with two pre-trained (located under *opencv/data/*) and ready to be used for face detection classifiers:
    - Haar Classifier
    - LBP Classifier

Haar Classifier

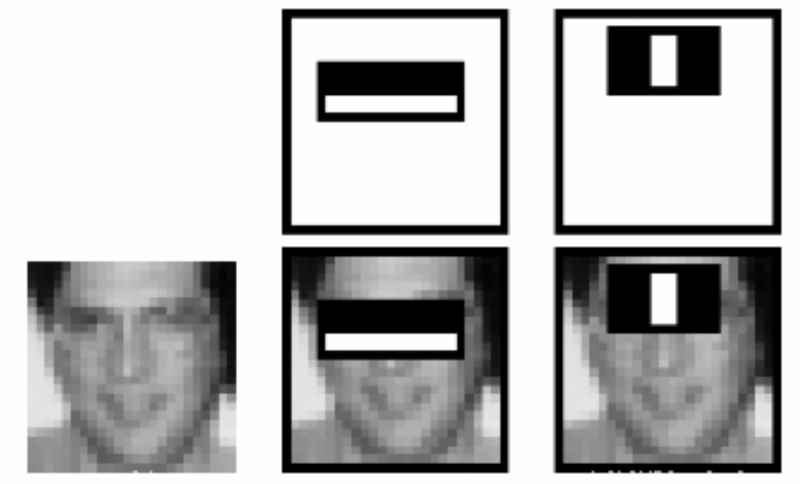
- It starts by extracting Haar features from each image as shown by the windows below





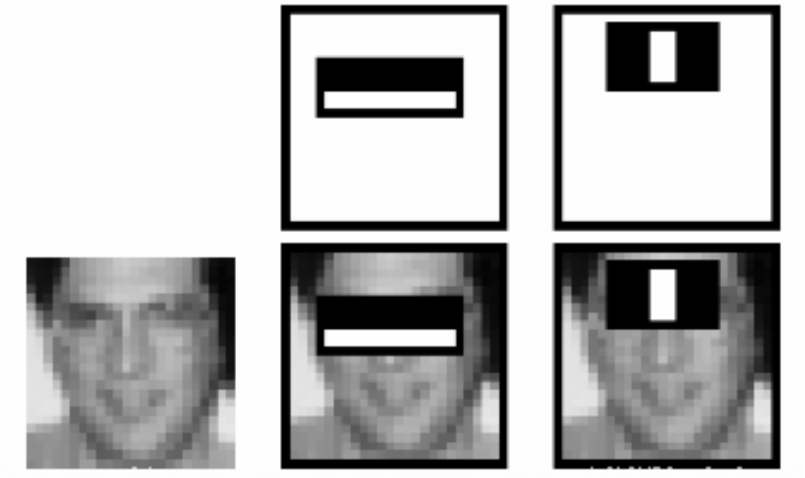
Haar Classifier

- Each kernel is applied to the image find the sum of the pixels under white and black rectangles to calculate a single feature
- All possible sizes and locations of each kernel are used to calculate lots of features (Even a 24x24 window results over 160000 features).



Haar Classifier

- The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks.
- The second feature selected relies on the property that the eyes are darker than the bridge of the nose.
- But the same windows applied to cheeks or any other place is irrelevant



Haar Classifier

- So how do we select the best features out of lots of features? It is achieved by a technique called **Adaboost**.
    - AdaBoost is a training process for face detection, which selects only those features known to improve the classification (face/non-face) accuracy of our classifier
- Apply each and every feature on all the training images.
    - For each feature, it finds the best threshold which will classify the faces to positive and negative. Obviously, there will be errors or misclassifications. We select the features with minimum error rate, which means they are the features that most accurately classify the face and non-face images.
    - The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others forms a strong classifier. The paper says even 200 features provide detection with 95% accuracy. Their final setup had around 6000 features.

Haar Classifier

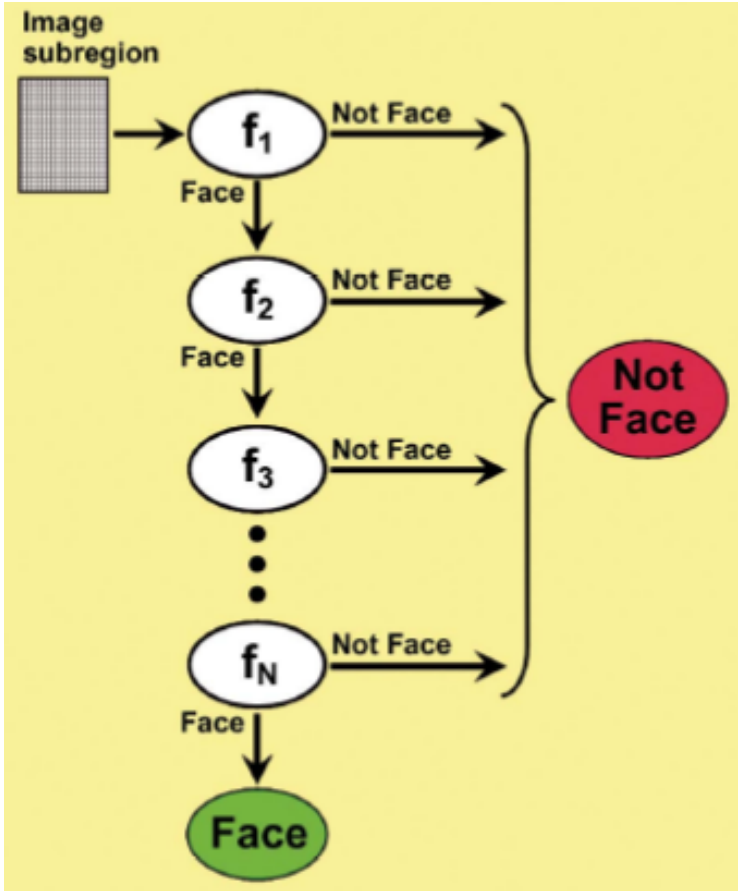
- So now you take an image. Take each 24x24 window. Apply 6000 features to it. Check if it is face or not. Wow.. Isn't it a little inefficient and time consuming? Yes, it is. The authors have a good solution for that.

- In an image, most of the image is non-face region. So it is a better idea to have a simple method to check if a window is not a face region. If it is not, discard it in a single shot, and don't process it again. Instead, focus on regions where there can be a face. This way, we spend more time checking possible face regions.

- The authors' detector had 6000+ features with 38 stages with 1, 10, 25, 25 and 50 features in the first five stages.

Haar Classifier

- Cascade of Classifiers
  - Instead of applying all 6000 features on a window, the features are grouped into different stages of classifiers and applied one-by-one. (Normally the first few stages will contain very many fewer features).
  - If a window fails the first stage, discard it. We don't consider the remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region.
- Training your own cascade of classifiers:
  - https://docs.opencv.org/3.4.1/dc/d88/tutorial_traincascade.html



OpenCV's pre-trained haar cascades of classifiers

- ~/OpenCV/opencv/data/haarcascades$ ls
- haarcascade_eye_tree_eyeglasses.xml haarcascade_mcs_leftear.xml
  haarcascade_eye.xml haarcascade_mcs_lefteye.xml
  haarcascade_frontalface_alt2.xml haarcascade_mcs_mouth.xml
  haarcascade_frontalface_alt_tree.xml haarcascade_mcs_nose.xml
  haarcascade_frontalface_alt.xml haarcascade_mcs_rightear.xml
  haarcascade_frontalface_default.xml haarcascade_mcs_righteye.xml
  haarcascade_fullbody.xml haarcascade_mcs_upperbody.xml
  haarcascade_lefteye_2splits.xml haarcascade_profileface.xml
  haarcascade_lowerbody.xml haarcascade_righteye_2splits.xml
  haarcascade_mcs_eyepair_big.xml haarcascade_smile.xml
  haarcascade_mcs_eyepair_small.xml haarcascade_upperbody.xml

Haar-cascade Detection in OpenCV

- obj_cascade = cv2.CascadeClassifier ('haarcascade_name.xml')
- objs = obj_cascade.detectMultiScale(image[, scaleFactor[, minNeighbors[, flags[, minSize[, maxSize]]]]])
  - **image** is the input grayscale image.
  - **objects** is the rectangular region enclosing the objects detected = {(x,y,w,h)}
  - **scaleFactor** is the parameter specifying how much the image size is reduced at each image scale. It is used to create the scale pyramid.
  - **minNeighbors** is a parameter specifying how many neighbors each candidate rectangle should have, to retain it. Higher number gives lower false positives.
  - **flags** : used for an old cascade as in the function cvHaarDetectObjects. It is not used for a new cascade.
  - **minSize** : Minimum possible object size. Objects smaller than that are ignored.
  - **maxSize** : Maximum possible object size. Objects larger than that are ignored

Demo: Detect Faces from an image

```
import cv2
import sys
import numpy as np

if __name__ == '__main__':

  faceCascade = cv2.CascadeClassifier('models/haarcascade_frontalface_default.xml')
  faceNeighborsMax = 10
  neighborStep = 1

  frame = cv2.imread("hillary_clinton.jpg")
  frame = cv2.imread("people.jpg")

  frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  for neigh in range(1, faceNeighborsMax, neighborStep):
    faces = faceCascade.detectMultiScale(frameGray, 1.2, neigh)
    frameClone = np.copy(frame)

    for (x, y, w, h) in faces:
      cv2.rectangle(frameClone, (x, y), (x + w, y + h), (255, 0, 0), 2)

    cv2.putText(frameClone, "# Neighbors = {}".format(neigh), (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 4)
    cv2.imshow('Face Detection Demo', frameClone)
    if cv2.waitKey(500) & 0xFF == 27:
      cv2.destroyAllWindows()
      sys.exit()
```

