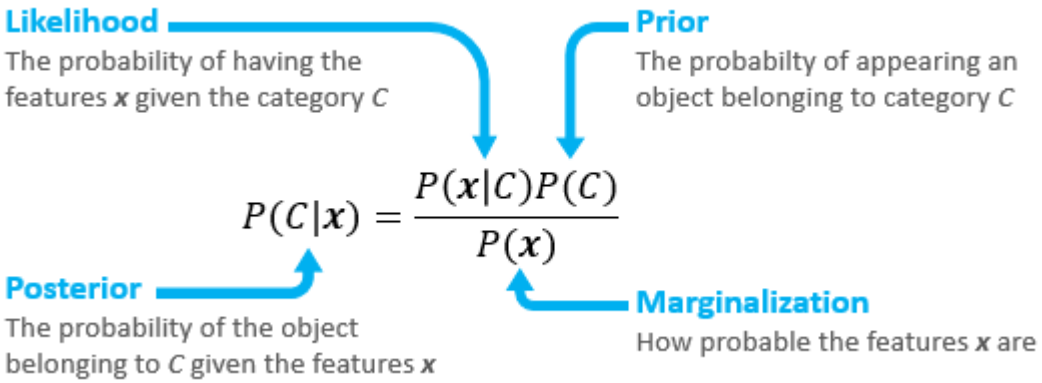
Cuadernos opcionales realizados: 7.2, 7.3, 7.5

7.2 Bayesian Classifier based on Normal Distribution

In this second notebook of the object recognition chapter we will explore a popular statistical classifiers, the Bayesian ones. These methods are based on the idea that, given a number of features characterizing an object (the famous vector $\mathbf{x}$), the Bayes' rule can be used to predict its class. Recall the Bayes' rule:



In this way, Bayesian classifiers build probabilistic (statistical) models of the features according to certain training data, and use them to classify new objects.

In this notebook, will explain:

- how (Naïve) Bayesian classifiers work (section 7.2.1) and,
- specifically, how to classify feature vectors supposing that they follow a normal distribution (section 7.2.2).

Problem context - Traffic sign recognition

In the previous notebook, *AliquindoiCars* contacted us looking for a TSR technique to be integrated into a self-driving car. They provided us with some segmented images of traffic signs that their autonomous cars captured during test drivings. These images are located in `images/circles/` containing circled signs, `images/triangles` containing signs having triangular shapes and `images/squares` containing signs having square shapes.

Previously, we extracted a feature vector from each image using Hu moments. Now, we will train a classifier using the feature vectors we saved the in previous notebook. For classification, there are many methods we can apply, such as kNN algorithm, random forest, etc. In this notebook, we will explore and use a classical one, the Naïve Bayes classifier!

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
import scipy.interpolate
matplotlib.rcParams['figure.figsize'] = (8.0, 8.0)

images_path = './images/'
import sys
sys.path.append("..")
from utils.PlotEllipse import PlotEllipse
```

7.2.1 (Naïve) Bayesian classifier

The simplest case of these classifiers is the **Naïve Bayesian** one, which considers the strong (naïve) independence assumption that the input features are conditionally independent of each other given the object class. That is, for example, if we are using compactness and extent to describe objects, this classifier assumes that the value for both features is not related if the object class is known, i.e. circle. For the visual system in our previous notebook, which is in charge of recognizing objects in a kitchen, this means that if it knows that an object is a spoon, then its compactness would be unrelated with its extent (we could not say anything about its possible extent given a certain compactness, for example).

In a general case with an arbitrary number of features n, the Bayes' rule can be written as:

$$P(C|x_1,\ldots,x_n) = \frac{P(x_1,\ldots,x_n|C)P(C)}{P(x_1,\ldots,x_n)}$$

if the following naïve conditional assumption:

$$P(x_i|C, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|C)$$

is considered for each feature i, then the rule can be simplified to:

$$P(C|x_1, \ldots, x_n) = \frac{P(C)\prod_{i=1}^{n} P(x_i|C)}{P(x_1, \ldots, x_n)}$$

Given that $P(x_1, \ldots, x_n)$ is constant for a certain input, and that we are looking for the class that has the highest posterior probability, the following classification rule is considered:

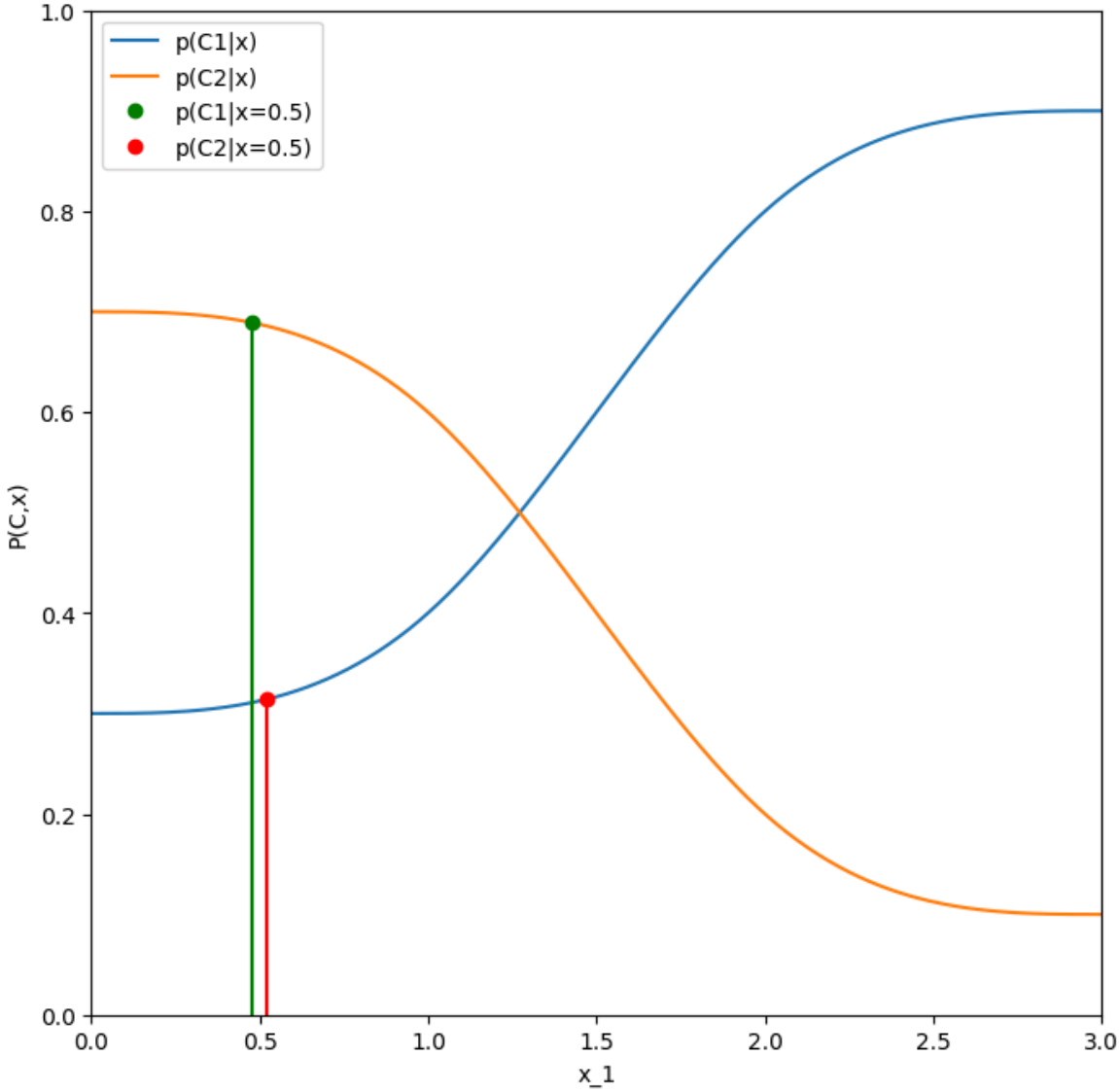
$$P(C|x_1, \ldots, x_n) \propto P(C)\prod_{i=1}^{n} P(x_i|C)$$

$$\hat{C} = arg\underset{C}{max} P(C)\prod_{i=1}^{n} P(x_i|C)$$

Summarizing, having a set of features describing an object $\mathbf{x} = [x_1, x_2, x_3, \ldots, x_n]^T$ and a set of possible belonging classes $C = [C_1, C_2, C_3, \ldots, C_n]$, the Bayesian classifier assigns $\mathbf{x}$ to the class C_i that has the highest posterior probability $P(C_i/x)$ (the more probable, the less probability of making a mistake). This is called a **MAP** (Maximum A Posteriori) prediction.

Considering the following joint probability distribution $P(C, \mathbf{x})$ (in this case $\mathbf{x}$ contains just one variable):

In [2]:
```
x = np.array([0, 1, 2, 3])
pC1_given_x = np.array([0.3, 0.4, 0.8, 0.9])
pC2_given_y = np.array([0.7, 0.6, 0.2, 0.1])

x_new = np.linspace(0, 3, 300)
a_BSpline = scipy.interpolate.make_interp_spline(x, pC1_given_x, bc_type="natura
b_BSpline = scipy.interpolate.make_interp_spline(x, pC2_given_y, bc_type="natura
pC1_given_x_new = a_BSpline(x_new)
pC2_given_y_new = b_BSpline(x_new)

plt.plot(x_new, pC1_given_x_new, label='p(C1|x)')
plt.plot(x_new, pC2_given_y_new, label='p(C2|x)')
plt.plot([0.48,0.48],[0, 0.687],'g'); plt.plot(0.48,0.689,'og', label="p(C1|x=0.
plt.plot([0.52,0.52],[0, 0.32],'r'); plt.plot(0.52,0.315,'or', label="p(C2|x=0.5

plt.ylabel('P(C,x)')
plt.xlabel('x_1')
plt.ylim([0,1])
plt.xlim([0,3])
plt.legend();
```

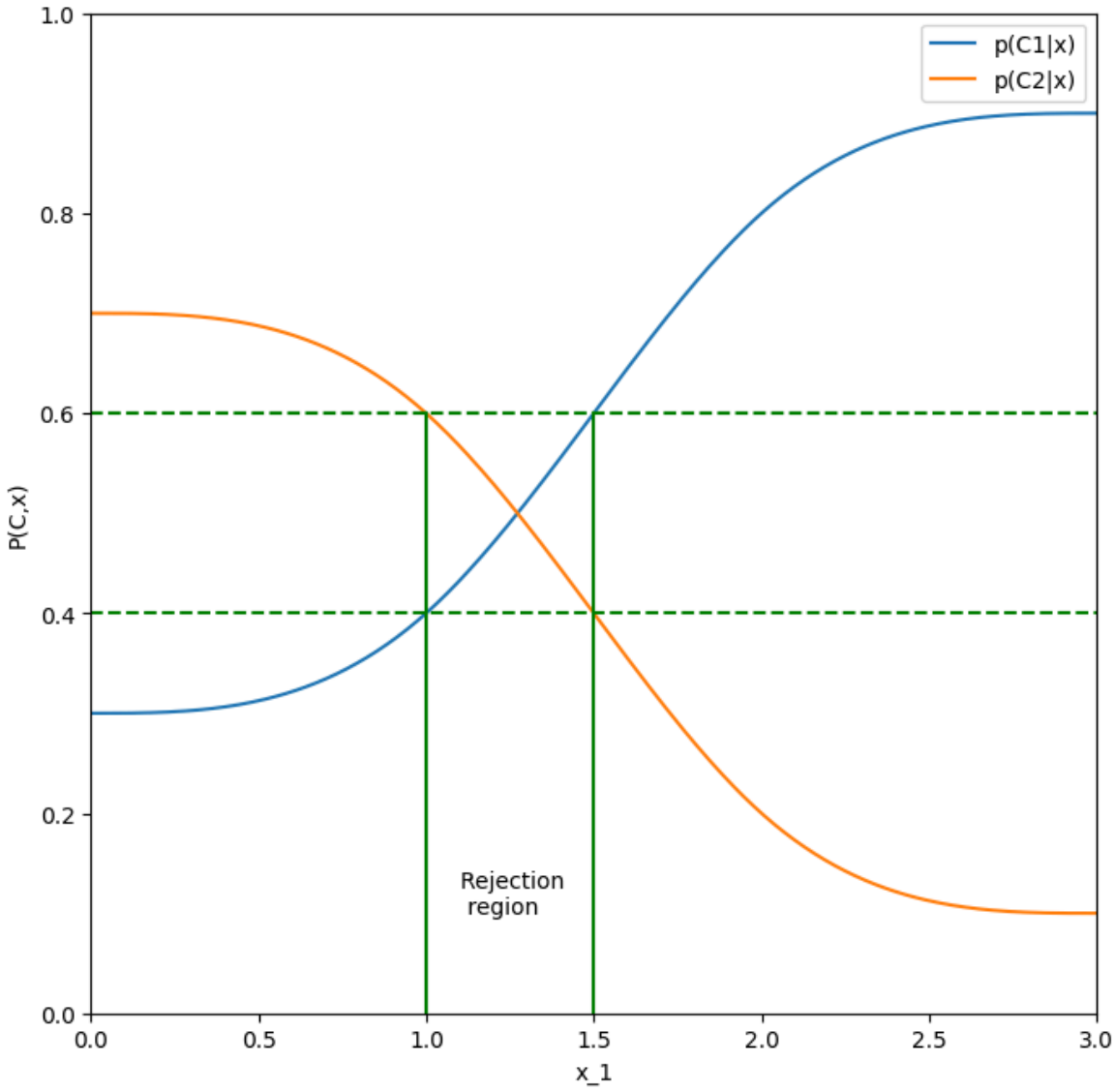


this MAP classification corresponds to assign the object to the class C_i with the highest value. Since such an object is characterized by $x = x_1 = 0.5$, it is assigned to C_2 (assuming equal prior probability for each class), the category with highest $P(C|x = x_1)$ value.

Depending on the application, it is convenient to consider a *rejection region*, where none probability is high enough and no decision is made (e.g. $P(C_1|x) = P(C_2|x) = 0.5$). The following code illustrates this, where a probability threshold θ defining such a region:

In [3]:
```
plt.plot(x_new, pC1_given_x_new, label='p(C1|x)')
plt.plot(x_new, pC2_given_y_new, label='p(C2|x)')

