

# Apprentissage de paramètres par maximum de vraisemblance

Dans ce TME, l'objectif est d'apprendre grâce à l'estimateur de maximum de vraisemblance les paramètres de lois normales à partir d'un ensemble de données. Ces lois normales seront ensuite exploitées pour faire de la classification (comme nous l'avions vu en cours avec les images de désert, forêt, mer et paysages enneigés).

Ici, notre base de données d'apprentissage est la base USPS. Celle-ci contient les images réelles de chiffres provenant de codes postaux écrits manuellement et scannés par le service des postes américain. Ces données scannées ont été normalisées de manière à ce qu'elles soient toutes des images de 16x16 pixels en teintes de gris, cf. Le Cun et al., 1990:

Y. LeCun, O. Matan, B. Boser, J. S. Denker, et al. (1990) *Handwritten zip code recognition with multilayer networks*. In ICPR, volume II, pages 35–40.

Voici quelques exemples d'images de cette base :

 Quelques exemples

In [33]:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle as pkl
import math as mt
```

## Chargement des données et premières visualisations

Nous utiliserons la librairie pickle qui permet de sérialiser les objets en python (ie, les sauver et les charger très facilement). Une fois les données chargées, nous allons étudier très rapidement la distribution des classes, visualiser une imagerie de chiffre et comprendre l'encodage de ces chiffres.

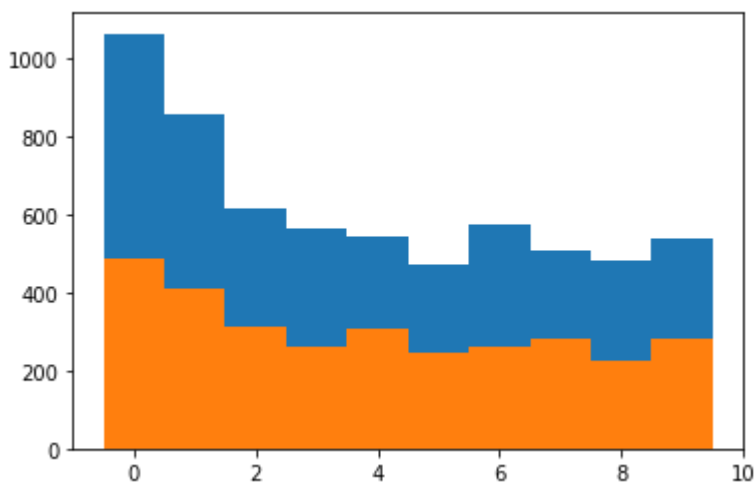
In [34]:

```
# Chargement des données
data = pickle.load(open("usps.pkl", 'rb'))
# data est un dictionnaire contenant les champs explicites X_train, X_test, Y_train, Y_test
X_train = np.array(data["X_train"], dtype=float) # changement de type pour éviter les problèmes d'affichage
X_test = np.array(data["X_test"], dtype=float)
Y_train = data["Y_train"]
Y_test = data["Y_test"]

# visualisation de la distribution des étiquettes (dans les 10 classes de chiffres)
plt.figure()
plt.hist(Y_train, np.linspace(-0.5, 9.5, 11))
plt.hist(Y_test, np.linspace(-0.5, 9.5, 11))
plt.savefig("distr_classes.png")
```

Out[34]:

```
(array([488., 412., 311., 260., 306., 244., 261., 282., 224., 281.]),
 array([-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5]),
 <a list of 10 Patch objects>)
```



In [35]:

```
# prise en main des matrices X, Y
print(X_train.shape)
# 6229 images composées de 256 pixels (image = 16x16)
print(X_test.shape, Y_train.shape, Y_test.shape)

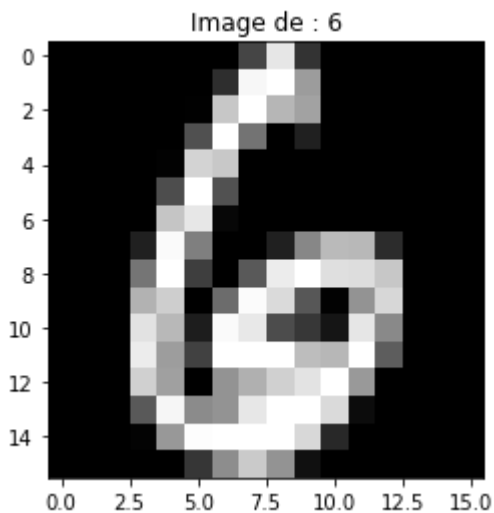
# Affichage de l'image 18 de la base de données et récupération de l'étiquette a
ssociée:
# (1) remise en forme de la ligne de 256 pixels en 16x16
# (2) affichage avec imshow (en niveaux de gris)
# (3) récupération de l'étiquette dans Y_train

plt.figure()
plt.imshow(X_train[18].reshape(16,16), cmap="gray")
plt.title("Image de : {}".format(Y_train[18]))
```

```
(6229, 256)
(3069, 256) (6229,) (3069,)
```

Out[35]:

```
Text(0.5, 1.0, 'Image de : 6')
```



In [36]:

```
# analyse des valeurs min et max, recherche du nombre de niveaux de gris dans le
s images:
print(X_train.min(), X_train.max() )
print("niveaux de gris : ", len(np.unique(X_train)))
```