Demo: Detect Smile Faces from an image

```
import cv2
import sys
import numpy as np


if __name__ == '__main__':

  faceCascade = cv2.CascadeClassifier('models/haarcascade_frontalface_default.xml')
  smileCascade = cv2.CascadeClassifier('models/haarcascade_smile.xml')
  smileNeighborsMax = 100
  neighborStep = 2

  frame = cv2.imread("hillary_clinton.jpg")
  #frame = cv2.imread("people.jpg")

  frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  faces = faceCascade.detectMultiScale(frameGray, 1.2, 3)
  for (x, y, w, h) in faces:

    cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
    faceRoiGray = frameGray[y: y + h, x: x + w]
    faceRoiOriginal = frame[y: y + h, x: x + w]

    for neigh in range(1, smileNeighborsMax, neighborStep):
      smile = smileCascade.detectMultiScale(faceRoiGray, 1.5, neigh)

      frameClone = np.copy(frame)
      faceRoiClone = frameClone[y: y + h, x: x + w]
      for (xx, yy, ww, hh) in smile:
        cv2.rectangle(faceRoiClone, (xx, yy), (xx + ww, yy + hh), (0, 255, 0), 2)

      cv2.putText(frameClone, "# Neighbors = {}".format(neigh), (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 4)
      cv2.imshow('Face and Smile Demo', frameClone)
      if cv2.waitKey(500) & 0xFF == 27:
        cv2.destroyAllWindows()
        sys.exit()
```

Demo: Detect Faces from a webcam

```python
import cv2
import sys
import numpy as np

if __name__ == '__main__':

  camera = cv2.VideoCapture(0)

  faceCascade = cv2.CascadeClassifier('models/haarcascade_frontalface_alt.xml')

  while (True):
    ret, frame = camera.read()

    frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)


    faces = faceCascade.detectMultiScale(frameGray, 1.1, 10)
    frameClone = np.copy(frame)

    for (x, y, w, h) in faces:
      cv2.rectangle(frameClone, (x, y), (x + w, y + h), (255, 0, 0), 2)

    cv2.imshow('Face Detection Demo', frameClone)
    if cv2.waitKey(500) & 0xFF == 27:
      cv2.destroyAllWindows()
      sys.exit()
```

Demo: Detect Humans/People from a video Demo

```
import cv2
import sys
import numpy as np

if __name__ == '__main__':

  camera = cv2.VideoCapture('768x576.avi')

  faceCascade = cv2.CascadeClassifier('models/haarcascade_fullbody.xml')

  while (True):
    ret, frame = camera.read()

    frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = faceCascade.detectMultiScale(frameGray, 1.1, 5)
    frameClone = np.copy(frame)

    for (x, y, w, h) in faces:
      cv2.rectangle(frameClone, (x, y), (x + w, y + h), (255, 0, 0), 2)

    cv2.imshow('Human Detection Demo', frameClone)
    if cv2.waitKey(500) & 0xFF == 27:
      cv2.destroyAllWindows()
      sys.exit()
```

Detect Faces using LBP cascade

Algorithm	Advantages	Disadvantages
Haar	1. High detection accuracy 2. Low false positive rate	1. Computationally complex and slow 2. Longer training time 3. Less accurate on black faces 4. Limitations in difficult lightening conditions 5. Less robust to occlusion
LBP	1. Computationally simple and fast 2. Shorter training time 3. Robust to local illumination changes 4. Robust to occlusion	1. Less accurate 2. High false positive rate

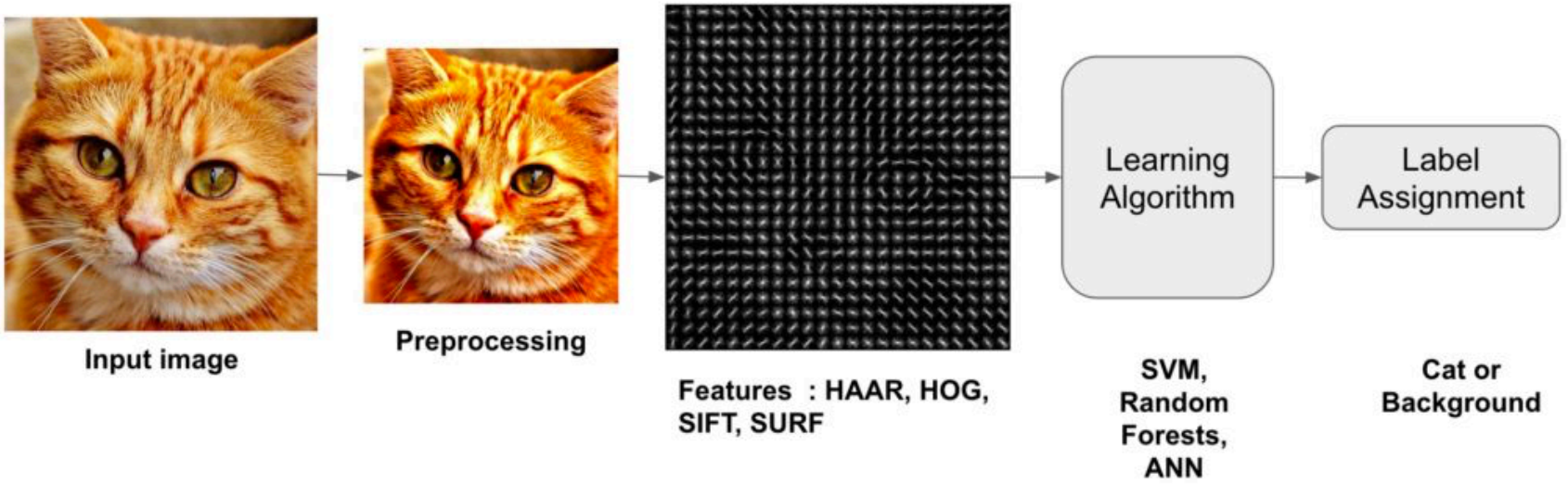
Demo: Detect Faces using other cascades: LBP & HOG

```python
import cv2
import sys
import numpy as np

if __name__ == '__main__':

#  faceCascade = cv2.CascadeClassifier('models/haarcascade_frontalface_default.xml')
  faceCascade = cv2.CascadeClassifier('models/lbpcascade_frontalface.xml')

# HOG no longer supported
#  faceCascade = cv2.CascadeClassifier('models/hogcascade_pedestrians.xml')

  faceNeighborsMax = 10
  neighborStep = 1

  frame = cv2.imread("test3.jpg")


  frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  for neigh in range(1, faceNeighborsMax, neighborStep):
    faces = faceCascade.detectMultiScale(frameGray, 1.1, neigh)
    frameClone = np.copy(frame)

    for (x, y, w, h) in faces:
      cv2.rectangle(frameClone, (x, y), (x + w, y + h), (255, 0, 0), 2)

    cv2.putText(frameClone, "# Neighbors = {}".format(neigh), (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 4)
    cv2.imshow('Face Detection Demo', frameClone)
    if cv2.waitKey(500) & 0xFF == 27:
      cv2.destroyAllWindows()
      sys.exit()
```

Homework

- 1. Write a program to detect cat face detection
- 2. Write a program to count the number of humans in an image

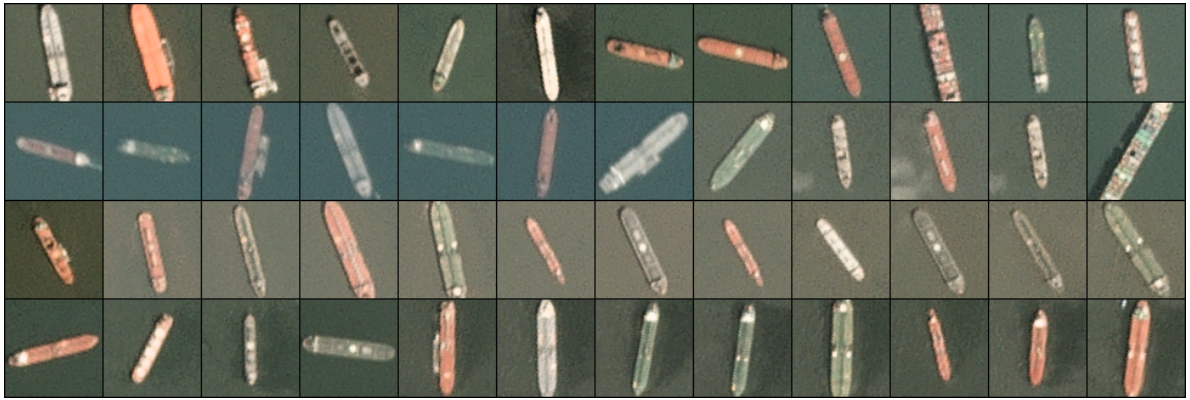
Part 2: Object Classification



2.1. Ship classification

- Dataset: 3600 80x80 RGB images labeled with either a "ship" or "no-ship" classification. Image chips derived from PlanetScope full-frame visual scene products

- The "ship" class includes 900 images.
    - Images in this class are near-centered on the body of a single ship. Ships of different sizes, orientations, and atmospheric collection conditions are included.