plt.ylabel('P(C,x)')
plt.xlabel('x_1')
plt.ylim([0,1])
plt.xlim([0,3])
plt.legend();

theta = 0.1 # threshold

plt.plot([0, 3],[0.5+theta, 0.5+theta],'g--')
plt.plot([0, 3],[0.5-theta, 0.5-theta],'g--')
```

```
plt.plot([1, 1],[0, 0.6],'g')
plt.plot([1.5, 1.5],[0, 0.6],'g')
plt.text(1.1, 0.1, 'Rejection \n region');
```



Thinking about it (1)

Now that you have notions about the rejection region, **answer the following questions**:

- Which class would an object with `x_1=0.5` belong to?

  It would belong to C2, as the value is higher than the set threshold.

- Which class would an object with `x_1=1.2` belong to?

  Since this value is in the rejection region, no decision is made so it does not belong to any class.

- Would the threshold theta be the same in any application?

  No, it depends on the application. For instance, if you really need to make sure it belongs to a specific class, the threshold would be higher.

Building discriminant functions

In the same way as we built linear and generalized discriminant functions in the previous notebook, those functions can be also defined to design a Bayesian classifier. For that, the goal is to obtain a discriminant function $d_i(x)$ for each class C_i, such that $d_i(x) > d_j(x)$ whenever $P(C_i|x) > P(C_j|x)$. Let's design them step by step!

From the Bayes' rule we have that :

$$d_k(x) = P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)P(C_k)}{P(\mathbf{x})}$$

This is, Bayes' rule defines a way to compute the posterior probability $P(C_k|\mathbf{x})$ of a feature vector $\mathbf{x}$ for each class C_k. We can further simplify this function:

$$d_k(x) = P(C_k/\mathbf{x}) = \frac{p(\mathbf{x}|C_k)P(C_k)}{P(\mathbf{x})} \tag{1}$$

$$P(\mathbf{x}) \text{ is a constant value } \forall k \Big\Downarrow \tag{2}$$

$$d_k(x) = p(\mathbf{x}|C_k)P(C_k) \tag{3}$$

$$\max \ln(f(\mathbf{x})) = \max f(\mathbf{x}) \Big\Downarrow \tag{4}$$

$$d_k(x) = \ln p(\mathbf{x}|C_k) + \ln P(C_k) \tag{5}$$

$$\text{If } P(C_k) = P(C_j)\ \forall j,k \Big\Downarrow \tag{6}$$

$$d_k(x) = lnp(\mathbf{x}|C_k) \tag{7}$$

The resulting formulation is also called **MLE (Maximum log-Likelihood Estimation)**.

7.2.2 Naive Bayesian classifier for normal distribution

The different Naïve Bayesian Classifiers differ in the probability distribution considered for modeling $P(x_i|C)$. In this section we will cover **Normal distribution based ones**, which suppose that input features follow the probability density function of a Gaussian distribution:

$$p(\mathbf{x}|C_i) = \frac{1}{(2\pi)^{n/2}|\Sigma^i|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu^i)^T\Sigma_i^{-1}(\mathbf{x}-\mu^i)}$$

so two set of parameters are considered:

- A mean vector for each class: $\mu = [\mu_1 \mu_2 \dots \mu_f \dots \mu_n]^T$
- **A covariance matrix for each class: $\hspace{5pt} \small \Sigma = E[(\mathbf{x}-\mathbf{\mu}) \cdot (\mathbf{x}-\mathbf{\mu})^T]**

$$\begin{bmatrix} \sigma_{11} & \cdots & \sigma_{1f} & \cdots & \sigma_{1n} \\ \vdots & & \vdots & & \vdots \\ \sigma_{n1} & \cdots & \sigma_{nf} & \cdots & \sigma_{nn} \end{bmatrix}$$

\underbrace{

$$\begin{bmatrix} \sigma_{11} & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & \sigma_{nn} \end{bmatrix}$$

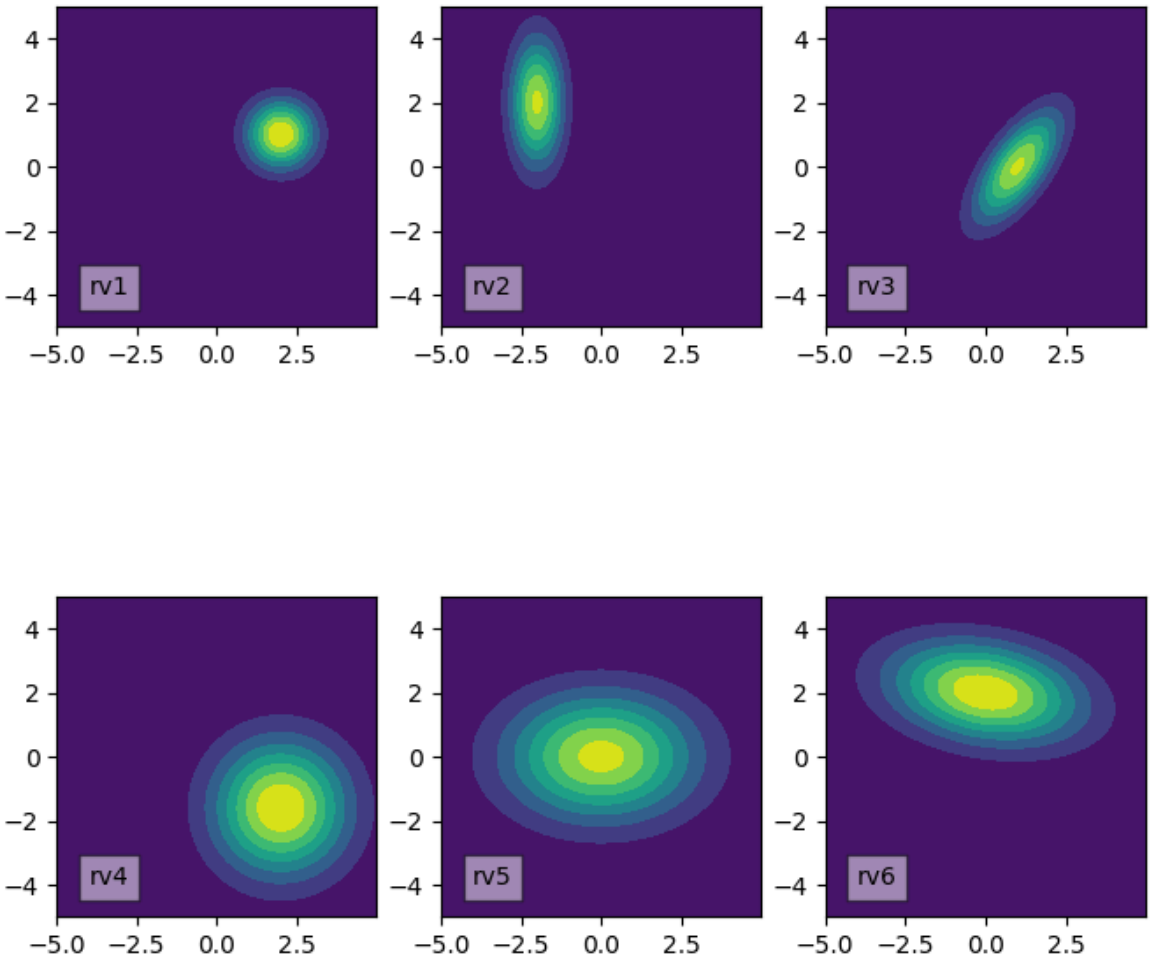
}_{\text{Independence assumption (Naïve Bayes)}} $

The simplification of the covariance matrix can be done by assuming independence (not correlation) among features, that is, the assumption done by Naïve Bayes!

The following code is just a review about how a Gaussian distribution with two variables depends on its two parameters μ and Σ. Feel free to change such parameters and experience their influence!

In [4]:
```
def plot_2d_gaussian(fig, rv, x, y, pos, position):
    """ Plot 2d contours of a 3D gaussian
    """
    label = "rv" + str(position)
    position = str(23)+str(position)
    ax = fig.add_subplot(int(position))
    ax.contourf(x, y, rv.pdf(pos))
    ax.text(-4,-4,label,bbox=dict(facecolor='white', alpha=0.5))
    ax.set_aspect('equal')

from scipy.stats import multivariate_normal
x, y = np.mgrid[-5:5:.01, -5:5:.01]
pos = np.dstack((x, y))

# Define 6 different Gaussian distributions
mean1 = np.array([2.0, 1.0])
covar1 = np.array([[0.5, 0.0],[0.0, 0.5]])
rv1 = multivariate_normal(mean1, covar1)

mean2 = np.array([-2.0, 2.0])
covar2 = np.array([[0.3, 0.0],[0.0, 1.8]])
rv2 = multivariate_normal(mean2, covar2)

mean3 = np.array([1.0, 0.0])
covar3 = np.array([[0.8, 0.7],[0.7, 1.3]])
rv3 = multivariate_normal(mean3, covar3)

mean4 = np.array([2.0, -1.6])
covar4 = np.array([[2.0, 0.0],[0.0, 2.0]])
rv4 = multivariate_normal(mean4, covar4)
```

```
mean5 = np.array([0.0, 0.0])
covar5 = np.array([[4.0, 0.0],[0.0, 1.8]])
rv5 = multivariate_normal(mean5, covar5)

mean6 = np.array([0.0, 2.0])
covar6 = np.array([[3.9, -0.5],[-0.5, 1.1]])
rv6 = multivariate_normal(mean6, covar6)