```
0.0 2.0
niveaux de gris : 2001
```

# A. Apprentissage et évaluation d'un modèle gaussien naïf

## A1- Maximum de vraisemblance

Nous allons étudier la distribution de probabilité des teintes de gris des images (en fait, nous allons étudier sa fonction de densité car on travaille sur des variables aléatoires continues) . Nous allons faire l'hypothèse (certes un peu forte mais tellement pratique) que, dans chaque classe, les teintes des pixels sont mutuellement indépendantes.

Autrement dit, si  $X_i, i \in \{0, \dots, 255\}$  représente la variable aléatoire "intensité de gris du ième pixel", alors  $p(X_0, \dots, X_{255})$  représente la fonction de densité des teintes de gris des images de la classe et:

$$p(X_0, \dots, X_{255}) = \prod_{i=0}^{255} p(X_i)$$

Ainsi, en choisissant au hasard une image dans l'ensemble de toutes les images possibles de la classe, si celle-ci correspond au tableau `np.array([x_0, ..., x_255])` , où les  $x_i$  sont des nombres réels compris entre 0 et 2, alors la valeur de la fonction de densité de l'image est égale à

$$p(x_0, \dots, x_{255}) = \prod_{i=0}^{255} p(x_i).$$

Nous allons de plus supposer que chaque  $X_i$  suit une distribution normale de paramètres  $(\mu_i, \sigma_i^2)$ . Autrement dit,

$$\forall i \in \{0, \dots, 255\}, X_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$$

Par maximum de vraisemblance, estimez, pour une classe donnée, l'ensemble des paramètres  $(\mu_0, \dots, \mu_{255})$  et  $(\sigma_0^2, \dots, \sigma_{255}^2)$  pour chaque classe (chiffre de 0 à 9). Pour cela, écrivez une fonction `learnML_parameters : float np.array x float np.array -> float np.array x float np.array` qui, étant donné le tableau d'images , renvoie un couple de tableaux, le premier élément du couple correspondant à l'ensemble des  $\mu_i$  et le 2ème à l'ensemble des  $\sigma_i^2$ . C'est-à-dire que

`learnML_parameters` renverra deux matrices:

$$mu \in \mathbb{R}^{10 \times 256}, sig \in \mathbb{R}^{10 \times 256}$$

- `mu` contient les moyennes des 256 pixels pour les 10 classes
- `std` contient les écarts-types des 256 pixels pour les 10 classes

In [37]:

```
def learnML_parameters(X,Y):
    # votre code
    mu=np.zeros((10, X.shape[1]))
    sig=np.zeros((10, X.shape[1]))
    for i in range(10):
        class_indices=np.where(Y==i)
        X_filtered=X[class_indices,:]
        mu[i, :]=np.mean(X_filtered[0],0 )
        sig[i,:]=np.std(X_filtered[0],0)
    return mu, sig
pass
```

In [38]:

```
mu,sig = learnML_parameters ( X_train, Y_train )  
print(mu.shape, sig.shape) # doit donner (10, 256) (10, 256)
```

(10, 256) (10, 256)

Check: pour la classe 0, les paramètres doivent être les suivants

```
mu[0]=  
[1.53774208e-03 4.46785940e-03 1.71216078e-02 6.31194048e-02  
1.84061642e-01 4.71391665e-01 8.97640989e-01 1.15019928e+00  
...  
1.42675380e+00 1.03130694e+00 5.32240296e-01 1.74166387e-01  
3.57644515e-02 5.52804884e-03 4.36592998e-04 0.00000000e+00]  
sig[0]=  
[5.01596286e-02 7.93695089e-02 1.46489017e-01 2.65522337e-01  
4.42306204e-01 6.35148001e-01 7.40462105e-01 7.48387032e-01  
...  
6.62741331e-01 6.75677391e-01 5.86224763e-01 3.56460503e-01  
1.71512333e-01 5.67475697e-02 1.20193571e-02 0.00000000e+00]
```

In [39]:

```
print(mu[0], sig[0])
```

[1.53774208e-03 4.46785940e-03 1.71216078e-02 6.31194048e-02  
1.84061642e-01 4.71391665e-01 8.97640989e-01 1.15019928e+00  
1.02070900e+00 6.16785408e-01 2.50766353e-01 8.09903122e-02  
1.49310824e-02 3.87033274e-03 1.70898437e-04 0.00000000e+00  
2.35601434e-03 7.93762565e-03 5.18573940e-02 2.00940178e-01  
5.59411980e-01 1.10202446e+00 1.53532559e+00 1.66378367e+00  
1.60174400e+00 1.37845195e+00 9.19353768e-01 4.15479248e-01  
1.22346858e-01 3.01527050e-02 4.24372534e-03 5.63428995e-06  
5.88210737e-03 3.00679919e-02 1.45009354e-01 4.68428296e-01  
9.87103163e-01 1.46020945e+00 1.63103905e+00 1.59445846e+00  
1.52785712e+00 1.52155705e+00 1.36630499e+00 9.33882722e-01  
4.29464169e-01 1.16865928e-01 2.02650169e-02 1.02089895e-03  
7.91202062e-03 6.88018163e-02 3.30783411e-01 8.24194929e-01  
1.31254975e+00 1.54256605e+00 1.44943446e+00 1.21191395e+00  
1.09220056e+00 1.22655949e+00 1.39070742e+00 1.27821441e+00  
8.46682745e-01 3.48925595e-01 8.07226924e-02 4.64439750e-03  
1.59532081e-02 1.76675445e-01 6.07781794e-01 1.12757141e+00  
1.46523519e+00 1.45472469e+00 1.13856317e+00 8.10338934e-01  
6.26168286e-01 7.76955398e-01 1.10779727e+00 1.32851374e+00  
1.16676063e+00 6.86548686e-01 2.41357803e-01 2.81289553e-02  
4.81442913e-02 3.73752078e-01 8.81483647e-01 1.31949916e+00  
1.46700523e+00 1.23718114e+00 8.16962582e-01 4.43474916e-01  
2.80405642e-01 4.08021647e-01 7.81619032e-01 1.17122113e+00  
1.27515733e+00 9.72332353e-01 4.74029267e-01 8.55221959e-02  
1.30783142e-01 5.93982138e-01 1.07686306e+00 1.39894059e+00  
1.36844099e+00 9.74245539e-01 5.33412071e-01 2.15832907e-01  
1.06282689e-01 1.88639261e-01 5.18731675e-01 9.65949319e-01  
1.24726249e+00 1.12132596e+00 6.89466858e-01 1.96154590e-01  
2.50674651e-01 7.61315399e-01 1.19695898e+00 1.38864862e+00  
1.19971341e+00 7.48517123e-01 3.30572606e-01 9.84733976e-02  
4.29241342e-02 1.03367730e-01 3.87229277e-01 8.44056198e-01  
1.20012881e+00 1.18425086e+00 8.20222630e-01 3.08674699e-01  
3.58261634e-01 8.62190202e-01 1.25563738e+00 1.34673373e+00  
1.07142879e+00 5.89123140e-01 1.96728277e-01 4.28680832e-02  
1.60351829e-02 7.26840328e-02 3.45407532e-01 8.19457636e-01  
1.19289312e+00 1.21496732e+00 8.84999523e-01 3.79906737e-01  
3.87331813e-01 9.06482830e-01 1.29035945e+00 1.31940006e+00  
9.78757769e-01 4.76966465e-01 1.27168587e-01 2.11923134e-02  
1.44063591e-02 8.57732477e-02 3.76648466e-01 8.87929238e-01  
1.23895217e+00 1.25035519e+00 8.86075297e-01 3.57954944e-01  
3.26138273e-01 8.93522235e-01 1.31510628e+00 1.35238794e+00  
9.61461921e-01 4.39577541e-01 1.13927428e-01 3.84975505e-02  
5.36327846e-02 1.69787235e-01 5.42599817e-01 1.06182197e+00  
1.37321399e+00 1.25057057e+00 8.18406790e-01 2.61721288e-01  
1.82330468e-01 7.59475951e-01 1.28679207e+00 1.44958046e+00  
1.10688967e+00 5.57203861e-01 2.14130381e-01 1.47069378e-01  
2.15676089e-01 4.32815974e-01 8.96862446e-01 1.36377634e+00  
1.49075578e+00 1.15939639e+00 6.22067020e-01 1.26992220e-01  
5.84152687e-02 4.67427550e-01 1.09806340e+00 1.51210615e+00  
1.42226714e+00 9.41197788e-01 5.57709665e-01 4.99719957e-01  
6.56042283e-01 1.00399766e+00 1.40647827e+00 1.61266601e+00  
1.41656614e+00 8.81148097e-01 3.21861387e-01 3.67738419e-02  
1.09871555e-02 1.48898601e-01 6.49790437e-01 1.29661503e+00  
1.64543706e+00 1.56182002e+00 1.31738978e+00 1.25782812e+00  
1.39819160e+00 1.60791875e+00 1.72751589e+00 1.54867872e+00  
1.01832667e+00 4.19518357e-01 9.27781289e-02 7.52478765e-03  
2.67797390e-03 2.13532309e-02 1.63836169e-01 6.27896414e-01  
1.26470542e+00 1.69852879e+00 1.80189263e+00 1.79891885e+00  
1.80963883e+00 1.76736086e+00 1.51130342e+00 9.40219832e-01  
3.74622653e-01 8.96866275e-02 1.24228554e-02 1.88855453e-03  
7.41913845e-05 1.44583295e-03 1.03138830e-02 7.97420072e-02