2.1. Ship classification

- The "no-ship" class includes 2700 images.
  - A third of these are a random sampling of different landcover features - water, vegetion, bare earth, buildings, etc. - that do not include any portion of an ship.
  - Another third are "partial ships" that contain only a portion of an ship, but not enough to meet the full definition of the "ship" class.
  - Last third are images that have previously been mislabeled by machine learning models, typically caused by bright pixels or strong linear features.

Additional packages

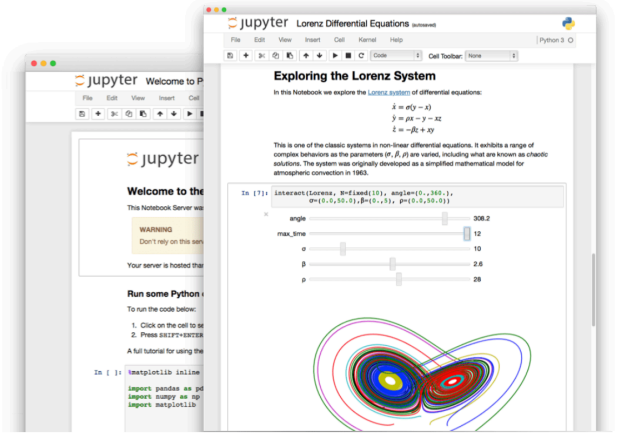
scikit-image
image processing in python

scikit-learn
Machine Learning in Python

Jupyter Notebook

jupyter

Install About Us Community Documentation NBViewer Widgets Blog



The Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

Try it in your browser | Install the Notebook

Language of choice

The Notebook has support for over 40 programming languages,

Share notebooks

Notebooks can be shared with others using email, Dropbox, GitHub

Interactive output

Your code can produce rich, interactive output: HTML, images,

Big data integration

Leverage big data tools, such as Apache Spark, from Python, R and

Demo: Ship classification in Jupyter Notebook

localhost

Home | script

jupyter **script** (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Code

In [ ]:
```
# Load some helpful packages

# linear algebra
import numpy as np

# dataset package
import json

# display package
from matplotlib import pyplot as plt

# image processing package (HOG)
from skimage import color
from skimage.feature import hog

# machine learning package (SVM)
from sklearn import svm
from sklearn.metrics import classification_report,accuracy_score

#from subprocess import check_output
#print(check_output(["ls", "./shipsnet"]).decode("utf8"))
```

In [ ]:
```
# Load the dataset pre-packaged in the json file
f = open(r'./shipsnet.json')
dataset = json.load(f)
f.close()

print(len(dataset))
print(dataset.keys())
print(len(dataset['data']))
```

Load the data and see check how an image looks like

In [ ]:
```
%matplotlib inline
data = np.array(dataset['data']).astype('uint8')
img_length = 80
data = data.reshape(-1,3,img_length,img_length).transpose([0,2,3,1])

print(data.shape)
plt.imshow(data[51])
```

Demo: Ship classification in Jupyter Notebook

Convert the images to grayscale colorspace before calculating the HOG features for each image

In [ ]:
```
data_gray = [ color.rgb2gray(i) for i in data]
plt.imshow(data_gray[51])
```

In [ ]:
```
ppc = 16
hog_images = []
hog_features = []
for image in data_gray:
    fd,hog_image = hog(image, orientations=8, pixels_per_cell=(ppc,ppc),cells_per_block=(4, 4),block_norm= 'L2',visuali
    hog_images.append(hog_image)
    hog_features.append(fd)
```

The hog function of skimage returns a matrix that can be used to visualize the gradients

In [ ]:
```
plt.imshow(hog_images[51])
```

In [ ]:
```
labels =  np.array(dataset['labels']).reshape(len(dataset['labels']),1)
```

Demo: Ship classification in Jupyter Notebook

Fit a simple SVM classifier to the data . Make sure to shuffle the data before fitting it to the model

In [ ]:
```
clf = svm.SVC()
hog_features = np.array(hog_features)
data_frame = np.hstack((hog_features,labels))
np.random.shuffle(data_frame)
```

In [ ]:
```
#What percentage of data you want to keep for training
percentage = 80
partition = int(len(hog_features)*percentage/100)
```

In [ ]:
```
x_train, x_test = data_frame[:partition,:-1],  data_frame[partition:,:-1]
y_train, y_test = data_frame[:partition,-1:].ravel() , data_frame[partition:,-1:].ravel()

clf.fit(x_train,y_train)
```

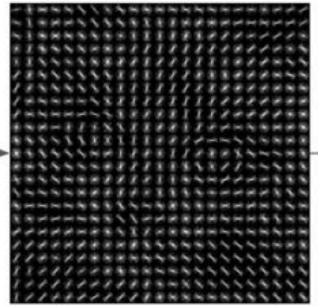
In [ ]:
```
y_pred = clf.predict(x_test)
```

In [ ]:
```
print("Accuracy: "+str(accuracy_score(y_test, y_pred)))
print('\n')
print(classification_report(y_test, y_pred))
```

This shows that we can gain considerably good results with computer vision approaches alone.

Input image

Preprocessing



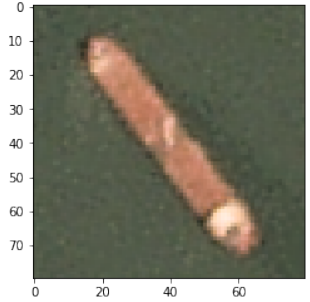
Features : HAAR, HOG, SIFT, SURF

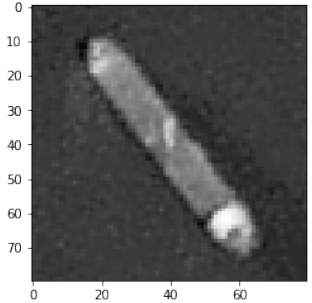
Learning Algorithm

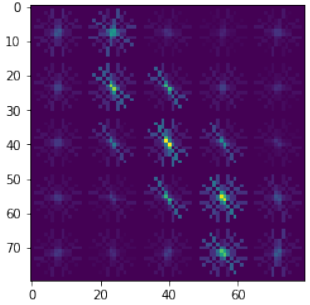
SVM, Random Forests, ANN

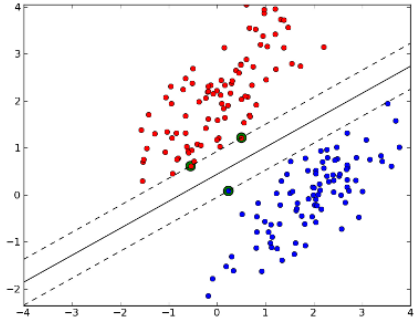
Label Assignment

Cat or Background









Ship/Not-ship?

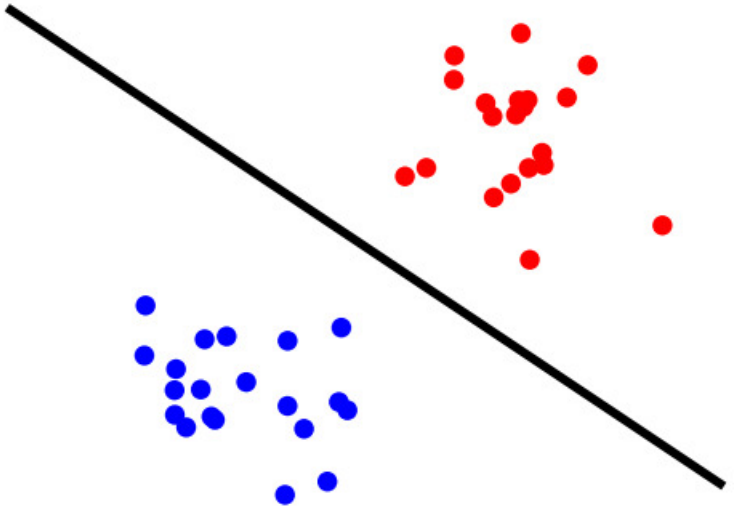
2.2. OCR – Handwritten digit classification

Calculate the HOG descriptor

```
import cv2

winSize = (20,20)
blockSize = (8,8)
blockStride = (4,4)
cellSize = (8,8)
nbins = 9
derivAperture = 1
winSigma = -1.
histogramNormType = 0
L2HysThreshold = 0.2
gammaCorrection = 1
nlevels = 64
signedGradient = True

hog =
cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins,derivAperture,winSigma,histogramNormType,L2H
ysThreshold,gammaCorrection,nlevels, signedGradient)


descriptor = hog.compute(im)
```