# Show the contours fo the 3D gaussians
fig = plt.figure()
plot_2d_gaussian(fig,rv1,x,y,pos,1)
plot_2d_gaussian(fig,rv2,x,y,pos,2)
plot_2d_gaussian(fig,rv3,x,y,pos,3)
plot_2d_gaussian(fig,rv4,x,y,pos,4)
plot_2d_gaussian(fig,rv5,x,y,pos,5)
plot_2d_gaussian(fig,rv6,x,y,pos,6)
```



Designing the discriminant function

It is time to see how we could define the discriminant function $d_k(\mathbf{x})$ for this classifier:

$$d_k(\mathbf{x}) = \ln P(C_k) + \ln p(\mathbf{x}|C_k) = \ln P(C_k) + \ln \frac{1}{(2\pi)^{n/2}|\Sigma^k|^{1/2}} e^{-\frac{1}{2}\overbrace{(\mathbf{x}-\mu^k)^T\Sigma_k^{-1}\mathbf{x}-\mu^}^{\text{Squared Mahalanobis dist}}}$$

$$= \ln P(C_k) - \ln 2\pi^{n/2} - \ln |\Sigma^k|^{1/2} - \frac{1}{2} D_k^2(x)$$

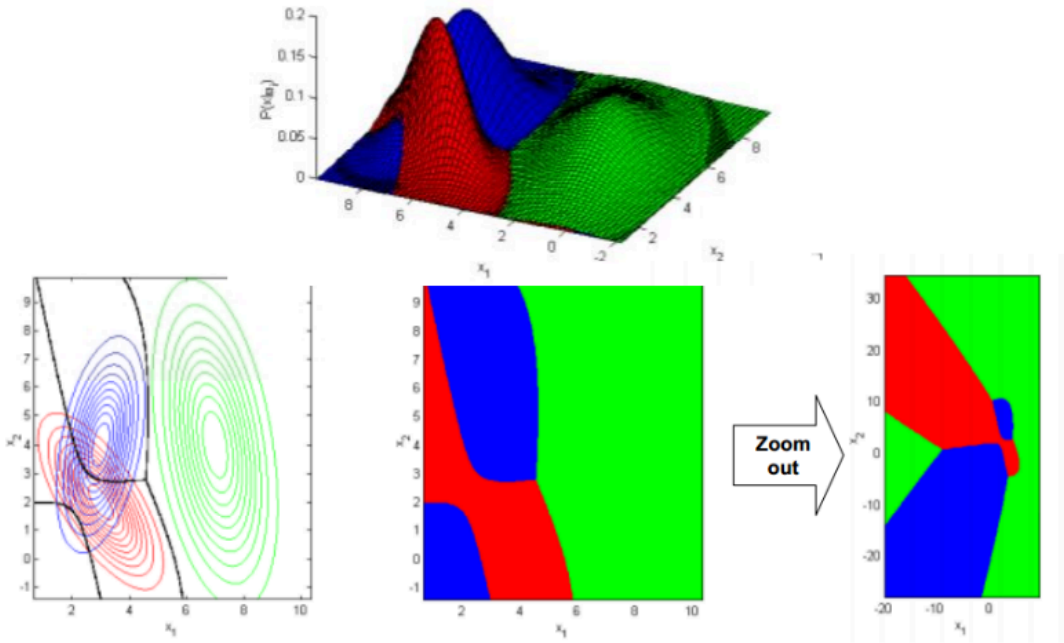
$$= \ln P(C_k) - \frac{1}{2}\left[\underbrace{n \ln 2\pi}_{\text{constant}} + \ln |\Sigma^k| + D_k^2(x)\right]$$

$$= \ln P(C_k) - \frac{1}{2}\left[\ln |\Sigma^k| + D_k^2(x)\right]$$

We can see that the resulting **discriminant function is quadratic**:

$$d_k(\mathbf{x}) = ln\ P(C_k) - \underbrace{\frac{1}{2}\left[ln\ |\Sigma^k| + \mu^{kT}(\Sigma^k)^{-1}\mu^k\right]}_{\text{Independent term}} + \underbrace{\mathbf{x}^T(\Sigma^k)^{-1}\mu^k}_{\text{linear term}} - \frac{1}{2}\underbrace{\mathbf{x}^T(\Sigma^k)^{-1}\mathbf{x}}_{\text{Quadratic term}}$$

Visually, the **division boundaries are parabolas**:



ASSIGNMENT 1a: Training the classifier

Now, we are going to implement this Naïve Bayes classifier for normal distributions **using the Hu moments** computed in the previous exercise.

The first step for training such classifier is computing the weights' matrix of the discriminant function. In this case, it depends on the **mean** (μ, dimension $(2,)$) and **covariance matrix** (Σ, dimension $(2,2)$), which can be retrieved from the training data through MLE.

In the previous notebook we proved that our TSR problem can be solved using only the first and second Hu moments. In this one **your first task** is:

- to load the firsts two Hu moments for the images from each class which, as commented, was computed in previous notebook (you can use `np.load()`).
- Then, **compute the mean** (or centroid) and **covariance matrix** for each class. You can compute the covariance matrix of a set of points using `np.cov()`.

In [5]:
```
# Assignment 1a
#
# Load first 2 Hu moments of each class
train_triangles = np.load("./data/hu_triangles.npy")[:,:2].T
train_circles = np.load("./data/hu_circles.npy")[:,:2].T
train_squares = np.load("./data/hu_squares.npy")[:,:2].T

# Compute covariance matrices
cov_triangles = np.cov(train_triangles)
cov_circles = np.cov(train_circles)
cov_squares = np.cov(train_squares)

# Compute means
mean_triangles = np.mean(train_triangles, axis=1)
mean_circles = np.mean(train_circles, axis=1)
mean_squares = np.mean(train_squares, axis=1)

print ('cov_triangles = \n' + str(cov_triangles))
print ('cov_circles = \n' + str(cov_circles))
print ('cov_squares = \n' + str(cov_squares))
print ('mean_triangles = ' + str(mean_triangles))
print ('mean_circles = ' + str(mean_circles))
print ('mean_squares = ' + str(mean_squares))
```

```
cov_triangles =
[[1.17603861e-06 2.24894396e-07]
 [2.24894396e-07 8.80708956e-08]]
cov_circles =
[[1.02189926e-07 3.33231869e-08]
 [3.33231869e-08 1.09259042e-08]]
cov_squares =
[[1.61508686e-06 5.21788207e-07]
 [5.21788207e-07 1.70838002e-07]]
mean_triangles = [0.19254439 0.00034407]
mean_circles = [1.59537533e-01 9.98702969e-05]
mean_squares = [0.16720802 0.00026462]
```

Expected output:

```
cov_triangles =
[[1.17603861e-06 2.24894396e-07]
 [2.24894396e-07 8.80708956e-08]]
cov_circles =
[[1.02189926e-07 3.33231869e-08]
 [3.33231869e-08 1.09259042e-08]]
cov_squares =
[[1.61508686e-06 5.21788207e-07]
 [5.21788207e-07 1.70838002e-07]]
mean_triangles = [0.19254439 0.00034407]
```

```
mean_circles = [1.59537533e-01 9.98702969e-05]
mean_squares = [0.16720802 0.00026462]
```

ASSIGNMENT 1b: Defining the discriminant function

Your **next task** is to develop a method, named `discriminant_function()`, that computes the discriminant function for each class $d_k(x)$. The inputs have to be:

- `features`: feature vector of dimension n (number of features, 2 in our problem).
- `mu`: mean vector of the class k.
- `cov`: covariance matrix with shape (n,n) of the class k.
- `prior`: prior probability of class k.

The method should evaluate (then return) the discriminant function.

In [6]:
```
# Assignment 1b
#
def discriminant_function(features, mu, cov, prior):
    """ Evaluates the discriminant function d(x)

        Args:
            features: feature vector of dimension n
            mu: mean vector of the class of which is being computed the probabil
            cov: covariance matrix with shape (n,n) of the class
            prior: prior probability of class k

        Returns:
            dx: result of discriminant function
    """
    covinv = np.linalg.inv(cov) # Auxiliar variable
    sq_mahalanobis = np.dot(np.dot(np.transpose(features - mu), covinv), (featur
    cov_det = np.linalg.det(cov) # Covariance matrix determinant

    dx = np.log(prior) - 0.5 * (np.log(cov_det) + sq_mahalanobis) # You can divi

    return dx
```

You can try your function with the following test:

In [7]:
```
f = np.array([0.5, 0.6])
mu = np.array([0.7, 0.9])
cov = np.array([[0.7, 0.3],[0.3, 0.9]])
prior = 0.5
d = discriminant_function(f, mu, cov, prior)

print ('d = ' + str(d))
```

```
d = -0.44338744418137005
```

Expected output:

```
d = -0.4433874441813701
```

ASSIGNMENT 1c: Testing the classifier

For testing our brand new classifier, we are going to classify some new images and check the results. *Note that the discriminant function is the logarithm of a probability, not a probability itself (values can be positive and negatives, but the result of the max function is the same).*

What to do?

- Complete the auxiliary function `classify_image()` that:
  1. computes the Hu moments of a testing image `sign_image`, and
  2. uses the discriminant function of each class to retrieve the highest output value. The class returning such a value would be the assigned one!

- After completing such a function, in the code cell below it, call it for the `test_circle.png`, `test_square.png` and `test_triangle.png` images in order to classify them.

We assume that there is no prior information about any class, so $P(C_i) = P(C_j)\ \forall i, j$. This can be interpreted as: while driving, we see the same number of circle, square and triangle shaped road signs.

In [8]:
```
# Assignment 1c
#
def image_moments(region):
    """ Compute moments of the external contour in a binary image.

        Args:
            region: Binary image

        Returns:
            moments: dictionary containing all moments of the region
    """

    # Get external contour
    contours,_ = cv2.findContours(region,cv2.RETR_EXTERNAL ,cv2.CHAIN_APPROX_NON
    cnt = contours[0]

    # Compute moments
    moments = cv2.moments(cnt)

    return moments
```

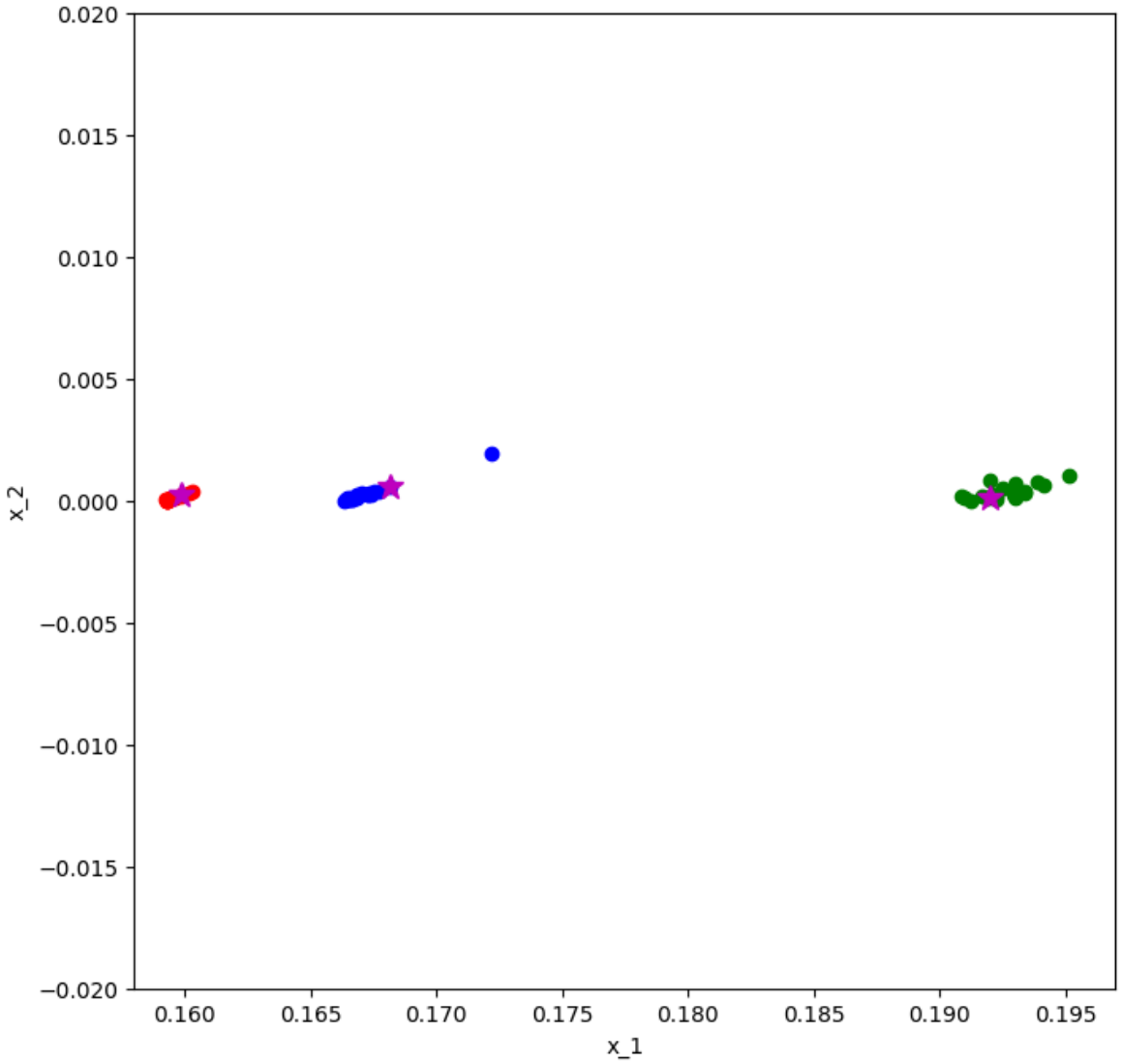
In [9]:
```
def classify_image(sign_image):
    """ Classify a traffic sign image by its shape using a bayesian classifier