3.19437704e-01 7.96441981e-01 1.28306524e+00 1.52049088e+00  
1.42675380e+00 1.03130694e+00 5.32240296e-01 1.74166387e-01  
3.57644515e-02 5.52804884e-03 4.36592998e-04 0.00000000e+00] [5.015  
96286e-02 7.93695089e-02 1.46489017e-01 2.65522337e-01  
4.42306204e-01 6.35148001e-01 7.40462105e-01 7.48387032e-01  
7.52036960e-01 6.78162781e-01 4.81813622e-01 2.95993695e-01  
1.02298252e-01 6.82956225e-02 3.69197145e-03 0.00000000e+00  
5.53305148e-02 9.36063684e-02 2.40012769e-01 4.83422217e-01  
7.27502091e-01 8.01985000e-01 6.94079513e-01 6.09677496e-01  
6.52843815e-01 7.58749067e-01 8.06674659e-01 6.41779078e-01  
3.75508554e-01 1.82421873e-01 5.60909703e-02 1.83784976e-04  
9.07390154e-02 1.92883673e-01 4.16989953e-01 7.07690582e-01  
8.35158840e-01 7.30403526e-01 6.28008151e-01 6.32851408e-01  
6.86100539e-01 6.71979869e-01 7.51457414e-01 8.24431822e-01  
6.63627535e-01 3.74208629e-01 1.38194178e-01 2.98619157e-02  
9.45533790e-02 2.82750058e-01 6.06158973e-01 8.42184682e-01  
8.02525637e-01 6.80684430e-01 7.48069966e-01 8.29618901e-01  
8.54595560e-01 8.13916791e-01 7.25657326e-01 7.80928249e-01  
8.29105508e-01 6.20032814e-01 3.01234018e-01 5.58154512e-02  
1.09365871e-01 4.41834837e-01 7.86403336e-01 8.54671841e-01  
7.25871029e-01 7.49769833e-01 8.60018826e-01 8.48475760e-01  
7.93461173e-01 8.30151279e-01 8.38238904e-01 7.46830716e-01  
8.23250136e-01 8.07749355e-01 5.21476656e-01 1.51179416e-01  
1.93296940e-01 6.43519638e-01 8.66847326e-01 7.99175192e-01  
7.32236741e-01 8.36613090e-01 8.58973603e-01 7.10036806e-01  
5.93002877e-01 6.76919355e-01 8.31300490e-01 8.22357229e-01  
7.87742805e-01 8.57211603e-01 7.19108684e-01 2.69220701e-01  
3.32313804e-01 7.92583272e-01 8.69718115e-01 7.69382790e-01  
7.94654102e-01 8.64282887e-01 7.76566173e-01 5.26547832e-01  
3.76112150e-01 4.76751198e-01 7.55143627e-01 8.53824054e-01  
7.91149389e-01 8.39106726e-01 8.31564203e-01 4.23639743e-01  
4.87032030e-01 8.63470062e-01 8.47748200e-01 7.72816355e-01  
8.52947815e-01 8.56126914e-01 6.26520525e-01 3.57070009e-01  
2.32239243e-01 3.50765435e-01 6.85159974e-01 8.53280525e-01  
8.21682300e-01 8.23020619e-01 8.73898674e-01 5.52603454e-01  
6.01596881e-01 8.84913792e-01 8.23667880e-01 7.97195327e-01  
8.79116478e-01 8.04003126e-01 4.88515527e-01 2.33873563e-01  
1.32954990e-01 2.82824021e-01 6.47546729e-01 8.55150837e-01  
8.32743227e-01 8.13760126e-01 8.79500639e-01 6.23278481e-01  
6.27274788e-01 8.87553616e-01 8.08346650e-01 7.99217941e-01  
8.83241286e-01 7.40576048e-01 3.95873214e-01 1.58194876e-01  
1.35774498e-01 3.17394838e-01 6.64478692e-01 8.69441926e-01  
8.24799842e-01 8.14857118e-01 8.75637883e-01 5.99084617e-01  
5.60878494e-01 8.79172781e-01 7.99994274e-01 7.79500355e-01  
8.71300238e-01 7.07469723e-01 3.70424796e-01 2.22900094e-01  
2.61840203e-01 4.53336326e-01 7.48030848e-01 8.66036662e-01  
7.69580365e-01 8.21516759e-01 8.58901592e-01 4.99634590e-01  
3.90619385e-01 8.34941475e-01 8.17641589e-01 7.24679551e-01  
8.40919298e-01 7.59144096e-01 5.11722843e-01 4.41270048e-01  
5.16084855e-01 6.84294249e-01 8.42083832e-01 7.76860627e-01  
7.07485281e-01 8.39049325e-01 7.73085343e-01 3.34022719e-01  
2.04963592e-01 6.81793854e-01 8.40705264e-01 6.82730699e-01  
7.26315544e-01 8.23864229e-01 7.46543700e-01 7.28041059e-01  
7.90768951e-01 8.23062208e-01 7.65169592e-01 6.35267424e-01  
7.38940651e-01 8.33830774e-01 5.75147368e-01 1.70537311e-01  
9.87858616e-02 3.81298193e-01 7.51553287e-01 7.80270482e-01  
5.63772865e-01 6.42284817e-01 7.48036762e-01 7.75665140e-01  
7.32949180e-01 6.31842594e-01 5.10614267e-01 6.55642504e-01  
8.07719344e-01 6.34404155e-01 3.08036725e-01 7.70031201e-02  
4.28491533e-02 1.49241169e-01 3.88664918e-01 7.09137364e-01  
7.47600259e-01 5.37788720e-01 4.69614715e-01 4.75974793e-01