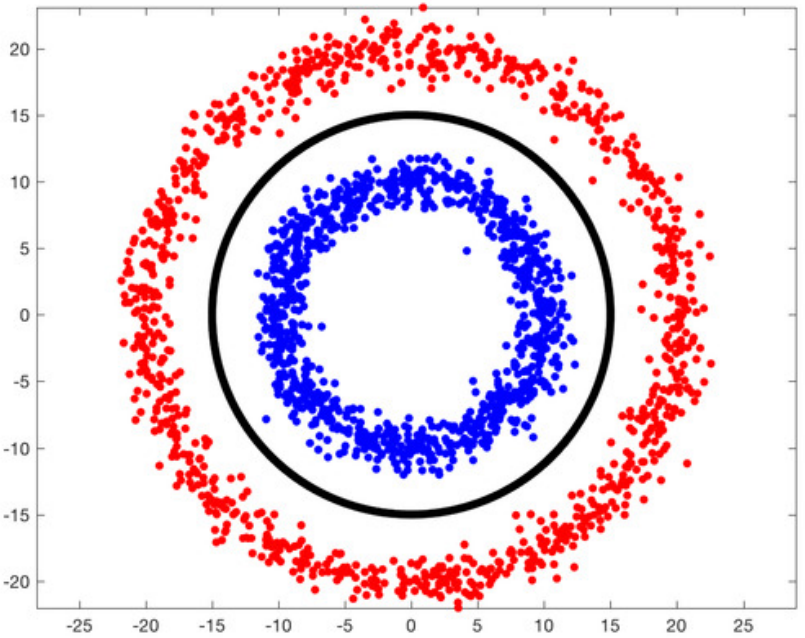
Training a Model with SVM

- Linear SVM will divide the space using planes such that different classes are on different sides of the plane.
- There are many lines that could have separated this data. SVM chooses the one that is at a maximum distance data points of either class



Training a Model with SVM

- Non-linear SVM using kernel tricks to learn non-linear classification planes



Training a Model (a.k.a Learning a Classifier)

```
1  def svmInit(C=12.5, gamma=0.50625):
2    model = cv2.ml.SVM_create()
3    model.setGamma(gamma)
4    model.setC(C)
5    model.setKernel(cv2.ml.SVM_RBF)
6    model.setType(cv2.ml.SVM_C_SVC)
7
8    return model
9
10 def svmTrain(model, samples, responses):
11   model.train(samples, cv2.ml.ROW_SAMPLE, responses)
12   return model
13
14 def svmPredict(model, samples):
15   return model.predict(samples)[1].ravel()
```

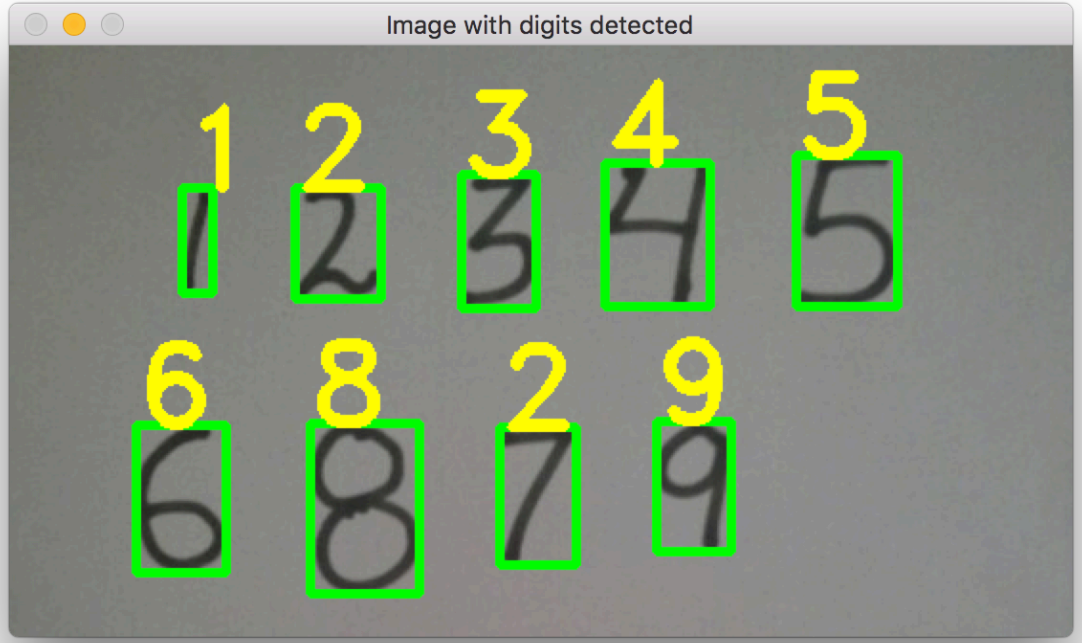
- model = svmInit()
- svmTrain(model, hog_descriptors_train, labels_train)

Predict using the trained Model

```
def svmInit(C=12.5, gamma=0.50625):
  model = cv2.ml.SVM_create()
  model.setGamma(gamma)
  model.setC(C)
  model.setKernel(cv2.ml.SVM_RBF)
  model.setType(cv2.ml.SVM_C_SVC)

  return model

def svmTrain(model, samples, responses):
  model.train(samples, cv2.ml.ROW_SAMPLE, responses)
  return model

def svmPredict(model, samples):
  return model.predict(samples)[1].ravel()
```

- img= cv2.resize(img, (20, 20), interpolation=cv2.INTER_AREA)
- hog_descriptor = cv2.transpose(hog.compute(img))
- nbr = svmPredict(model, hog_descriptor)

Demo: OCR



Still curious?

- MNIST database of handwritten digits
  - 70000 samples of 10 handwritten digits from 0 to 9, approximate 7000 samples for each digit
- HOG
  - https://www.youtube.com/watch?v=0Zib1YEE4LU
- Best OCR open source
  - https://github.com/tesseract-ocr/tesseract’
  - https://www.learnopencv.com/deep-learning-based-text-recognition-ocr-using-tesseract-and-opencv/

Homework

- Write a program to recognise your handwritten digits from your webcam

Part 3. Object Detection

- 3.1. Theory
- 3.2. Using pre-trained human detection model
- 3.3. Building your own detection model

3.1. Theory

- Object Classification
  - Classify an image into different object classes regardless of the object location
- Object Detection = Multiple-objects classification + localization
  - Detect all objects in the image, their classes and locations

Techniques to be covered

- Histogram of Oriented Gradients (HOG)
- Image pyramids
- Sliding windows
- Support Vector Machine (SVM)

Histogram of Oriented Gradients (HOG)

- HOG is a feature descriptor, widely used in image/video processing to detect objects
  - to generalize the object in such a way that the same object produces as close as possible to the same feature descriptor when viewed under different conditions.
  - global feature: an entire region is represented by a single feature vector
- Principal working mechanism
  - an image is divided into portions
  - a gradient for each portion is calculated
  - a Support Vector Machine (SVM) was trained to to recognize/classify

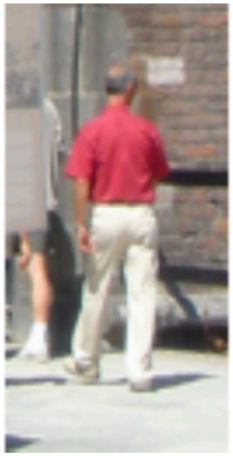
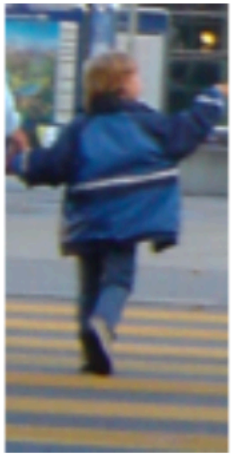
N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection, CVPR 2005.

Histogram of Oriented Gradients (HOG)

- Uses a sliding detection window which is moved around the image
- At each position of the detector window, a HOG descriptor is computed for the detection window
- This descriptor is then shown to the trained SVM, which classifies it as either “person” or “not a person”
- To recognize persons at different scales, the image is subsampled to multiple sizes. Each of these subsampled images is searched

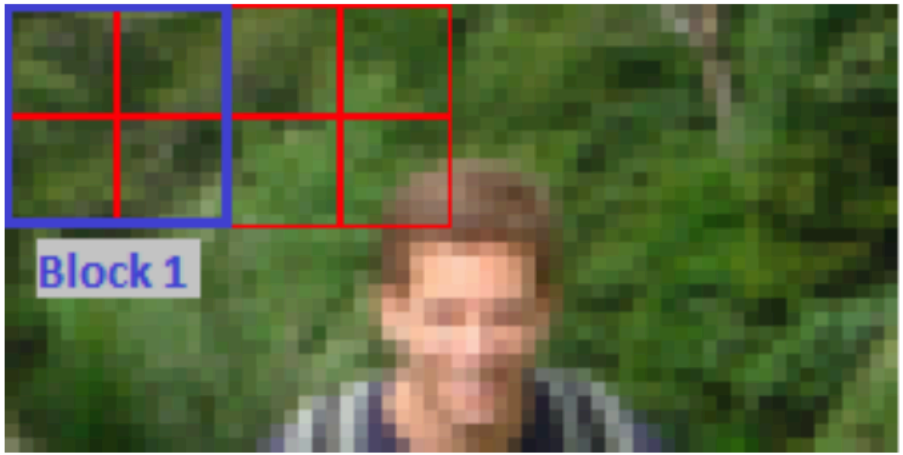
N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection, CVPR 2005.