        Args:
            sign_image: Binarized image
    """

    # Compute Hu moments
    moments = image_moments(sign_image)
    hu = cv2.HuMoments(moments).flatten()[:2]

    # Classify circle test image
    prior = 1/3
    triangle = discriminant_function(hu,mean_triangles,cov_triangles,prior)
```

```
    circle = discriminant_function(hu,mean_circles,cov_circles,prior)
    square = discriminant_function(hu,mean_squares,cov_squares,prior)

    # Search the maximum
    classification = max([triangle,circle,square])

    if classification == triangle:
        print("The sign is a triangle\n")
    elif classification == circle:
        print("The sign is a circle\n")
    else:
        print("The sign is a square\n")

    return hu
```

In [10]:

```
# Read images
test_circle = cv2.imread(images_path + "test_circle.png", 0)
test_triangle = cv2.imread(images_path + "test_triangle.png", 0)
test_square = cv2.imread(images_path + "test_square.png", 0)

# Classify them
print("Circle: ")
moments_circle = classify_image(test_circle)
print("Triangle: ")
moments_triangle = classify_image(test_triangle)
print("Square: ")
moments_square = classify_image(test_square)

# Create figure
fig, ax = plt.subplots()
plt.axis([0.158, 0.197, -0.02, 0.02])

# Plot hu moments
plt.plot(train_triangles[0,:],train_triangles[1,:],'go')
plt.plot(train_circles[0,:],train_circles[1,:],'ro')
plt.plot(train_squares[0,:],train_squares[1,:],'bo')
plt.xlabel('x_1')
plt.ylabel('x_2')

# Plot testing data
plt.plot(moments_circle[0],moments_circle[1],'*m',markersize=12)
plt.plot(moments_triangle[0],moments_triangle[1],'*m',markersize=12)
plt.plot(moments_square[0],moments_square[1],'*m',markersize=12);
```

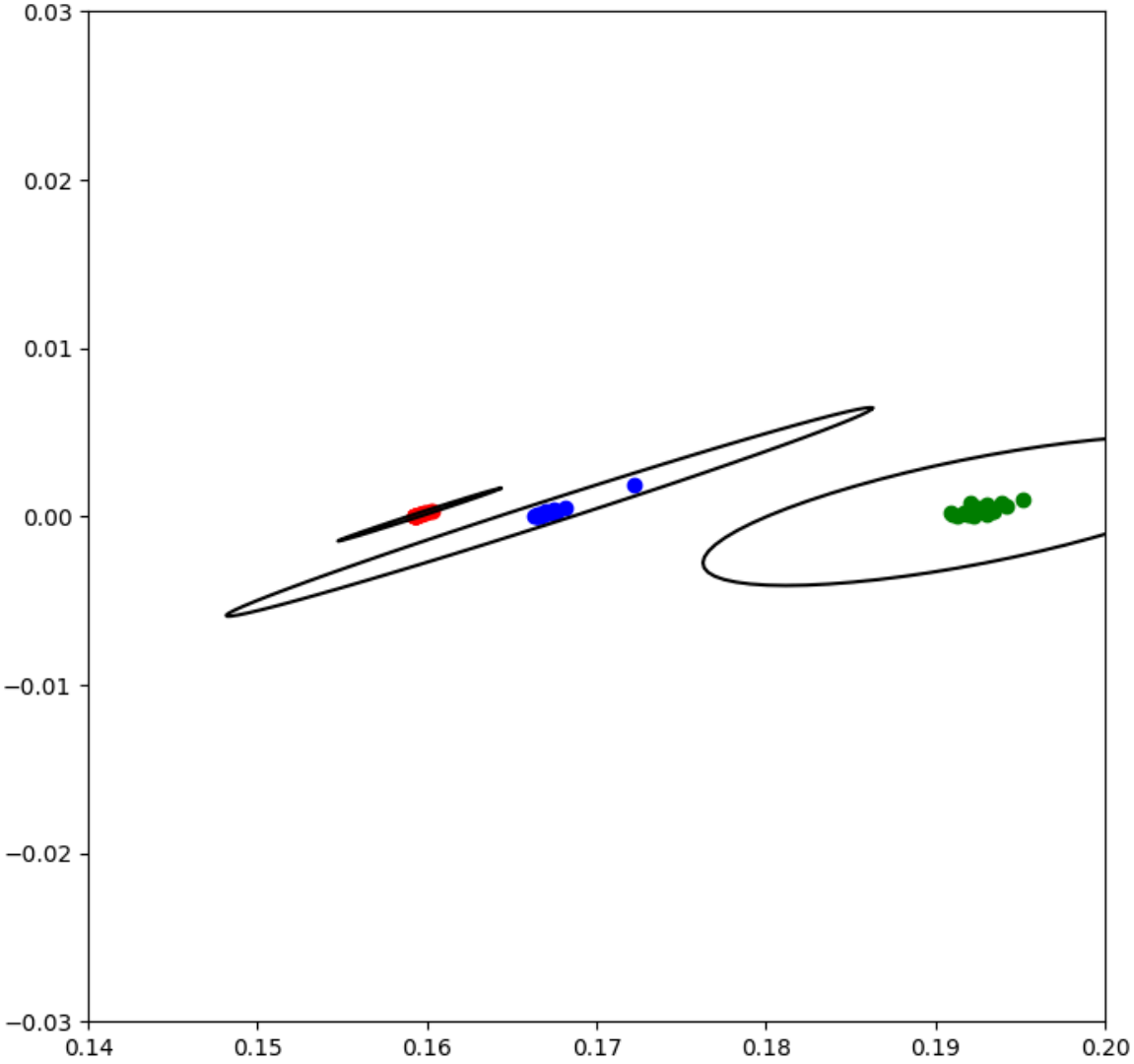
```
Circle:
The sign is a circle

Triangle:
The sign is a triangle

Square:
The sign is a square
```



ASSIGNMENT 1d: Analyzing covariances

Finally, we can see how this classifier divides the feature space showing the computed covariance ellipses. **You have to** complete the following code cell to make it works, showing the covariance ellipses of each class.

In [11]:
```
# Create figure
fig, ax = plt.subplots()
plt.axis([0.14, 0.2, -0.03, 0.03])

# Plot hu moments
plt.plot(train_triangles[0,:],train_triangles[1,:],'go')
plt.plot(train_circles[0,:],train_circles[1,:],'ro')
plt.plot(train_squares[0,:],train_squares[1,:],'bo')

# Plot ellipses representing covariance matrices
PlotEllipse(fig, ax, np.vstack(mean_triangles), cov_triangles, 15, color='black'
PlotEllipse(fig, ax, np.vstack(mean_circles), cov_circles, 15, color='black')
PlotEllipse(fig, ax, np.vstack(mean_squares), cov_squares, 15, color='black')

fig.canvas.draw()
```

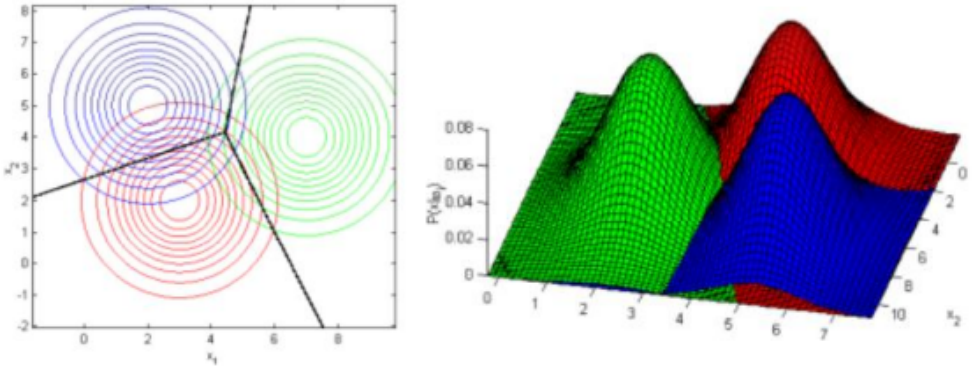


Simplification of the Naïve classifier

The classifier at hand can be simplified if the Euclidean distance is considered instead of the Mahalanobis one. This can be achieved using isotropic covariance matrices:

$$\Sigma^k = \Sigma = \sigma^2 \cdot I = \sigma^2 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In this way, decision boundaries are lines, and covariances are spherical. This is called a **natural classifier**:



Example of feature space with 3 classes characterized by Gaussian distributions with isotropic covariances. Black lines are decision boundaries.

In this case, the discriminant function can be simplified, and the quadratic term disappears:

$$d_k(x) = -(\mathbf{x} - \mu^{\mathbf{k}})^T(\mathbf{x} - \mu^{\mathbf{k}}) = -||\mathbf{x} - \mu^{\mathbf{k}}||^2$$

ASSIGNMENT 2: Playing with isotropic covariance matrices

What to do? Repeat the previous steps but using isotropic covariance matrices. Recall that `np.eye()` defines an identity matrix.

In [12]:
```
# Assignment 2
#
def discriminant_function_isotropic(features, mu):
    """ Evaluates the discriminant function of a naive Bayes clasifier using iso

        Args:
            features: feature vector of dimension n
            mu: mean vector of the class of which is being computed the probabil

        Returns:
            dx: result of discriminant function
    """
    dx = -np.dot(np.transpose(features - mu), features - mu)
    return dx
```

In [13]:
```
def classify_image_isotropic(sign_image):
    """ Classify a traffic sign image by its shape using a bayesian classifier

        Args:
            sign_image: Binarized image
    """

    # Compute Hu moments
    moments = image_moments(sign_image)
    hu = cv2.HuMoments(moments).flatten()[:2]

    # Classify circle test image
    triangle = discriminant_function_isotropic(hu,mean_triangles)
    circle = discriminant_function_isotropic(hu,mean_circles)
    square = discriminant_function_isotropic(hu,mean_squares)