```

4.70450663e-01 5.17338109e-01 6.74591782e-01 7.68727058e-01
5.82843252e-01 2.99680167e-01 9.71493362e-02 5.02788596e-02
2.30063923e-03 3.19935119e-02 8.28517967e-02 2.53333160e-01
4.80454805e-01 6.58647918e-01 6.69771998e-01 6.43217647e-01
6.62741331e-01 6.75677391e-01 5.86224763e-01 3.56460503e-01
1.71512333e-01 5.67475697e-02 1.20193571e-02 0.00000000e+00]

```

## A2- Log-vraisemblance d'une image pour une classe

Écrivez une fonction `log_likelihood` : `float np.array x float np.array x float np.array -> float` qui, étant donné une image (donc un tableau de 256 nombres réels) et un couple de paramètres ( `array ( [μ0,...,μ255] )`, `array ( [σ20,...,σ255] )` ), renvoie la log-vraisemblance qu'aurait l'image selon cet ensemble de  $\mu_i$  et  $\sigma_i$  (correspondant à une classe de chiffre). Rappelez-vous que (en mettant  $-\frac{1}{2}$  en facteur) :

$$\log(p(x_0, \dots, x_{255})) = \sum_{i=0}^{255} \log p(x_i) = -\frac{1}{2} \sum_{i=0}^{255} \left[ \log(2\pi\sigma_i^2) + \frac{(x_i - \mu_i)^2}{\sigma_i^2} \right]$$

Notez que le module `np` contient une constante `np.pi` ainsi que toutes les fonctions mathématiques classiques directement applicables sur des vecteurs. Vous pouvez donc éventuellement coder la ligne précédente sans boucle, en une ligne.

**Attention:** dans la matrice `sig` calculée dans la question précédente, pour certains pixels de certaines classes, la valeur de  $\sigma^2$  est égale à 0 (toutes les images de la base d'apprentissage avaient exactement la même valeur sur ce pixel).

- cette valeur pose problème dans le calcul précédent (division par 0)
- Réfléchir à différente manière de traiter ce problème:
  - faible valeur par défaut de  $\sigma$  reflétant une variance très faible mais évitant la division par 0 (usage de `np.maximum` par exemple)
  - vraisemblance de 1 pour le ou les pixels impactés

In [40]:

```

# on utilisera dans la suite le paramètre defeps:
#   positif, il donne la valeur minimale d'écart type
#   = -1, il faut prendre une vraisemblance de 1 pour les pixels concernés
def log_likelihood(img, mu, sig, defsig = 1e-5):
    # votre code
    # demander code pour faire tenir ça en une ligne
    temp=np.maximum(sig,np.ones(sig.shape)*defsig)
    where=np.where(temp>0)
    return (-1/2)*np.sum(np.log(2*np.pi*np.square(temp[where]))+np.square(img[where]-mu[where])/np.square(temp[where]))
    pass

```

In [41]:

```
print(log_likelihood(X_train[0], mu[0], sig[0], 1e-5))
# vraisemblance de l'image 0 selon les paramètres de la classe 0

print([log_likelihood(X_train[0], mu[i], sig[i], 1e-5) for i in range(10)])
# vraisemblance de l'image 0 pour toutes les classes

-90.69963035168726
[-90.69963035168726, -231211311074.5327, -364.8317101985202, -487.01085544875843, -513.128064745155, -387.75946984198, -59610.117733618186, -75567222244.77489, -271.980542616389, -857252055.4774221]
```

In [42]:

```
print(log_likelihood(X_train[0], mu[0], sig[0], -1))
# vraisemblance de l'image 0 selon les paramètres de la classe 0

print([log_likelihood(X_train[0], mu[i], sig[i], -1) for i in range(10)])
# vraisemblance de l'image 0 pour toutes les classes

-111.88760421521837
[-111.88760421521837, -1716629080.9897287, -364.8317101985202, -487.01085544875843, -544.9100255404517, -387.75946984198, -59747.83956373113, -581523.2639945432, -303.76250341168577, -13497.825910916887]
```

Check : le code ci-dessus avec une valeur par défaut de  $1e-5$  pour les sigmas nuls doit donner:

```
-90.69963035168726
```

puis pour toutes les classes:

```
[-90.69963035168726, -231211311074.5327, -364.8317101985202, -487.01085544875843, -513.128064745155, -387.75946984198, -59610.117733618186, -75567222244.77489, -271.980542616389, -857252055.4774221]
```

Avec une vraisemblance de 1 pour les pixels problématiques:

```
[-111.88760421521835, -1716629080.989729, -364.83171019852006, -487.01085544875855, -544.9100255404516, -387.7594698419803, -59747.8395637312, -581523.2639945432, -303.762503411686, -13497.825910916881]
```

## A3- Classification d'une image

Écrire une fonction `classify_image` : `float np.array x float np.array x float np.array` -> `int` qui, étant donnée une image et l'ensemble de paramètres déterminés dans les questions précédentes, renvoie la classe la plus probable de l'image, c'est-à-dire celle dont la log-vraisemblance est la plus grande.

In [43]:

```
def classify_image(img, mu, sig, defeps=1e-5):
    # votre code
    proba=np.ones((10,1))
    for i in range (10):
        proba[i]=log_likelihood(img, mu[i], sig[i], defeps )
    return np.argmax(proba)
    pass
```

In [44]:

```
# check
classify_image(X_train[0], mu, sig, -1)
# l'image 0 est de la classe 0
```

Out[44]:

0

## A4- Classification de toutes les images

Écrire une fonction `classify_all_images` : float np.array x float np.array x float np.array -> float np.array qui, étant donné un tableau  $X$  des images ( $N \times 256$ ) et l'ensemble de paramètres déterminés dans les questions précédentes, renvoie un tableau  $\hat{Y}$  qui donne la prédiction de classe pour toutes les images

In [45]:

```
def classify_all_images(X, mu, sig, defeps=1e-5):
    # votre code
    sh=X.shape[0]
    res=np.zeros(sh)
    for i in range(X.shape[0]):
        res[i]=classify_image(X[i], mu, sig, defeps)
    return res
    pass
```

In [46]:

```
# check
Y_train_hat = classify_all_images(X_train, mu, sig, -1)

print(Y_train_hat) # doit rendre: [0 9 7 ... 6 3 2]
```


[0. 9. 7. ... 6. 3. 2.]

## A5-Matrice de confusion et affichage du résultat des classifications

La matrice de confusion est de la forme  $C \times C$  où  $C$  est le nombre de classe. Les lignes sont les vraies classes, les colonnes sont les classes prédites. Chaque case (i,j) contient le nombre d'images correspondant à la vraie classe i et à la prédiction j. Si votre classifieur est performant, vous devriez observer des pics sur la diagonale.

La fonction `matrice_confusion(Y, Y_hat)` prend en argument un vecteur d'étiquettes réelles et un vecteur de même taille d'étiquettes prédites et retourne la matrice de confusion.

Vous devriez obtenir une matrice de la forme:

 Matrice de confusion

In [47]:

```
def matrice_confusion(Y, Y_hat):  
    # votre code  
    #n=len(np.unique(Y))  
    #m=len(np.unique(Y_hat))  
    n=10  
    m=10  
    res=np.zeros((n,m) )  
    #res=np.zeros((10,10) )  
    temp=np.zeros(Y.size)  
    for i in range(n):  
        for j in range(m):  
            res[i, j]=np.sum(np.bitwise_and(Y==i, Y_hat==j))+temp  
    return res.astype(int)  
pass
```

In [48]:

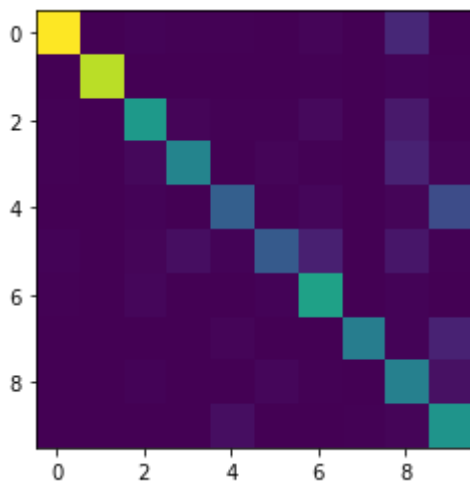
```
# affichage de la matrice de confusion
m = matrice_confusion(Y_train, Y_train_hat)
#matrice_confusion(Y_train, Y_train_hat)
print("Taux de bonne classification: {}".format(np.where(Y_train == Y_train_hat,
1, 0).mean()))

plt.figure()
plt.imshow(m)
```

Taux de bonne classification: 0.8099213356879114

Out[48]:

<matplotlib.image.AxesImage at 0x7efc0627f668>



## A6- Ensemble d'apprentissage, ensemble de test

Dans la procédure que nous avons suivie jusqu'ici, nous avons triché. Les mêmes données servent à apprendre les paramètres et à évaluer le modèle. Evidemment, le modèle est parfaitement adapté et les performances sur-estimées.

Afin de réduire ce biais, nous allons maintenant évaluer les performances sur les données de test. Les performances devraient être plus basses... Mais plus réalistes.

Effectuer ces calculs et afficher le taux de bonne classification et la matrice de confusion.

**Attention:** il faut donc utiliser les paramètres appris sur de nouvelles données sans réapprendre des paramètres spécifiques sinon ça ne marche pas

Afin de mieux comprendre les erreurs (et de vérifier vos connaissances sur numpy): afficher une image de chiffre mal classée, son étiquette prédite et son étiquette réelle. Normalement, vous devez retrouver automatiquement que le premier chiffre mal classé est l'image 10:

 exemple d'erreur

In [49]:

```
# votre code
Y_test_hat=classify_all_images(X_test, mu, sig, -1)
m2=matrice_confusion(Y_test, Y_test_hat)

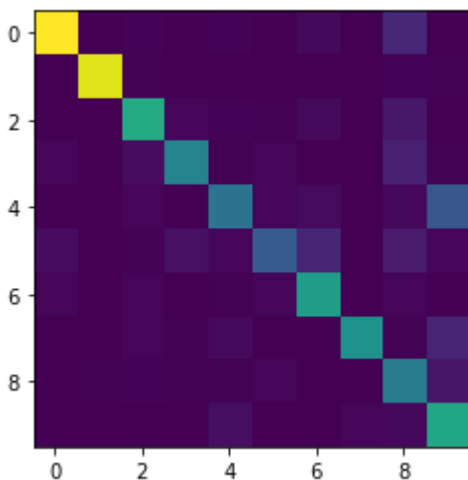
print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1
, 0).mean()))