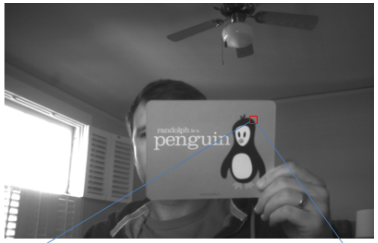

- Input images:
  - 64 x 128

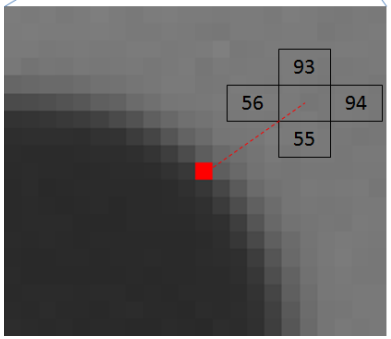
Sliding detection window

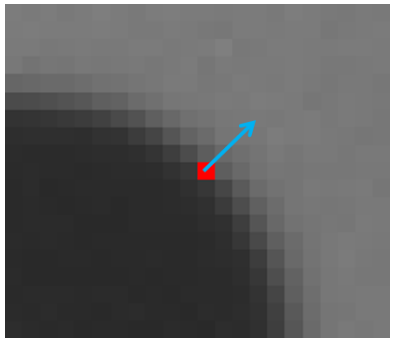


- Blocks
  - 16 x 16
  - Divided into 2x2 cells (each cell size is 8x8)

Gradient Histograms







- Within each cell, compute the gradient vector at each pixel
  - Convolution with [-1 0 1] filters

-1	0	1

F1

-1
0
1

F2

x direction, apply F1, 94-56 = 38

y direction, apply F2, 93-55 = 38

Gradient vector: $\begin{bmatrix} 38 \\ 38 \end{bmatrix}$

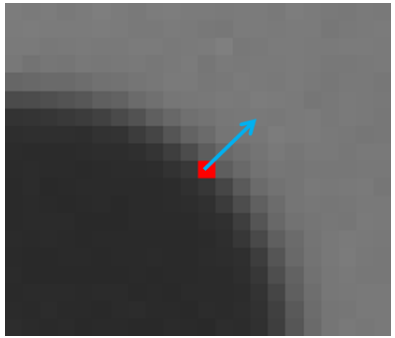
$$\text{Magnitude} = \sqrt{(38)^2+(38)^2} = 53.74$$

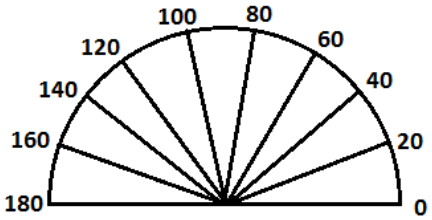
$$\text{Angle} = \arctan\left(\frac{38}{38}\right) = 0.785 \text{ rads}$$

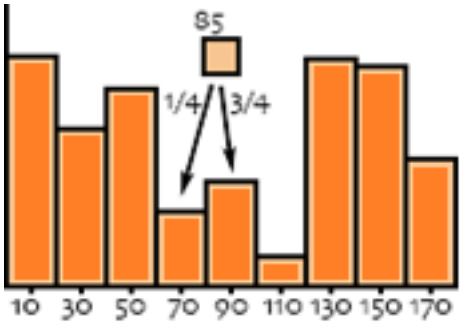
$$= 45 \text{ degrees}$$

Gradient Histograms

- With 64 gradient vectors (in our 8x8 pixel cell) and put them into a 9-bin histogram. The Histogram ranges from 0 to 180 degrees, so there are 20 degrees per bin.
    - The vote is the gradient magnitude weighted
    - if a gradient vector has an angle of 85 degrees, then we add 1/4th of its magnitude to the bin centered at 70 degrees, and 3/4ths of its magnitude to the bin centered at 90.

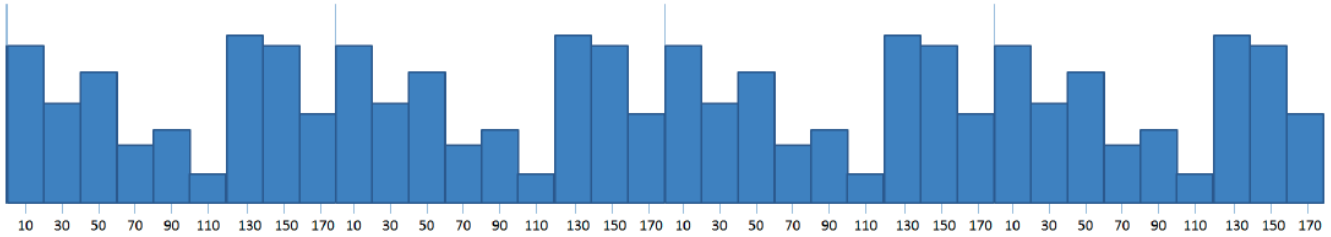






Gradient Histograms

- Normalize histogram
  - Rather than normalize each histogram individually, the cells are first grouped into blocks of 2x2 cells and normalized based on all histograms in the block, the blocks have 50% overlap
  - Block normalization is performed by concatenating the histograms of the four cells within the block into a vector with 36 components (4 cell histograms x 9 bins per histogram), divide this vector by its magnitude to normalize it

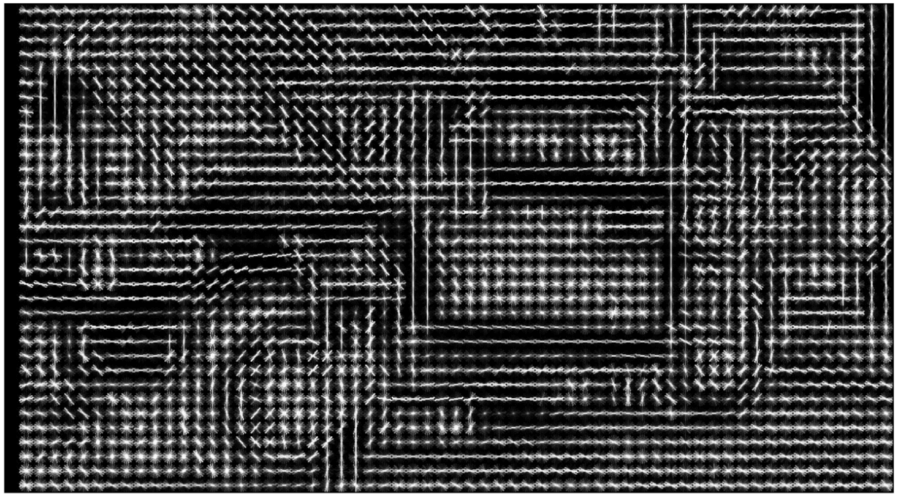


Final HOG descriptor

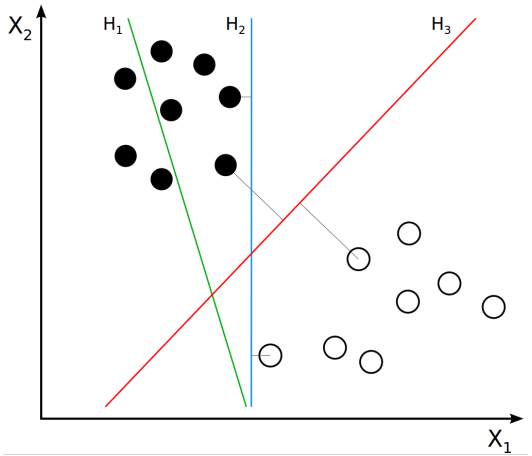
- The 64 x 128 pixel detection window is divided into 105 blocks:
  - 7 blocks horizontally
  - 15 blocks vertically
- 36 values per block:
  - 4 cells each with a 9-bin histogram for each cell
- HOG descriptor has 3,780 values
  - 7 blocks x 15 blocks x 4 cells per block x 9-bins per histogram

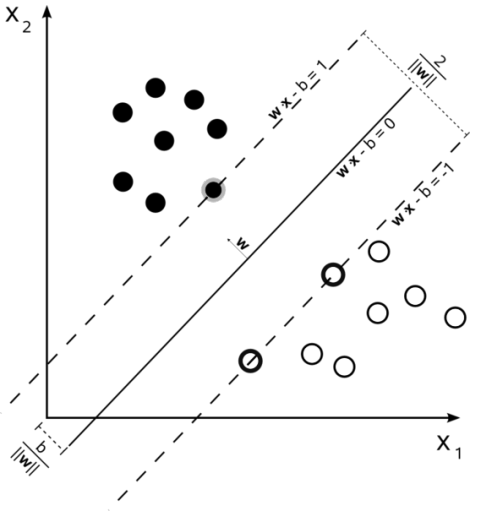
How HOG *sees* the world?

A sample truck image & its HOG version



Support Vector Machine (SVM) classification





- Maximum margin classifier: H3 separates classes “better” than H2
- Training examples defining separating hyperplane: support vectors
- If not linearly separable: use kernel trick to map to higher-dimensional space

3.2. Using pre-trained human detection model

- 1. OpenCV comes with HOGDescriptor that performs people detection.

```
import cv2
import numpy as np

def draw_person(image, person):
    x, y, w, h = person
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 2)

img = cv2.imread("../images/people.jpg")
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

found, w = hog.detectMultiScale(img)
for person in enumerate(found):
     draw_person(img, person)

cv2.imshow("people detection", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
import cv2
import numpy as np

def is_inside(o, i):
    ox, oy, ow, oh = o
    ix, iy, iw, ih = i
    return ox > ix and oy > iy and ox + ow < ix + iw and oy + oh < iy + ih
def draw_person(image, person):
    x, y, w, h = person
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 2)

img = cv2.imread("../images/people.jpg")
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

found, w = hog.detectMultiScale(img)
found_filtered = []
for ri, r in enumerate(found):
    for qi, q in enumerate(found):
        if ri != qi and is_inside(r, q):
            break
        else:
            found_filtered.append(r)

for person in found_filtered:
      draw_person(img, person)

cv2.imshow("people detection", img)
cv2.waitKey(0)
```

HOG detectMultiScale parameters explained

https://www.pyimagesearch.com/2015/11/16/hog-detectmultiscale-parameters-explained/