    # Search the maximum
    classification = max([triangle,circle,square])

    if classification == triangle:
        print("The sign is a triangle\n")
    elif classification == circle:
        print("The sign is a circle\n")
```

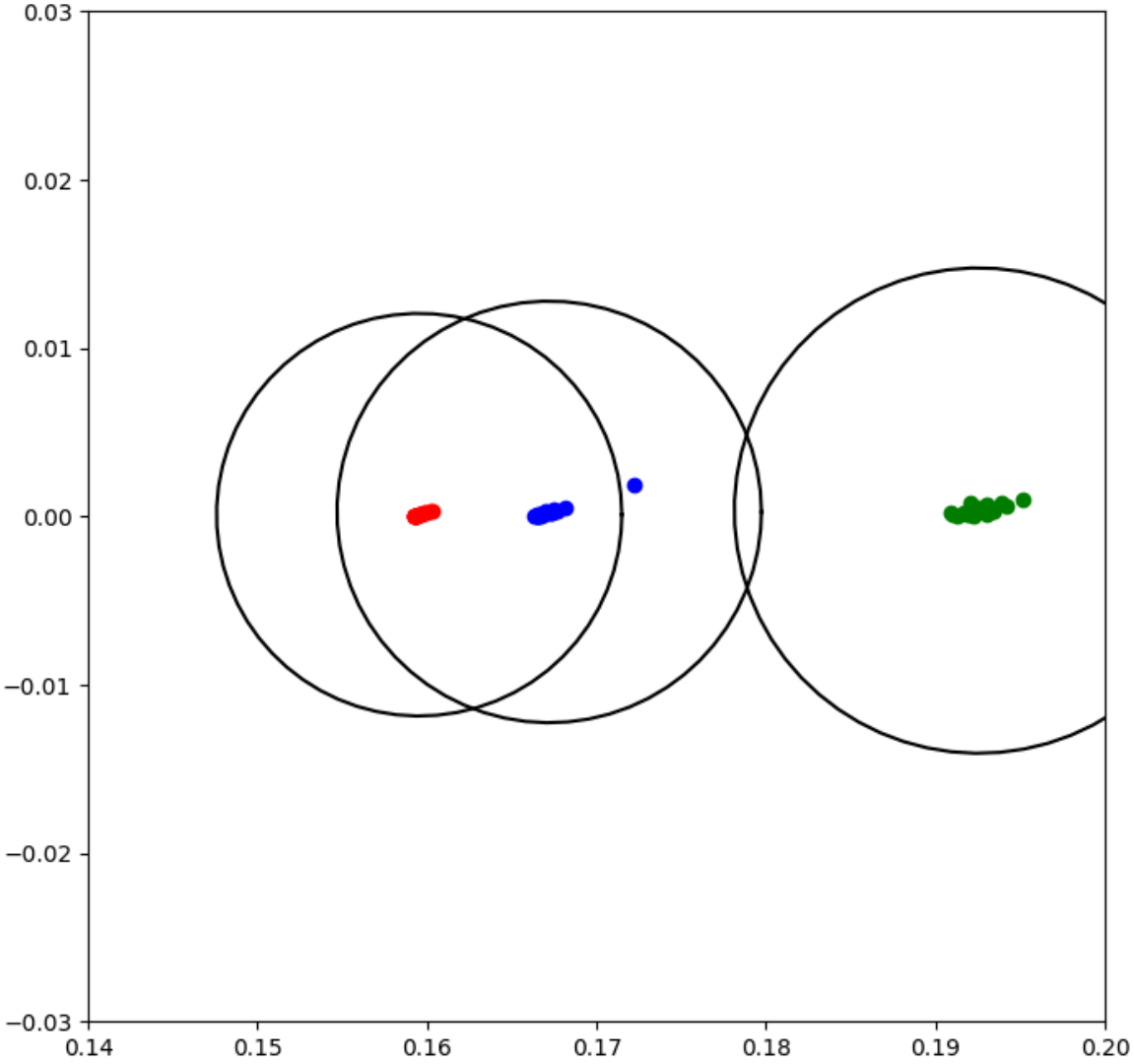
```
    else:
        print("The sign is a square\n")
```

In [14]:
```
# Read images
test_circle = cv2.imread(images_path + "test_circle.png", 0)
test_triangle = cv2.imread(images_path + "test_triangle.png", 0)
test_square = cv2.imread(images_path + "test_square.png", 0)

# Classify them
print("Circle: ")
classify_image_isotropic(test_circle)
print("Triangle: ")
classify_image_isotropic(test_triangle)
print("Square: ")
classify_image_isotropic(test_square)
```

```
Circle:
The sign is a circle

Triangle:
The sign is a triangle

Square:
The sign is a square
```

Complete the following code to observe the isotropic covariance matrices.

In [15]:
```
# Create figure
fig, ax = plt.subplots()
plt.axis([0.14, 0.2, -0.03, 0.03])

# Plot hu moments
plt.plot(train_triangles[0,:],train_triangles[1,:],'go')
plt.plot(train_circles[0,:],train_circles[1,:],'ro')
plt.plot(train_squares[0,:],train_squares[1,:],'bo')

# Plot ellipses representing covariance matrices
PlotEllipse(fig, ax, np.vstack(mean_triangles), 2500 * np.var(train_triangles) *
PlotEllipse(fig, ax, np.vstack(mean_circles), 2500 * np.var(train_circles) * np.
PlotEllipse(fig, ax, np.vstack(mean_squares), 2500 * np.var(train_squares) * np.

fig.canvas.draw()
```



Thinking about it (1)

Now that you are an expert concerning the Naïve Bayesian classifier, **answer the following questions**:

- Considering the classifier that you implemented in assignment 1c, to which class would be assigned an object with `x_1=0.17` and `x_2=-0.01`? And considering the one in assignment 2?

  Following the classifier in 1c, it would be assigned to triangle (green class), as the Mahalanobis distance between the object and said class is the smallest among the 3. That is, if we made the ellipses bigger, we would see that the point x = [0.17, -0.01] would be in that of the triangle class. Nevertheless, if we consider the classifier used in assignment 2, it would be assigned to square (blue class), as the Euclidean distance between the object and this class is the smallest of the 3. That is, looking at the circles, the point x = [0.17, -0.01] is in that of the square class.

- What are the pros and cons of using isotropic covariances?

  The pros would be their simplicity (thanks to the diagonal covariance matrix making Euclidean distance the new distance considered) and ease to classify objects visually in the graph. The cons would be that it considers features to be independent from

each other, which, in some scenarios, is not realistic nor logical to do so. For instance, if we approach a classification problem of fruits based on size and colour, there would be no problem as these features are clearly independent from each other. However, if we classify them based on size and weight, for example, then they would not be independent making the use of these covariances a likely mistake.

- In what type of problems could isotropic matrices be used?

  As I said, in those where features are independent from each other.

- In your opinion, is it worth to consider a Bayes classifier when dealing with this problem? In which situations could this classifier show its potential?

  Yes it is, if we consider Mahalanobis distance, as Hu moments are dependant of each other. This classifier has a big potential, so it can be used in a wide variety of situations, as long as you are aware of the dependency of features thus using the correct version.

In [16]:
```
# Code to prove the first question

# Object
x = np.array([0.17, -0.01])

# Equal prior probability
prior = 1/3

# Mahalanobis discriminant function
mdf_triangle = discriminant_function(x, mean_triangles, cov_triangles, prior)
mdf_circle = discriminant_function(x, mean_circles, cov_circles, prior)
mdf_square = discriminant_function(x, mean_squares, cov_squares, prior)

# Search the maximum
print("Classification according to Mahalanobis distance:\n")

classification = max([mdf_triangle,mdf_circle,mdf_square])

if classification == mdf_triangle:
    print("The object is a triangle\n")
elif classification == mdf_circle:
    print("The object is a circle\n")
else:
    print("The object is a square\n")

# Euclidean discriminant function
edf_triangle = discriminant_function_isotropic(x, mean_triangles)
edf_circle = discriminant_function_isotropic(x, mean_circles)
edf_square = discriminant_function_isotropic(x, mean_squares)

# Search the maximum
print("Classification according to Euclidean distance:\n")

classification = max([edf_triangle,edf_circle,edf_square])

if classification == edf_triangle:
    print("The object is a triangle\n")
elif classification == edf_circle:
    print("The object is a circle\n")
```

```
else:
    print("The object is a square\n")
```

```
Classification according to Mahalanobis distance:

The object is a triangle

Classification according to Euclidean distance:

The object is a square
```

Conclusion

Awesome! You now know how to design a classifier for previously segmented and characterized objects. Note that for more complex shapes, you can use the **7 Hu moments instead of the two that we used**. We reduced their number just for visualization and simplicity purposes.

In this notebook you have learned to:

- construct a Naïve Bayesian classifier and apply it to a real problem where features follow a normal distribution,
- build a simplified classifier where isotropic covariances are assumed, and
- improve a classifier (if needed) using rejection regions.

7.3 Naive Bayes Classifier based on the Binomial distribution

In the previous notebook we built discriminant functions that carried out a Maximum Log-Likelihood Estimation (MLE). Said functions were defined as $d_k(x) = ln\ p(\mathbf{x}|C_k)$, where the vector of features $\mathbf{x}$ followed a normal distribution. In this notebook we are going to explore the case in $\mathbf{x}$ follows a **a binomial distribuion**, where the features in $\mathbf{x}$ are either 0 or 1 (Bernoulli trial).

Recall that we are assuming again the **naive** Bayes classifier model, so features are considered **independent**:

$$P(x_i \cap x_j) = P(x_i)p(x_j)\ \forall i,j$$

In this notebook we will learn how a naive Bayes classifier can be trained (section 7.3.1) and tested (section 7.3.2) using binomial features.

Problem contex - Digit recognition

Digit recognition systems aim to recognize the digits appearing in different sources like emails, bank cheques, papers, images, etc. They have many real-world applications as diverse as online handwriting recognition on papers, computers, or tablets, the recognition of license plate numbers in vehicles, the processing of bank cheques' amounts, etc.

For completing the plate detector and recognition system in which we worked on in previous chapters, we are going to learn how to construct a digit recognition system using a naive Bayes classifier based on the binomial distribution. This system could be generalized to recognize any possible character appearing in the place.

For this problem we need a set of digit images that will be used as training dataset for our classifier. For this application we would need to classify both digits and letters, however, to simplify this, we are going to use only digit images. In this way, the problem is reduced to 10 possible classes. You can find the provided images in `./images/train_binary/imagen{0-9}_{1-500}.png`, having 500 images for each digit.

Note that the provided images contain handwritten digits instead of plate ones, but the training and working principles are the same.

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (16.0, 8.0)

images_path = './images/'
```

7.3.1 Training a naive Bayes classifier (binomial distribution)

We will follow the same procedure as in the previous notebook, but adapted to the binomial distribution. First, we need to remember how the discriminant function of a Bayesian classifier is created:

[Math Processing Error]

In the context of our digit recognition problem, we are going to deal with images with $28x28$ pixels, where each pixel is a feature that can take the value 0 (black pixel) or 1 (white pixel).These images are rearranged as vectors with $28x28 = 784$ features:

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_f\ \ldots\ x_{784}]$$

This way, each image could be interpreted as a point in space with 784 dimensions. Fascinating!

Thus, considering the binomial distribution, the probability that a pixel x_f belongs to a class C_i is modeled as:

$$p(x_f|C_i) = (p_f^i)^{x_f} * (1 - p_f^i)^{(1-x_f)}$$

Being:

- p_f^i the probability that $x_f = 1$ if $\mathbf{x} \in C_i$, and
- $1 - p_f^i$ the probability that $x_f = 0$ if $\mathbf{x} \in C_i$.

Again, we are assuming independence among features (we are building a **naive Bayes classifier**), so the probability $p(\mathbf{x}|C_i)$ can be expressed as the multiplication of each individual probability $p(x_f|C_i)$, that is:

$$p(\mathbf{x}|C_i) = \Pi_{f=1}^{n} p(x_f|C_i) = \Pi_{f=1}^{n} (p_f^i)^{x_f} * (1 - p_f^i)^{(1-x_f)}$$

Applying logarithms:

$$ln\ p(\mathbf{x}|C_i) = \Sigma_{f=1}^{n} [x_f \cdot ln(p_f^i) + (1 - x_f) \cdot ln\ (1 - p_f^i)] =$$

$$= \Sigma_{f=1}^{n} x_f \cdot ln\ \frac{p_f^i}{1 - p_f^i} + \Sigma_{f=1}^{n} ln\ (1 - p_f^i)$$

Finally, for obtaining **the discriminant function** for a category C_i, we have to also consider the prior probability:

$$d_i(x) = ln\ p(\mathbf{x}|C_i) + ln\ P(C_i) = \underbrace{ln\ P(C_i) + \Sigma_{f=1}^{n} ln\ (1 - p_f^i)}_{w_{n+1}^i} + \Sigma_{f=1}^{n} x_f \cdot \underbrace{ln\ \frac{p_f^i}{1 - p_f^i}}_{w_f^i}$$

As we can see, **the function is lineal** (and ideal!).