3.3. Building your own detection model

- The classification model does not deal well with the potential variations:
  - Angle view, object deformation
    - Bag of Word
  - Size
    - Pyramid
  - Localization
    - Sliding window

3.3. Building your own detection model

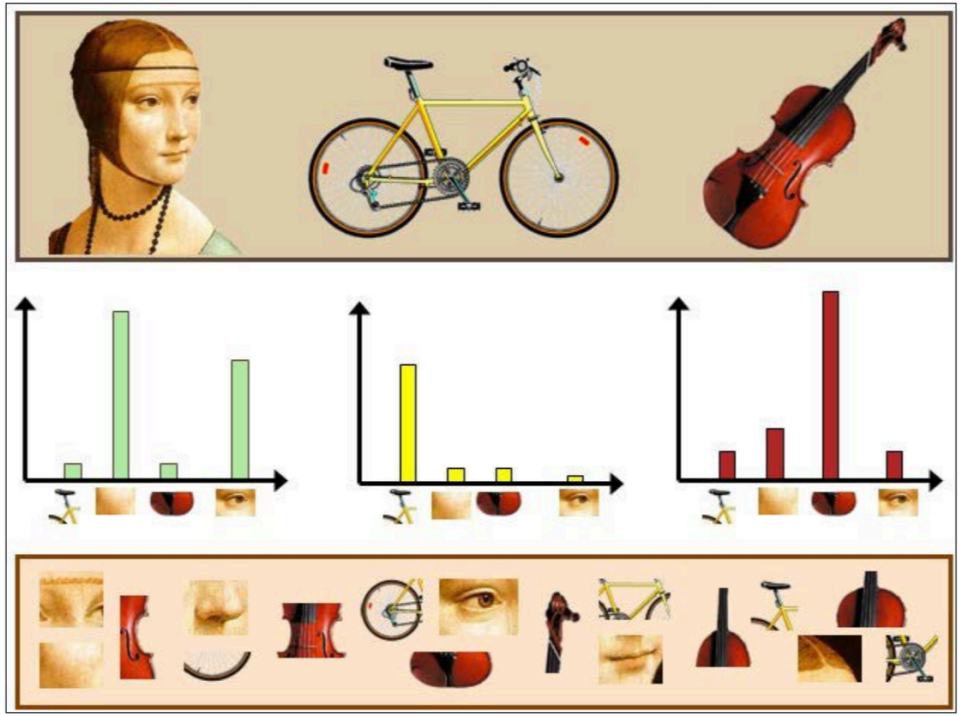
- Bag-of-words
  - Originally from language processing and information retrieval
  - BOW is the technique by which we assign a count weight to each word in a series of documents; we then represent these documents with vectors that represent these set of counts.

- **Document 1**: I like OpenCV and I like Python
- **Document 2**: I like C++ and Python
- **Document 3**: I don't like artichokes

```
[2, 2, 1, 1, 1, 0, 0, 0]
[1, 1, 0, 1, 1, 1, 0, 0]
[1, 1, 0, 0, 0, 0, 1, 1]
```

```
{
    I: 4,
    like: 4,
    OpenCV: 2,
    and: 2,
    Python: 2,
    C++: 1,
    dont: 1,
    artichokes: 1
}
```

BOW in computer vision



- 1. Take a sample dataset.
- 2. For each image in the dataset, extract descriptors (with SIFT, SURF, and so on).
- 3. Add each descriptor to the BOW trainer.
- 4. Cluster the descriptors to *k* clusters whose centers (centroids) are our visual words.

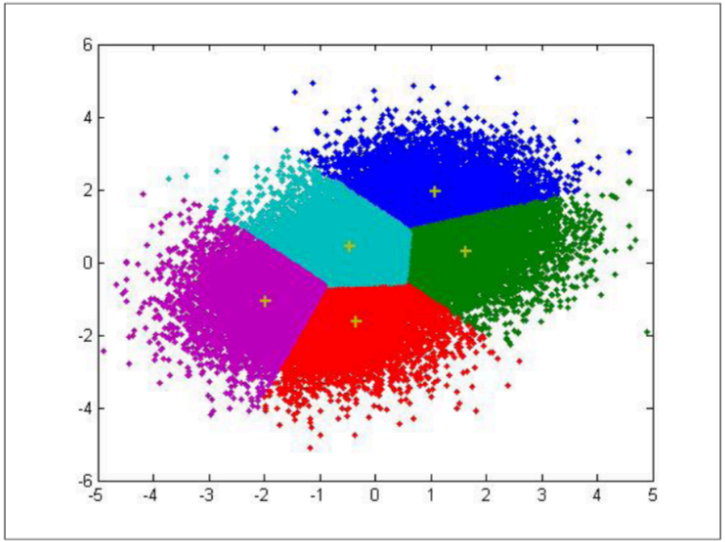
BOW in computer vision

- TRAIN:
  - 1. Take a sample dataset.
  - 2. For each image in the dataset, extract descriptors (with SIFT, SURF, and so on).
  - 3. Add each descriptor to the BOW trainer.
  - 4. Cluster the descriptors to *k* clusters whose centers (centroids) are our visual words.

- TEST
  - given a test image, we can extract features and quantize them based on their distance to the nearest centroid to form a histogram.

The k-means clustering

- **Clustering** refers to the grouping of points in a dataset into clusters.
- Given a dataset, *k* represents the number of clusters in which the dataset is going to be divided.
- "-mean" of a cluster is its **centroid** or the geometrical center of points in the cluster.



Build your own model: car detection

- There is no virtual limit to the type of objects you can detect in your images and videos. However, to obtain an acceptable level of accuracy, you need a sufficiently large dataset, containing train images that are identical in size.

- You can collect data yourself (for your projects), but here we use ready-made datasets of cars for demonstration

    - **The University of Illinois**: http://l2r.cs.uiuc.edu/~cogcomp/Data/ Car/CarData.tar.gz

Code time

```
import cv2
import numpy as np
from os.path import join

datapath = "./CarData/TrainImages/"
def path(cls,i):
  return "%s/%s%d.pgm"  % (datapath,cls,i+1)

pos, neg = "pos-", "neg-"

detect = cv2.xfeatures2d.SIFT_create()
extract = cv2.xfeatures2d.SIFT_create()

flann_params = dict(algorithm = 1, trees = 5)
matcher = cv2.FlannBasedMatcher(flann_params, {})

bow_kmeans_trainer = cv2.BOWKMeansTrainer(40)
extract_bow = cv2.BOWImgDescriptorExtractor(extract,
matcher)

def extract_sift(fn):
  im = cv2.imread(fn,0)
  return extract.compute(im, detect.detect(im))[1]

for i in range(8):
  bow_kmeans_trainer.add(extract_sift(path(pos,i)))
  bow_kmeans_trainer.add(extract_sift(path(neg,i)))

voc = bow_kmeans_trainer.cluster()
extract_bow.setVocabulary( voc )

def bow_features(fn):
  im = cv2.imread(fn,0)
  return extract_bow.compute(im, detect.detect(im))
```

```
traindata, trainlabels = [],[]
for i in range(20):
  traindata.extend(bow_features(path(pos, i))); trainlabels.append(1)
  traindata.extend(bow_features(path(neg, i))); trainlabels.append(-1)

svm = cv2.ml.SVM_create()
svm.train(np.array(traindata), cv2.ml.ROW_SAMPLE,
np.array(trainlabels))

def predict(fn):
  f = bow_features(fn);
  p = svm.predict(f)
  print(fn, "\t", p[1][0][0])
  return p

car, notcar = "../images/car.jpg", "../images/bb.jpg"
car_img = cv2.imread(car)
notcar_img = cv2.imread(notcar)
car_predict = predict(car)
not_car_predict = predict(notcar)

font = cv2.FONT_HERSHEY_SIMPLEX

if (car_predict[1][0][0] == 1.0):
  cv2.putText(car_img,'Car Detected',(10,30), font,
1,(0,255,0),2,cv2.LINE_AA)

if (not_car_predict[1][0][0] == -1.0):
  cv2.putText(notcar_img,'Car Not Detected',(10,30), font, 1,(0,0,
255),2,cv2.LINE_AA)

cv2.imshow('BOW + SVM Success', car_img)
cv2.imshow('BOW + SVM Failure', notcar_img)
cv2.waitKey(0); cv2.destroyAllWindows()
```

```
import cv2
import numpy as np
from os.path import join
```

- Our usual imports

```
datapath = "./CarData/TrainImages/"
def path(cls,i):
  return "%s/%s%d.pgm"  % (datapath,cls,i+1)

pos, neg = "pos-", "neg-"
```

- Variables and function to make life easier

```
detect = cv2.xfeatures2d.SIFT_create()
extract = cv2.xfeatures2d.SIFT_create()

flann_params = dict(algorithm = 1, trees = 5)
matcher = cv2.FlannBasedMatcher(flann_params, {})
```

- SIFT features & matcher

```
bow_kmeans_trainer = cv2.BOWKMeansTrainer(40)
extract_bow = cv2.BOWImgDescriptorExtractor(extract,
matcher)
```

- A BOW trainer with 40 clusters, using visual words extracted by the SIFT features & flann matcher

```
def extract_sift(fn):
  im = cv2.imread(fn,0)
  return extract.compute(im, detect.detect(im))[1]
```

- A utility method to extract the SIFT features from an image

```
for i in range(8):
  bow_kmeans_trainer.add(extract_sift(path(pos,i)))
  bow_kmeans_trainer.add(extract_sift(path(neg,i)))
```

- Read eight images per class (8 positives & 8 negatives)

```
voc = bow_kmeans_trainer.cluster()
extract_bow.setVocabulary( voc )
```

- Performs the k-means classification & returns the vocabulary

```
def bow_features(fn):
  im = cv2.imread(fn,0)
  return extract_bow.compute(im, detect.detect(im))
```

- A function that takes the path to an image and returns the descriptor as computed by the BOW descriptor extractor

```
traindata, trainlabels = [],[]
for i in range(20):
  traindata.extend(bow_features(path(pos, i)));
  trainlabels.append(1)
  traindata.extend(bow_features(path(neg, i)));
  trainlabels.append(-1)
```

- Create two arrays to accommodate the train data and labels

```
svm = cv2.ml.SVM_create()
svm.train(np.array(traindata), cv2.ml.ROW_SAMPLE,
np.array(trainlabels))
```

- Create & train an SVM by the train data and labels

FINISH TRAINING

```
def predict(fn):
  f = bow_features(fn);
  p = svm.predict(f)
  print(fn, "\t", p[1][0][0])
  return p
```

- A function to predict the class for a new image

```
car, notcar = "../images/car.jpg", "../images/bb.jpg"
car_img = cv2.imread(car)
notcar_img = cv2.imread(notcar)
car_predict = predict(car)
not_car_predict = predict(notcar)
```

- Test one positive and one negative image

```
font = cv2.FONT_HERSHEY_SIMPLEX

if (car_predict[1][0][0] == 1.0):
  cv2.putText(car_img,'Car Detected',(10,30), font,
1,(0,255,0),2,cv2.LINE_AA)

if (not_car_predict[1][0][0] == -1.0):
  cv2.putText(notcar_img,'Car Not Detected',(10,30),
font, 1,(0,0, 255),2,cv2.LINE_AA)

cv2.imshow('BOW + SVM Success', car_img)
cv2.imshow('BOW + SVM Failure', notcar_img)
cv2.waitKey(0); cv2.destroyAllWindows()
```

```
import cv2
import numpy as np
from os.path import join

datapath = "./CarData/TrainImages/"
def path(cls,i):
  return "%s/%s%d.pgm"  % (datapath,cls,i+1)

pos, neg = "pos-", "neg-"

detect = cv2.xfeatures2d.SIFT_create()
extract = cv2.xfeatures2d.SIFT_create()

flann_params = dict(algorithm = 1, trees = 5)
matcher = cv2.FlannBasedMatcher(flann_params, {})

bow_kmeans_trainer = cv2.BOWKMeansTrainer(40)
extract_bow = cv2.BOWImgDescriptorExtractor(extract,
matcher)

def extract_sift(fn):
  im = cv2.imread(fn,0)
  return extract.compute(im, detect.detect(im))[1]

for i in range(8):
  bow_kmeans_trainer.add(extract_sift(path(pos,i)))
  bow_kmeans_trainer.add(extract_sift(path(neg,i)))

voc = bow_kmeans_trainer.cluster()
extract_bow.setVocabulary( voc )

def bow_features(fn):
  im = cv2.imread(fn,0)
  return extract_bow.compute(im, detect.detect(im))
```

```
traindata, trainlabels = [],[]
for i in range(20):
  traindata.extend(bow_features(path(pos, i))); trainlabels.append(1)
  traindata.extend(bow_features(path(neg, i))); trainlabels.append(-1)

svm = cv2.ml.SVM_create()
svm.train(np.array(traindata), cv2.ml.ROW_SAMPLE,
np.array(trainlabels))
```

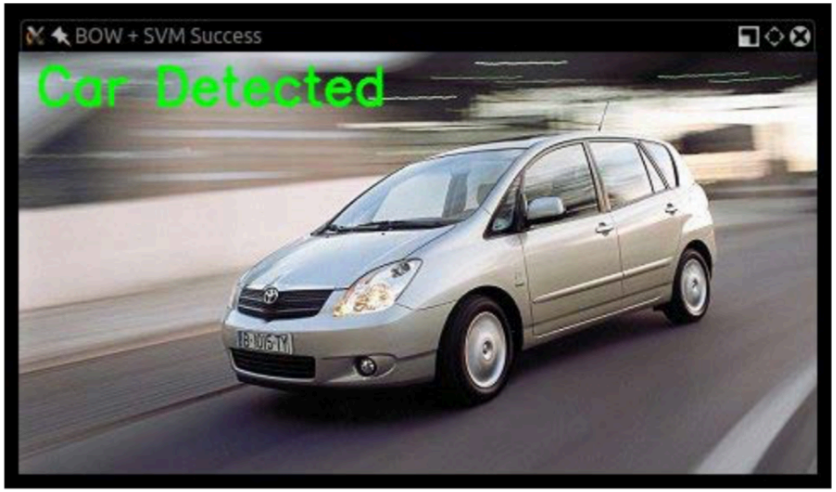
TRAIN

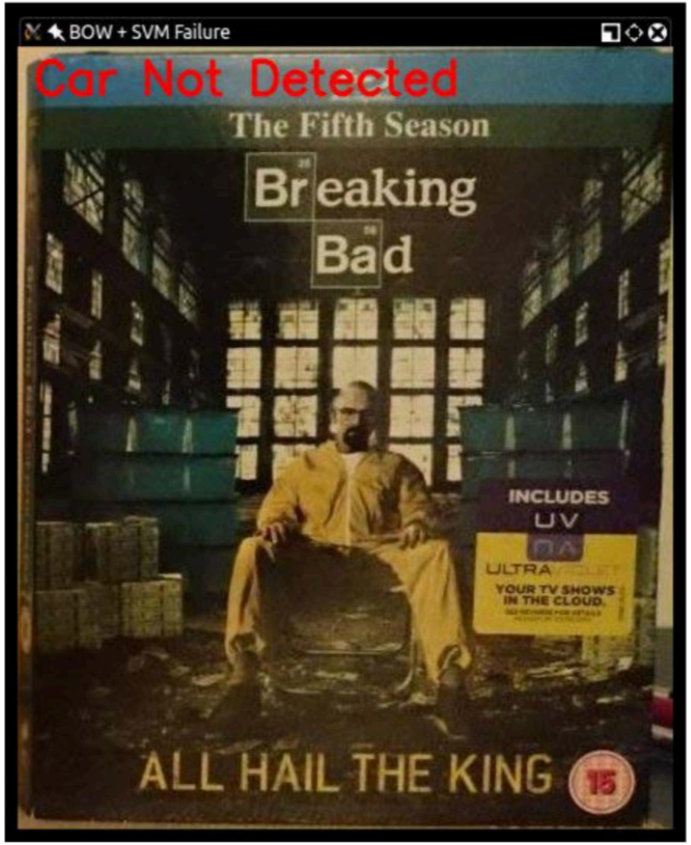
```
def predict(fn):
  f = bow_features(fn);
  p = svm.predict(f)
  print(fn, "\t", p[1][0][0])
  return p

car, notcar = "../images/car.jpg", "../images/bb.jpg"
car_img = cv2.imread(car)
notcar_img = cv2.imread(notcar)
car_predict = predict(car)
not_car_predict = predict(notcar)
```

TEST

```
font = cv2.FONT_HERSHEY_SIMPLEX

if (car_predict[1][0][0] == 1.0):
  cv2.putText(car_img,'Car Detected',(10,30), font,
1,(0,255,0),2,cv2.LINE_AA)

if (not_car_predict[1][0][0] == -1.0):
  cv2.putText(notcar_img,'Car Not Detected',(10,30), font, 1,(0,0,
255),2,cv2.LINE_AA)

cv2.imshow('BOW + SVM Success', car_img)
cv2.imshow('BOW + SVM Failure', notcar_img)
cv2.waitKey(0); cv2.destroyAllWindows()
```

Expected results



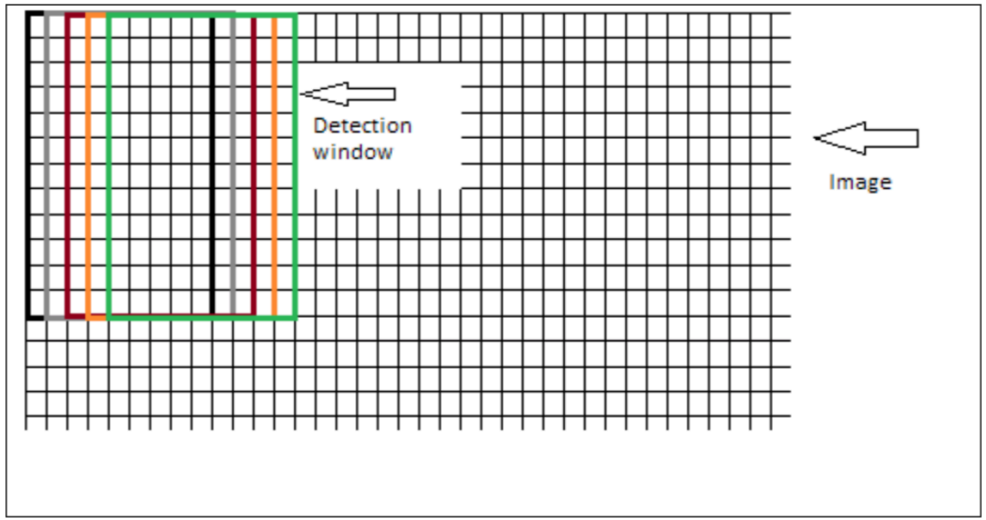


- Now demo time

A real-life detection model

- Detecting multiple objects of the same kind in an image
- Determining the position of a detected object in an image