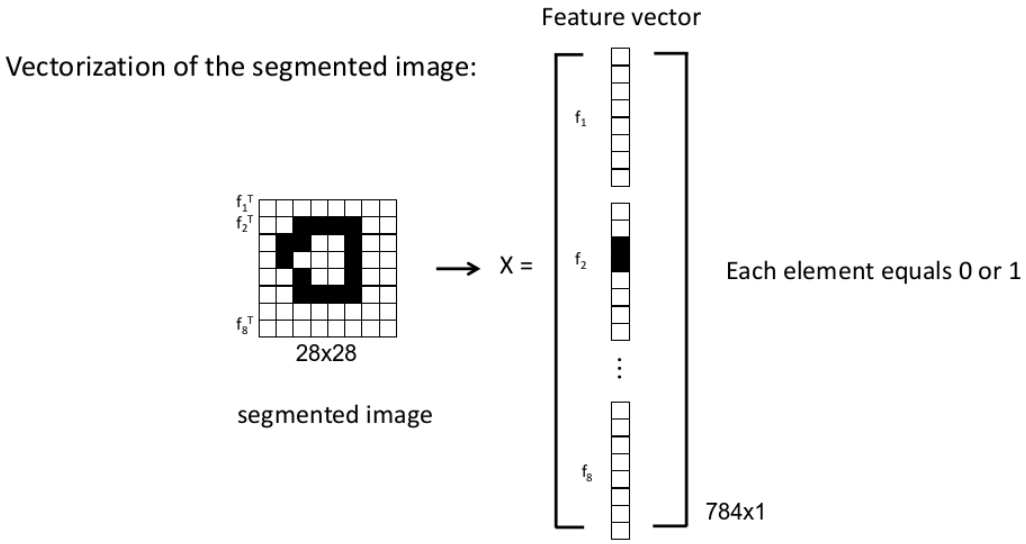
Implementation detail: it is recommended to avoid values of 0 or 1 for p_f^i since numerical problems may appear. Instead, take values close to 0 or 1. We will see this in a later assignment.

ASSIGNMENT 1a: Setting up the data

The first step for training the classifier is preparing the data (this is often called the preprocessing step). In the context of our problem, this is done by vectorizing each image.

Our task is to construct a matrix containing all the images' data, which can be found at: `./images/train_binary/imagen{0-9}_{1-500}.png`. This matrix will have 3 dimensions: $(n\_pixels, n\_images, n\_classes)$. Thus, each column of the matrix (second dimension) would contain a vectorized version of its corresponding image, having `n_images` in total. The third dimension indexes the different classes, having this problem a total of 10 (digits from 0 to 9).

The image below illustrates the vectorization process of an image:



In this way, **you have to** read all the images (there are 500 for each digit) and obtain the mentioned matrix storing all the data. Notice that the provided images are grayscale, so you will have to **binarize** them first using OpenCV, thus they can be used to build a binomial distribution.

In [2]:
```
## Training

# Initialize matrix
r,c = 28,28
dataset = np.zeros((r*c,500,10))

# For each class
for number in range(10):
    # For each image in the class
    for i in range(1,501):
        # Read the image
        path = images_path + "train_binary/imagen" + str(number) + "_" + str(i)
        image = cv2.imread(path,0)
        # Binarize it
        _,binarized = cv2.threshold(image,100,1,cv2.THRESH_BINARY)
        vector_binarized = binarized.reshape((-1,1))
        # Reshape it
        dataset[:,i-1,number] = vector_binarized[:,0]
```

ASSIGNMENT 1b: Computing probabilities

After the preprocessing step, we will compute the different p_f^i modeling the probability that $x_f = 1$ if $\mathbf{x} \in C_i$. These probabilities will allow us to compute the weights in a later

step. Recall how the decision functions are built:

$$d_i(x) = ln\ p(\mathbf{x}/C_i) + ln\ P(C_i) = \underbrace{ln\ P(C_i) + \Sigma_{f=1}^{n} ln\ (1 - p_f^i)}_{w_{n+1}^i} + \Sigma_{f=1}^{n} x_f \cdot \underbrace{ln\ \frac{p_f^i}{1 - p_f^i}}_{w_f^i}$$

Luckily, considering a binomial distribution, computing these probabilities is straightforward. We just have to count the number of times that each pixel (feature x_f) takes the value `1` in each class, and divide it by the number of images used (N):

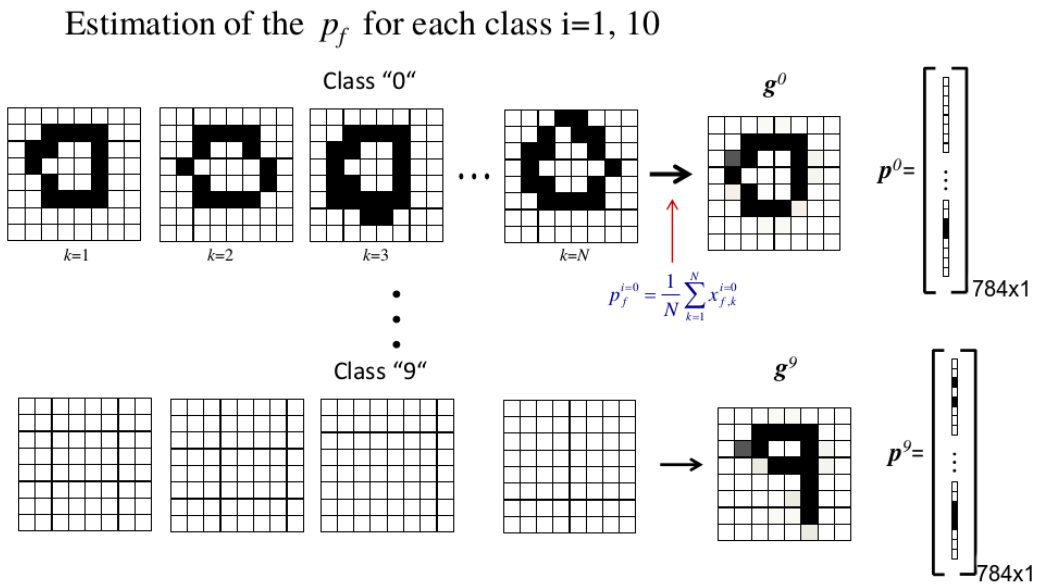
$$p_f^i = \frac{1}{N} \Sigma_{k=1}^{N} x_{f,k}^i$$

$\[3pt]$

Regarding our problem:

- $N = 500$ as we have 500 images for each class.
- The range of the index of features f is $[0 - 783]$ as images have $28x28 = 784$ pixels.

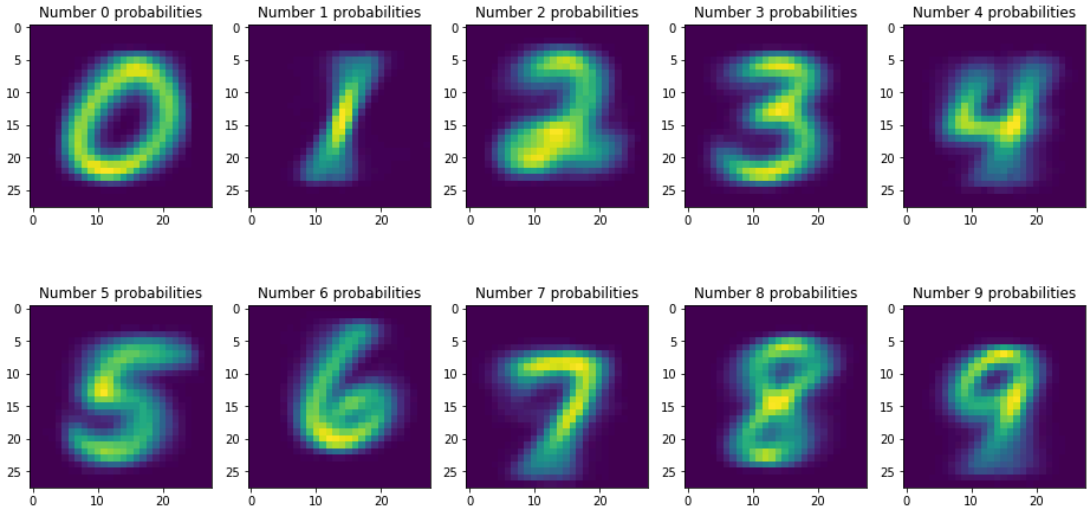
The following image illustrates the process of computing these probabilities:



What to do?

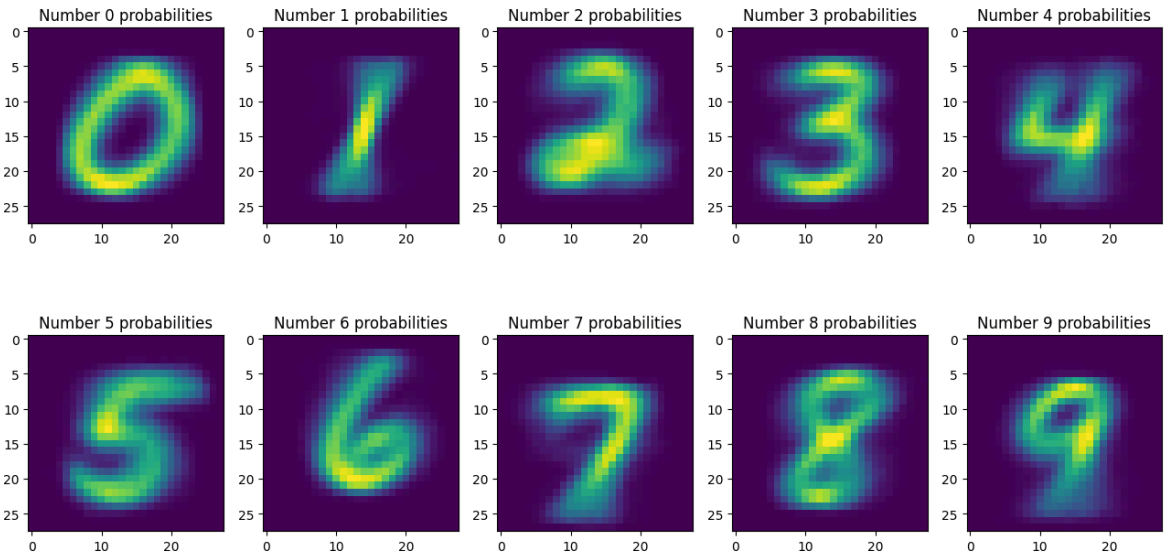
1. Construct a 2D matrix **containing all the probabilities** ($matrix[f, i] = p_f^i$).
2. Then, we are going to visualize the **prototype images**, also called **heatmaps**. There is a heatmap for each digit. In these images, the value of a pixel represent its probability of being `1` (black on a white paper sheet). For doing that, we have to **unvectorize the probabilities** in the matrix (2D $\rightarrow$ 3D) and **plot the heatmaps**. This is also a good way to check if we are doing well.

You should obtain something similar to this:



Tip: try to use `np.sum()` *for computing the probaiblities in order to avoid loops, which are usually slower (check the* `axis` *argument).* ***You can compute probabilities in just one code line!***

In [3]:
```
# Compute probabilities
probabilities = np.sum(dataset, axis=1)/500

# Reshape to obtain heatmaps
heatmaps = probabilities.reshape(r,c,10)

# Plot heatmaps
for i in range(2):
    for j in range(5):
        plt.subplot(2,5, i*5+(j+1))
        plt.title("Number " + str(i*5+j) + " probabilities")
        plt.imshow(heatmaps[:,:,i*5+j])
```



ASSIGNMENT 1c: Computing weights

Once we have retrieved the probabilities p_f^i, we are ready to obtain **the weights** that define the discriminant function for each class. This is the last training step! Recall that these weights are defined as:

$$w_f^i = ln\ \frac{p_f^i}{1 - p_f^i} \tag{5}$$

$$w_{n+1}^i = ln\ P(C_i) + \Sigma_{f=1}^{n} ln(1 - p_f^i) \tag{6}$$

Being:

- i: one of the 10 possible classes, and
- f: a feature in the range $[0, \ldots, 783]$.

What to do? Construct a 2D matrix containing the weights of the classifier. Consider the following:

- Since we can't neither divide by 0 nor compute the logarithm of 0, during the weights computation, **0 and 1 probabilities should be replaced by close numbers** (e.g. 0.0001 and 0.9999). This can be done using `np.where()`.
- As we don't have any prior information about the occurences of each class, the prior probability $P(C_i)$ should be $1/n\_classes$ for each class i.

Hint: For applying the natural logarithm to all the elements of a matrix, you can use `np.log()`.

In [4]:
```
## Weight calculation

# Replace 0 and 1 values
probabilities = np.where(probabilities==0, 0.001, probabilities)
probabilities = np.where(probabilities==1, 0.999, probabilities)

# Initialize matrix
weights = np.zeros((r*c+1,10))

# Compute weights
weights[:-1,:] = np.log(probabilities/(1-probabilities))
weights[-1,:] = np.log(0.1) + np.sum(np.log(1-probabilities),axis=0)
```

7.3.2 Testing the classifier

In the last step we retrieved the weights w_{n+1}^i and w_f^i, so now we have all the building blocks needed to design the discriminant function of each class $d_i(x)$ $(i = 0, 1, \ldots, 9)$. These decision functions permit us to evaluate a new vector $\mathbf{x}$ corresponding to a new vectorized image and retrieve its most probable category:

$$d_i(\mathbf{x}) = w_{n+1}^i + \Sigma_{f=0}^{783} w_f^i \cdot x_f \quad i = 0, \ldots, 9$$

In this way, each discriminant function, after evaluating the vector of features, will return a number. As we are estimating the **maximum log-likelihood**, the assigned class will be the one corresponding to the discriminant function returning the highest value.

ASSIGNMENT 2

For this exercise, it is provided a `numPy` matrix called `digitos.npy`, which is located at `./test_binary/`. It is a 1D matrix with 50 elements containing the ground truth information of 50 images, `./test_binary/timage{1-50}`. That is, the matrix codifies the digit, from 0 to 9, that each image contains.

Your task is to classify those 50 images and check if the output of your classifier matches with the actual class, as provided in `digitos.npy`. In this way, we can estimate how good is our classifier.

Note: If you trained your classifier with the provided images, you should expect 41/50 hits, or a 82% of accuracy.

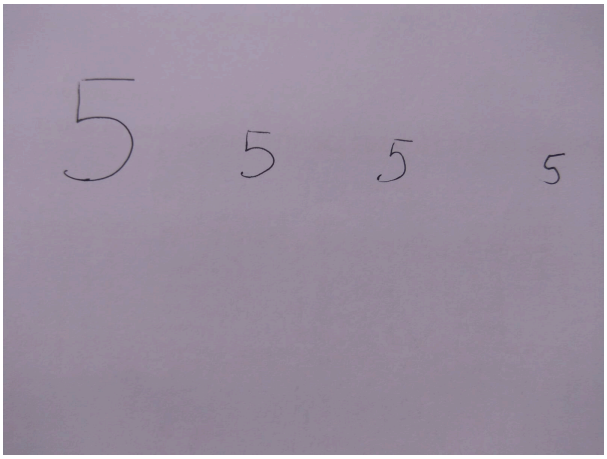
In [5]:
```
## Testing

# Load provided matrix
results = np.load(images_path + "test_binary/digitos.npy")
hits = 0

# Classify each testing image
for i in range(1,51):
    f_pesos = np.zeros((10))
    # Read the image
    path = images_path + "test_binary/timage" + str(i) + ".png"
    image = cv2.imread(path,0)
    # Binarize it
    _,binarized = cv2.threshold(image,100,1,cv2.THRESH_BINARY)
    # Vectorize it
    vector_binarized = binarized.reshape((-1,1))
    vector_binarized = np.append(vector_binarized,[1])
    # Evaluate the discriminant functions
    for j in range(10):
        f_pesos[j] = np.sum(weights[:,j] * vector_binarized)
    # Get the maximum
    res = np.where(f_pesos == np.amax(f_pesos))
    if res == results[i-1]:
        hits = hits + 1

# Print the accuracy
print("Accuracy=" + str(hits) + "/50 (" + str(100.0*hits/50.0) + "%)" )
```

```
Accuracy=41/50 (82.0%)
```

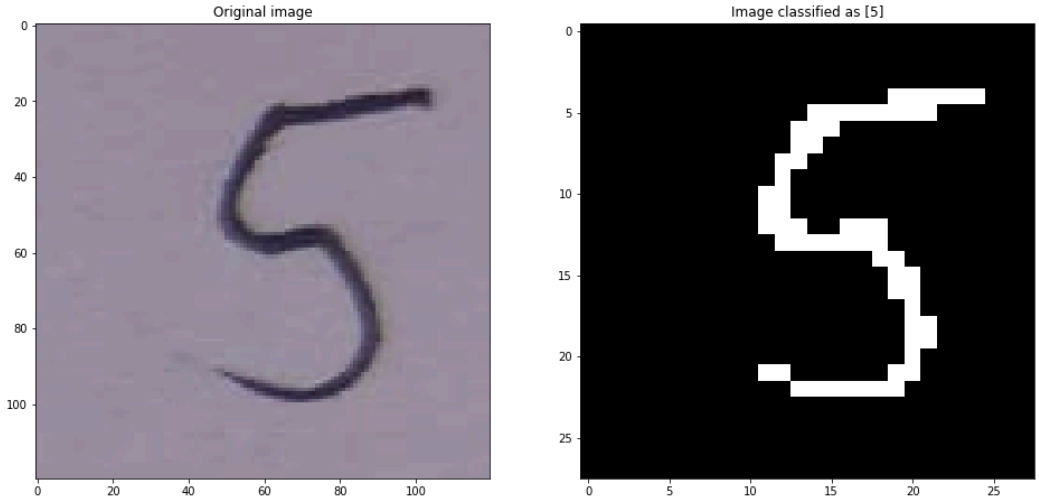
EXTRA: ASSIGNMENT 3

Do you dare to try your brand new classifier with your own images? Brilliant! For that:

1. Write a digit in a paper and make a photo with your phone. The following image shows an example (you can try with different numbers, of course!).



2. Then, segment the region of interest (crop the digit, see the left image in the figure below).
3. Resize the image to 28×28 pixels (this is the input image size of our classifier). You can use `cv2.resize()` for that.
4. Finally, binarize (see the right image in the figure below) and vectorize the image.

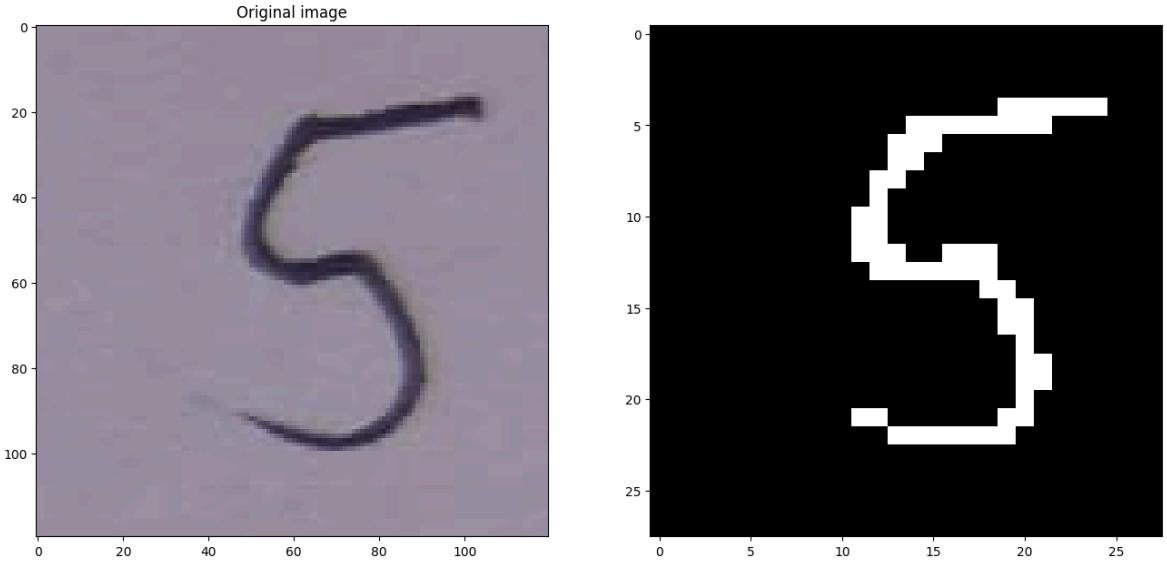


5. Follow the steps done in *assignment 2* to classify the image and show the results!

In [6]:
```
## Testing with our own image

# Read image and convert it to grayscale
color_image = cv2.imread(images_path + 'testcamera.jpeg', -1)
color_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB)

# Crop ROI
color_image = color_image[380:500,1390:1510,:]
image = cv2.cvtColor(color_image, cv2.COLOR_RGB2GRAY)

# Binarize it
image = cv2.resize(image,(r,c))
_,binarized = cv2.threshold(image,130,1,cv2.THRESH_BINARY_INV)

# Reshape the image
vector_binarized = binarized.reshape((-1,1))
vector_binarized = np.append(vector_binarized,[1])

# Evaluate the discriminant functions
for j in range(10):
    f_pesos[j] = np.sum(weights[:,j] * vector_binarized)
```

```
# Get the maximum
res = np.where(f_pesos == np.amax(f_pesos))

# Plot the results
plt.subplot(121)
plt.imshow(color_image)
plt.title("Original image")
plt.subplot(122)
plt.imshow(binarized,cmap="gray")
print("Image classified as " + str(res[0]))
```

```
Image classified as [5]
```



Conclusion

Excellent! You have learned how to:

- design and train a naïve Bayes classifier based on the binomial distribution for binary images,
- test and obtain the accuracy of the classifier,
- recognize digits in the context of plate characters (notice that the handwritten digits dataset used in the assignments could be replaced by any other containing plate characters!).

7.5 Deep Learning methods for object detection

In the previous notebooks we have developed different **Machine Learning** (ML) algorithms for classifying images containing objects according to high-level features (shape, color, texture, etc.). In a nutshell, these methods use a number of representatitive images to train a classifier, which is composed by a set of parameters. Once trained, the model can be used to infer the category of objects appearing in new images not seen before.

As you may now, in the recent years there is an explosion of **Deep Learning** (DL) techniques that are achieving a high performance in different tasks. Indeed, DL models are a subgroup of ML with the peculiarity of having thousands, millions of parameters. This is achieved by using multiple layers, each one consisting of a given number of parameters, hence the *deep* of DL.

The choice of ML or DL techniques strongly depend on the application at hand. On the one hand, if there is available a large amount of data to train the model and the target platform where the model will run is powerful enough, then DL techniques are a good option. On the other hand, when data or computational resources' constrains exist, ML methods stands out. Moreover, while ML works with explainable high-level features and processes, DL employs low-level features and are considered black boxes with an input (e.g. an image) and an ouput (e.g. a set of detected objects).

As commented, untill now we have been working with ML models. Let's take a look at DL ones!

Problem context - Object detection for mobile robots

This time we will focus on mobile robots operaing in human-centered environments, for example houses. In order to provide services, a mobile robot must be able to understand its surroundings, including the elements in its workspace. For that object detection is a critital task since it permits the robot to locate objects that can be useful in its duties. Fig 1 provides some examples of objects detected in images coming from a comera mounted on a mobile robot:

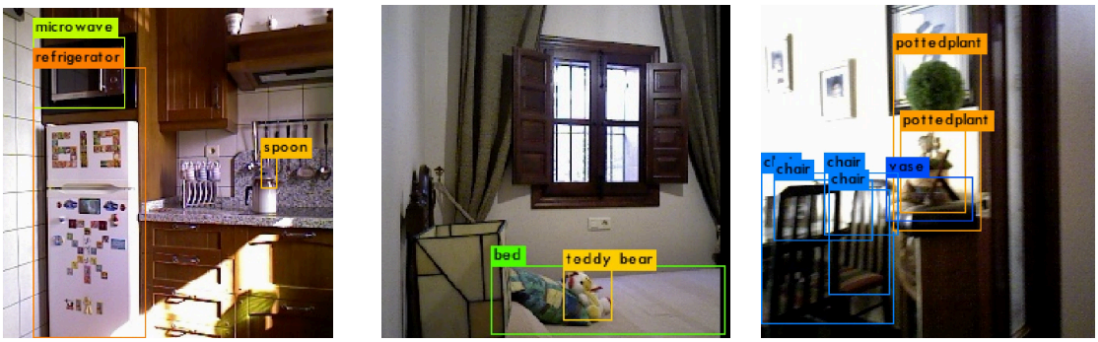


Figure 1. Images processed by a DL model with annotations about detected objects.

Fusing these detections with additional information like the robot pose within the workspace and the relative pose of the camera w.r.t., the objects can be localized in said workspace. For example, Fig. 2 shows the reconstruction of two houses where the robot has detected and placed a number of objects.