- A sliding windows approach



Localize multiple objects at multiple scales

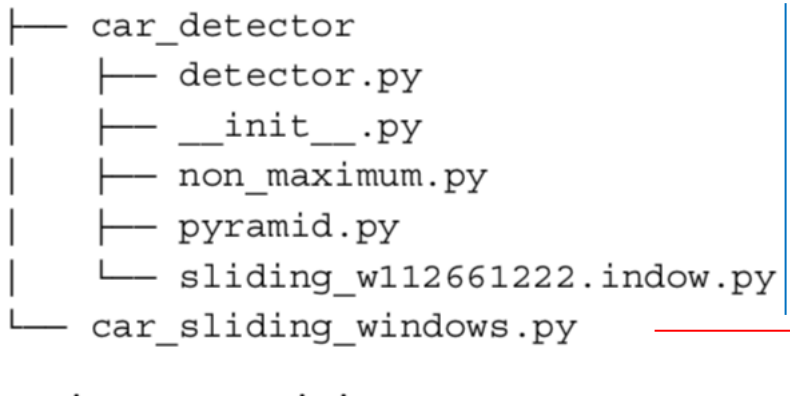
- Observe the movement of the block:
  - We take a region of the image, classify it, and then move a predefined step size to the right-hand side. When we reach the rightmost end of the image, we'll reset the x coordinate to 0 and move down a step, and repeat the entire process.
  - At each step, we'll perform a classification with the SVM that was trained with BOW.
  - Keep a track of all the blocks that have *passed* the SVM predict test.
  - When you've finished classifying the entire image, scale the image down and
  - repeat the entire sliding windows process.
- Continue rescaling and classifying until you get to a minimum size.

A real-life car detector

- Let's summarize the process before diving into the code:

1. Obtain a train dataset.
2. Create a BOW trainer and create a visual vocabulary.
3. Train an SVM with the vocabulary.
4. Attempt detection using sliding windows on an image pyramid of a test image.
5. Apply non-maximum suppression to overlapping boxes.
6. Output the result.

Hierarchy of a python project



Utility codes in a folder

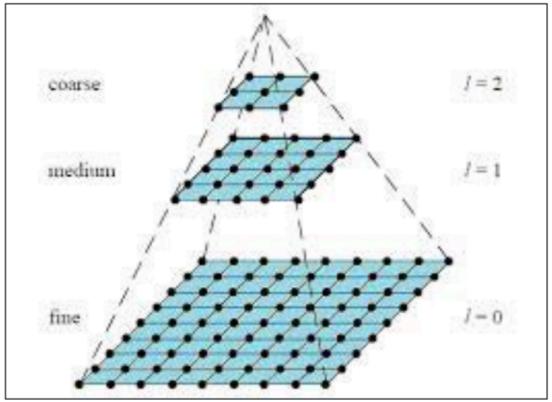
__init__.py file in the folder to be detected as a module

Main program

pyramid.py

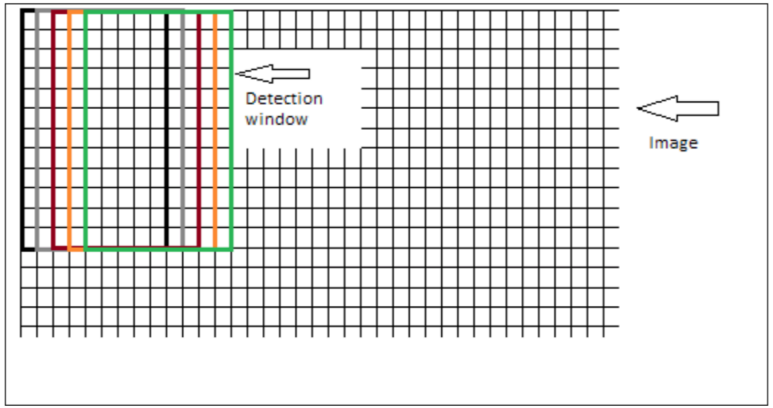
```
import cv2

def resize(img, scaleFactor):
  return cv2.resize(img, (int(img.shape[1] * (1 /
scaleFactor)), int(img.shape[0] * (1 /
scaleFactor))), interpolation=cv2.INTER_AREA)

def pyramid(image, scale=1.5, minSize=(200, 80)):
  yield image

  while True:
    image = resize(image, scale)
    if image.shape[0] < minSize[1] or image.shape[1]
< minSize[0]:
      break

    yield image
```

Generator



sliding_window.py

```
def sliding_window(image, step, window_size):
  for y in xrange(0, image.shape[0], step):
    for x in xrange(0, image.shape[1], step):
      yield (x, y, image[y:y + window_size[1], x:x +
window_size[0]])
```



non_maximum.py

simply takes a list of rectangles and sorts them by their score. Starting from the box with the highest score, it eliminates all boxes that overlap beyond a certain threshold by calculating the area of intersection and determining whether it is greater than a certain threshold.

```
import numpy as np

# Malisiewicz et al.
# Python port by Adrian Rosebrock
def non_max_suppression_fast(boxes, overlapThresh):
  # if there are no boxes, return an empty list
  if len(boxes) == 0:
    return []

  # if the bounding boxes integers, convert them to floats --
  # this is important since we'll be doing a bunch of divisions
  if boxes.dtype.kind == "i":
    boxes = boxes.astype("float")

  # initialize the list of picked indexes
  pick = []

  # grab the coordinates of the bounding boxes
  x1 = boxes[:,0]
  y1 = boxes[:,1]
  x2 = boxes[:,2]
  y2 = boxes[:,3]
  scores = boxes[:,4]
  # compute the area of the bounding boxes and sort the bounding
  # boxes by the score/probability of the bounding box
  area = (x2 - x1 + 1) * (y2 - y1 + 1)
  idxs = np.argsort(scores)[::-1]
```

```
# keep looping while some indexes still remain in the indexes
  # list
  while len(idxs) > 0:
    # grab the last index in the indexes list and add the
    # index value to the list of picked indexes
    last = len(idxs) - 1
    i = idxs[last]
    pick.append(i)

    # find the largest (x, y) coordinates for the start of
    # the bounding box and the smallest (x, y) coordinates
    # for the end of the bounding box
    xx1 = np.maximum(x1[i], x1[idxs[:last]])
    yy1 = np.maximum(y1[i], y1[idxs[:last]])
    xx2 = np.minimum(x2[i], x2[idxs[:last]])
    yy2 = np.minimum(y2[i], y2[idxs[:last]])

    # compute the width and height of the bounding box
    w = np.maximum(0, xx2 - xx1 + 1)
    h = np.maximum(0, yy2 - yy1 + 1)

    # compute the ratio of overlap
    overlap = (w * h) / area[idxs[:last]]

    # delete all indexes from the index list that have
    idxs = np.delete(idxs, np.concatenate(([last],
      np.where(overlap > overlapThresh)[0])))

  # return only the bounding boxes that were picked using the
  # integer data type
  return boxes[pick].astype("int")
```

detector.py

```
import cv2
import numpy as np
from car_detector.detector import car_detector,
bow_features
from car_detector.pyramid import pyramid
from car_detector.non_maximum import
non_max_suppression_fast as nms
from car_detector.sliding_window import sliding_window
import urllib

def in_range(number, test, thresh=0.2):
  return abs(number - test) < thresh

test_image = "../images/cars.jpg"
img_path = "../images/test.jpg"

urllib.urlretrieve(test_image, img_path)

svm, extractor = car_detector()
detect = cv2.xfeatures2d.SIFT_create()

w, h = 100, 40
img = cv2.imread(img_path)
#img = cv2.imread(test_image)

rectangles = []
counter = 1
scaleFactor = 1.25
scale = 1
font = cv2.FONT_HERSHEY_PLAIN

for resized in pyramid(img, scaleFactor):
  scale = float(img.shape[1]) / float(resized.shape[1])
  for (x, y, roi) in sliding_window(resized, 20, (100,
40)):
```

```
# keep looping while some indexes still remain in the indexes
  # list
  while len(idxs) > 0:
    # grab the last index in the indexes list and add the
    # index value to the list of picked indexes
    last = len(idxs) - 1
    i = idxs[last]
    pick.append(i)

    # find the largest (x, y) coordinates for the start of
    # the bounding box and the smallest (x, y) coordinates
    # for the end of the bounding box
    xx1 = np.maximum(x1[i], x1[idxs[:last]])
    yy1 = np.maximum(y1[i], y1[idxs[:last]])
    xx2 = np.minimum(x2[i], x2[idxs[:last]])
    yy2 = np.minimum(y2[i], y2[idxs[:last]])

    # compute the width and height of the bounding box
    w = np.maximum(0, xx2 - xx1 + 1)
    h = np.maximum(0, yy2 - yy1 + 1)

    # compute the ratio of overlap
    overlap = (w * h) / area[idxs[:last]]

    # delete all indexes from the index list that have
    idxs = np.delete(idxs, np.concatenate(([last],
      np.where(overlap > overlapThresh)[0])))

  # return only the bounding boxes that were picked using the
  # integer data type
  return boxes[pick].astype("int")
```

Deep learning object detectors from opencv 3.4.1+

- https://github.com/opencv/opencv/tree/master/samples/dnn