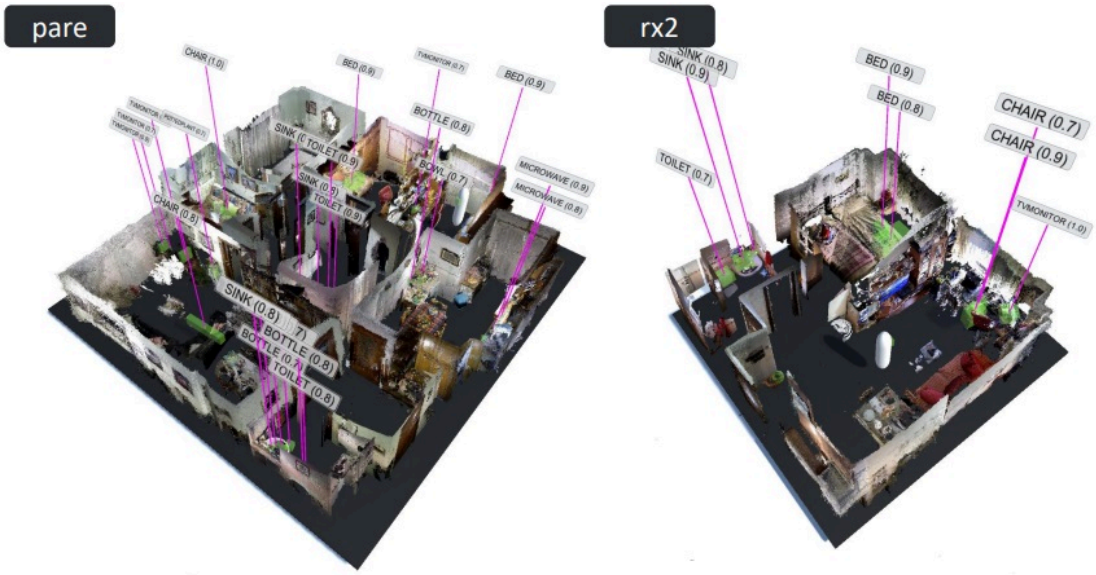


Fig 2. Reconstructions of real houses built by a mobile robot.

In [1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

7.5.1 Preparing the DL model

The task of training a DL model is highly time and resources consuming, typically demanding a cluster of GPUs and huge datasets. Luckily for us, there is a bunch of already trained models freely available that we can use. And what is more, our lovely OpenCV incorporates the **DNN module** (Deep Neural Network), with permits us to run **DL inference**, that is, to feed a network with an image and get the results of its processing. This is quite interesting since we don't need to install other heavy frameworks like Tensorflow, Pytorch, Caffe or Keras often used to deal with these models. Moreover, it's not necessary to have a powerful GPU to make inference since OpenCV DNN is highly optimized to run in CPU. Good news everywhere!

Since our problem is object detection, we will make use of a network trained for that aim. Concretely wee need two resources:

- a file containing the configuration of the model to be used, and
- another file with the weights (parameters) of said network.

Depending on the dataset used to train the network, the categories that it would be able to detect differ. In this notebook we are going to employ a popular network trained with data from Micrososf COCO. This is the most used dataset for dealing with object detection related-tasks and considers 80 different categories:

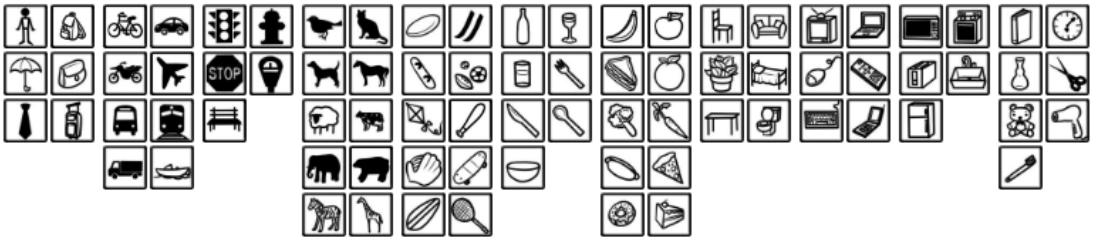


Figure 3. Object categories considered in the COCO dataset. [Source: COCO webpage]

The following figure shows an example of an image from COCO, which also includes annotations about the objects that belong to one of the said categories:

Figure 4. And image of a kitchen from COCO with annotation about the regions belonging to one of the considered categories. [Source: COCO webpage]

The COCO dataset has a very nice webpage with many resources, including an explorer where you can see examples like the previous one. We encorage you to take a look at it! Having choosen the network and knowing the dataset it was trained with, its time to load the needed information.

Let's start by loading a file containing the names of the categories from COCO. This is useful since the network will provide the detected objects as numbers related to the indices that the categories has in this file. The following code reads this file and also builds a numpy array that assigns to each category a random color (used for results visualization).

In [2]:
```
# load the COCO class names
with open('./data/object_detection_classes_coco.txt', 'r') as f:
    class_names = f.read().split('\n')

# get a different color array for each of the classes
COLORS = np.random.uniform(0, 255, size=(len(class_names), 3))
type(COLORS)
```

Out[2]:
```
numpy.ndarray
```

ASSIGNMENT 1: Exploring categories

Let's explore a bit the loaded information. Write the needed code to print the categories appearing from index 40 to index 60 with the format shown in the expected output. You can also try other ranges!

In [3]:
```
# Assignment 1
for index in range(40, 61):
    print(f"[ {index} ] {class_names[index]}")
```

```
[ 40 ] skateboard
[ 41 ] surfboard
[ 42 ] tennis racket
[ 43 ] bottle
[ 44 ] plate
[ 45 ] wine glass
[ 46 ] cup
[ 47 ] fork
[ 48 ] knife
[ 49 ] spoon
[ 50 ] bowl
[ 51 ] banana
[ 52 ] apple
[ 53 ] sandwich
[ 54 ] orange
[ 55 ] broccoli
[ 56 ] carrot
[ 57 ] hot dog
[ 58 ] pizza
[ 59 ] donut
[ 60 ] cake
```

Expected output

```
[ 40 ]  skateboard
[ 41 ]  surfboard
[ 42 ]  tennis racket
[ 43 ]  bottle
[ 44 ]  plate
[ 45 ]  wine glass
```

```
[ 46 ]  cup
[ 47 ]  fork
[ 48 ]  knife
[ 49 ]  spoon
[ 50 ]  bowl
[ 51 ]  banana
[ 52 ]  apple
[ 53 ]  sandwich
[ 54 ]  orange
[ 55 ]  broccoli
[ 56 ]  carrot
[ 57 ]  hot dog
[ 58 ]  pizza
[ 59 ]  donut
[ 60 ]  cake
```

Once the dataset-related information is loaded, let's do the same with the network configuration and its file of weights. We are going to give a try to SSD Mobile net, but this is just one of the possibilities supported by OpenCV DNN. Here you can find a list with all the networks that have been tested by the OpenCV community. If you want to try another one, this file contains the url to download the models' weights, while this repository provides the networks' configuration files.

SSD comes from Single Shot MultiBox Detector, and it is a popular network presented at the popular European Conference on Computer Vision (ECCV) in 2016[1]. This network consists of two parts:

- A backbone (VGG-16 in the image, but in this case MobileNet is employed) used to extract feature maps from the image and a number of extra feature layers to detect objects at different scales, and
- a number of convolutional layers that carry out the object detections (shown over the lines with the prefix *Classifier*).

The network architecture is depicted in Fig. 5.

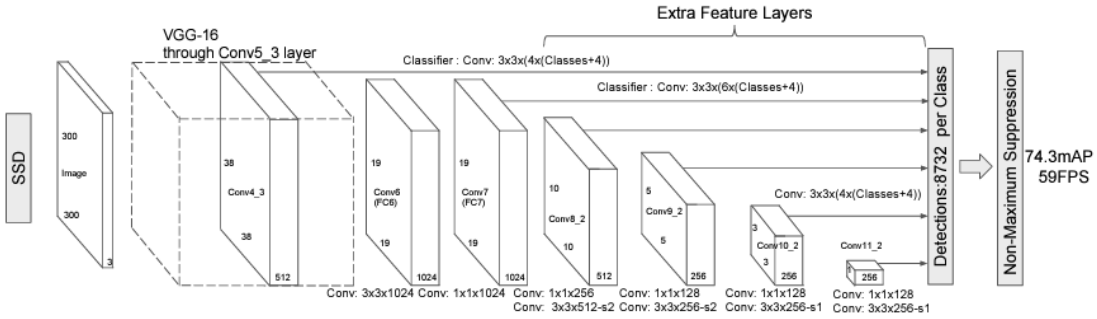


Figure 5. Architecture of the SSD network [Source SSD paper].

Let's load this model.

In [4]:
```
# Load the DNN model
model = cv2.dnn.readNet(model='./data/frozen_inference_graph.pb',
                        config='./data/ssd_mobilenet_v2_coco_2018_03_29.pbtxt',f
```

7.5.2 Doing inference and showing results

The following code loads an image from file, gets its height and width, and set it as input to the model.

In [5]:
```
# Read the image from disk
image = cv2.imread('./images/dnn/kitchen_2.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_height, image_width, _ = image.shape
# Create blob from image
blob = cv2.dnn.blobFromImage(image=image, size=(300, 300), mean=(104, 117, 123),
# Set the blob to the model
model.setInput(blob)
```

Once the information related to the dataset and the network are loaded, as well as an image to be processed, we can perform inference by just calling the `forward()` function.

In [6]:
```
# Forward pass through the model to carry out the detection
output = model.forward()
```

ASSIGNMENT 2: Showing the results

The network output is a set of predictions consisting of object categories, scores and bounding boxes, concretely: `prediction=[not_used,category,score,bbox_x_ul,bbox_y_ul,bbox_x_br,bbox_y_br]`

For example, the next output:

```
[ 0.         52.          0.9821959   0.3924626   0.60919476
0.85021377  0.9348804 ]
```

says that it has been detected an apple with a very high score (it ranges from 0 to 1) with a bounding box with the upper left corner at position $[x_{ul}, y_{ul}] = [0.3924626 \times img\_width, 0.609194 \times img\_height]$ and with the bottom right corner at position $[x_{br}, y_{br}] = [0.850213 \times img\_width, 0.93488 \times img\_height]$.

You are tasked to:

- Get the confidence (score) value from the `detection` vector, and set a threshold so only detections with a confidence value higher than it are shown (try different values until you get a result similar to the one shown in Fig. 6),
- get the boundig boxes positions and sizes as commented before,
- show the rectangles representing the bounding boxes, and
- build the `text` string containing the class names and confidence values as shown in Fig. 6.

Figure 6. Illustration of an image processed by the network with annotations of the detected objects, their categories and scores.

In [7]:
```
# loop over each of the detection
image_copy = image.copy()

for detection in output[0, 0, :, :]:

    # extract the confidence of the detection
    confidence = detection[2]

    # draw bounding boxes only if the detection confidence is above...
    # ... a certain threshold, else skip
    if confidence > 0.3:

        # get the class id
        class_id = detection[1]
        # map the class id to the class
        class_name = class_names[int(class_id)-1]
        color = COLORS[int(class_id)]
        # get the bounding box upper left corner coordinates
        box_x_ul = detection[3] * image_width
        box_y_ul = detection[4] * image_height
        # get the bounding box bottom right corner coordinates
        box_x_br = detection[5] * image_width
        box_y_br = detection[6] * image_height
        # draw a rectangle around each detected object
        cv2.rectangle(image_copy, (int(box_x_ul), int(box_y_ul)), (int(box_x_br)
        # put the FPS text on top of the frame
        text = class_name + f"({confidence:.2f})"
        cv2.putText(image_copy, text, (int(box_x_ul+2), int(box_y_ul + 30)), cv2

# Show image_copy with the detected objects
plt.imshow(image_copy);
plt.axis('off');
```

With this type of models onboard of a mobile robot, it would be able to detect the objects in its sourroundings and use this information during its operation. Awesome!

Conclusion

Cool! You have experienced the trendy Deep Learning world a bit.

In this notebook you have learned:

- how look for different DL models that has been tested by the OpenCV community,
- how to load information related to the dataset at hand, as well as how to load the network weights (model) and configuration,
- how to do inference and visualize the results.

Optional

Now that you know how to use OpenCV DNN you can challenge the model with other images or, even better, try another object detection network or a model targeted at a different task.

References

[1]: LIU, Wei, et al. Ssd: Single shot multibox detector. In European Conference on Computer Vision (ECCV). Springer, Cham, 2016. p. 21-37.