plt.figure()
plt.imshow(m2)
```

Taux de bonne classification: 0.7934180514825676

Out[49]:

<matplotlib.image.AxesImage at 0x7efc061d3ac8>



## observation

nous voyons bien que le taux de bonne classification de l'ensemble de test est moins bonne que celle de l'entrainement

# Autres modélisations possibles pour les images

## B. Modélisation par une loi de Bernoulli

Soit les indices  $i$  donnant les images et les indices  $j$  référant aux pixels dans l'image, nous cherchons à déterminer la probabilité d'illumination d'un pixel  $j$  pour une collection d'image (d'une seule classe, par exemple les 0).

Collection de 0:

$$X = \{\mathbf{x}_i\}_{i=1, \dots, N}, \quad \mathbf{x}_i \in \{0, 1\}^{256}$$

Modélisation de la variable de Bernoulli  $X_j$ , valeur du pixel  $j$  en écriture factorisée:

$$p(X_j = x_{ij}) = p_j^{x_{ij}}(1 - p_j)^{(1-x_{ij})} = \begin{cases} p_j & \text{si } x_{ij} = 1 \\ 1 - p_j & \text{si } x_{ij} = 0 \end{cases}$$

Expression de la vraisemblance

Maximisation de la vraisemblance  $\Rightarrow \nabla_{\theta} \mathcal{L}(X, \theta) = 0$ :

$$p_j^* = \frac{\sum_i x_{ij}}{N}$$

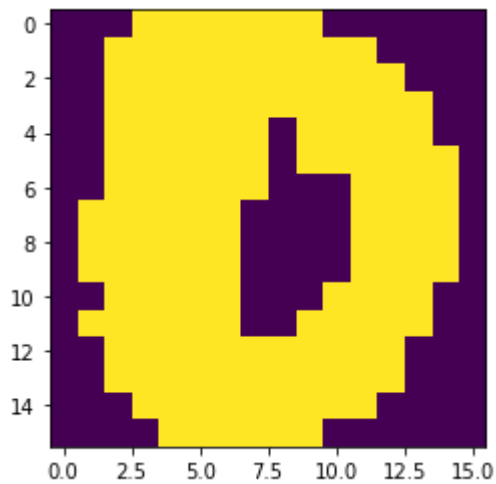
Intuitif: nombre de 1 pour le pixel  $j$  divisé par le nombre d'image = pourcentage d'illumination du pixel  $j$

In [50]:

```
# binarisation des images pour coller avec l'hypothèse de Bernoulli:  
  
Xb_train = np.where(X_train>0, 1, 0)  
Xb_test  = np.where(X_test>0, 1, 0)  
  
# affichage d'une image binaire:  
plt.figure()  
plt.imshow(Xb_train[0].reshape(16,16))
```

Out[50]:

<matplotlib.image.AxesImage at 0x7efc061acc50>





## B-1: Ecrire la fonction d'apprentissage des paramètres qui retourne la matrice theta suivante:

$$\theta^* = \begin{bmatrix} [p_0^*, \dots, p_{255}^*] & \text{Paramètres optimaux de la classe 0 au sens du max de vraisemblance} \\ [p_0^*, \dots, p_{255}^*] & \text{Paramètres optimaux de la classe 1 au sens du max de vraisemblance} \\ \vdots & \\ [p_0^*, \dots, p_{255}^*] & \text{Paramètres optimaux de la classe 9 au sens du max de vraisemblance} \end{bmatrix}$$

Il faut ensuite calculer les :

$$\log p(\mathbf{x}_i | \theta^{(c)}) = \sum_j \log p(X_j = x_{ij}) = \sum_j x_{ij} \log p_j + (1 - x_{ij}) \log(1 - p_j)$$

Faire passer les  $N$  images dans les  $C$  modèles donne un tableau de la forme :

$$\log p(X|\theta) = \begin{bmatrix} \log p(\mathbf{x}_0 | \theta^{(0)}) & \log p(\mathbf{x}_0 | \theta^{(1)}) & \dots & \log p(\mathbf{x}_0 | \theta^{(9)}) \\ \vdots & \vdots & \vdots & \vdots \\ \log p(\mathbf{x}_N | \theta^{(0)}) & \log p(\mathbf{x}_N | \theta^{(1)}) & \dots & \log p(\mathbf{x}_N | \theta^{(9)}) \end{bmatrix}$$

Chaque ligne donne pour une image sa probabilité d'appartenance à chaque classe  $c$ . Un argmax par ligne donne une estimation de la classe.

In [51]:

```
#retour: res[i,j] la proba d'allumage de chaque pixel j dans la classe i
def learnBernoulli ( X,Y ):
    # votre code
    sh=X.shape[1]
    res=np.zeros((10,sh))
    #print(X.shape[1])
    for i in range(10):
        where=np.where(Y==i)
        len=where[0].shape[0]
        temp=X[where[0],:]
        for j in range(sh):
            res[i,j]=np.sum(temp,axis= 0)[j]/len
    return res
pass
```

In [52]:

```

theta = learnBernoulli ( Xb_train,Y_train )
print(theta.shape)
print(theta)

(10, 256)
[[0.00093897 0.00657277 0.03192488 ... 0.02347418 0.00375587 0.
]
 [0.          0.          0.          ... 0.00233372 0.          0.
]
 [0.01941748 0.05987055 0.13430421 ... 0.27993528 0.20711974 0.11326
861]
...
 [0.06666667 0.16078431 0.2745098 ... 0.          0.          0.
]
 [0.01033058 0.05371901 0.1322314 ... 0.01446281 0.00206612 0.
]
 [0.0037037  0.0037037  0.01111111 ... 0.00555556 0.00185185 0.
]]

```

Check du résultat précédent:

```

(10, 256)
[[0.00093897 0.00657277 0.03192488 ... 0.02347418 0.00375587 0.          ]
 [0.          0.          0.          ... 0.00233372 0.          0.          ]
 [0.01941748 0.05987055 0.13430421 ... 0.27993528 0.20711974 0.11326861]
...
 [0.06666667 0.16078431 0.2745098 ... 0.          0.          0.          ]
 [0.01033058 0.05371901 0.1322314 ... 0.01446281 0.00206612 0.          ]
 [0.0037037  0.0037037  0.01111111 ... 0.00555556 0.00185185 0.          ]]

```

## B-2: Ecrire ensuite une fonction de calcul de la vraisemblance d'une image par rapport à ces paramètres

**Attention**  $\log(0)$  n'est pas défini et  $\log(1 - x)$  avec  $x = 1$  non plus ! La solution à ce problème est assez simple, il suffit de seuiller les probabilités d'illumination entre  $\epsilon$  et  $1 - \epsilon$ .

In [53]:

```
# attention: X représente une image seulement donc
#retour: tableau de taille (1,10)
# pour chaque modèle ( classe), calcul du log vraisemblance associé à l'image
def logpobsBernoulli(X, theta):
    # votre code ici
    res=np.zeros((1,10))
    epsilon=1e-4
    #seuillage
    seuil=np.copy(theta)
    seuil[np.where(theta<epsilon)]=epsilon
    seuil[np.where(seuil>=1-epsilon)]=1-epsilon
    for j in range(10):
        res[0,j]=np.dot(np.log(seuil[j,:]),X)+np.dot(np.log(np.ones(seuil[j,:].shape)-seuil[j,:]),(np.ones(X.shape)-X))
    return res
'''methode 2
seuil=np.where(theta<epsilon,epsilon,theta)
seuil=np.where(seuil>1-epsilon,1-epsilon,seuil)
theta=seuil
logp= [ (X*np.log(mod)+(1-X)*np.log(1-mod)).sum() for mod in theta ]
return np.array(logp)'''

pass
```

In [54]:

```
logpobsBernoulli(X_train[0], theta)
```

Out[54]:

```
array([[ 95.28940214, -913.86894309, -131.15364866, -104.77977757,
        -209.07303017,  -85.14159392, -122.04368898, -384.11935833,
        -71.06243118, -252.53913188]])
```

**il faut appliquer la fonction sur Xb\_train contenant que des valeurs binaires pour avoir un log compris entre 0 et 1**

In [55]:

```
logpobsBernoulli(Xb_train[0], theta)
# check avec un epsilon = 1e-4 :
# array([ 95.28940214, -913.86894309, -131.15364866, -104.77977757,
#        -209.07303017,  -85.14159392, -122.04368898, -384.11935833,
#        -71.06243118, -252.53913188])

# ce résultat vous parait-il normal? Qu'est ce qui peut expliquer cette valeur étonnante?
```

Out[55]:

```
array([[ -84.92517398, -742.65796653, -171.38766957, -175.31753619,
        -216.95715849, -161.36508121, -208.21533436, -360.17024632,
        -172.16285293, -287.87685778]])
```

### B-3: Evaluer ensuite vos performances avec les mêmes méthodes que précédemment

In [69]:

```
theta = learnBernoulli ( Xb_train,Y_train )
Y_test_hat = [np.argmax(logpobsBernoulli(Xb_test[i], theta)) for i in range (len
(Xb_test))]
Y_test=np.array(Y_test)
Y_test_hat=np.array(Y_test_hat)
m = matrice_confusion(Y_test, Y_test_hat)

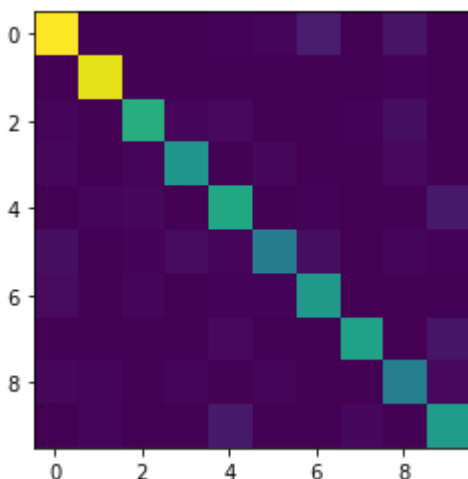
print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1
, 0).mean()))

plt.figure()
plt.imshow(m)
```

Taux de bonne classification: 0.8533724340175953

Out[69]:

<matplotlib.image.AxesImage at 0x7efc0602e400>



## C. Modélisation des profils de chiffre

Comme expliqué dans le TD 2, il est possible de jouer avec les profils des images: chaque image est alors séparée en 16 lignes et pour chaque ligne, nous modélisons l'apparition du premier pixel allumé avec une loi géométrique. Pour plus de simplicité, nous vous donnons ci-dessous la fonction de transformation de la base d'image et son application.

In [57]:

```
#####
# modelisation geometrique
def transfoProfil(X):
    x2 = []
    for x in X:
        ind = np.where(np.hstack((x.reshape(16, 16), np.ones((16,1))))>0.3)
        x2.append( [ind[1][np.where(ind[0] == i)][0] for i in range(16)])
    return np.array(x2)

Xg_train = transfoProfil(Xb_train)
Xg_test  = transfoProfil(Xb_test)
```

In [58]:

```
print(Xg_train[0]) # [3 2 2 2 2 2 2 1 1 1 2 1 2 2 3 4]
# une image est maintenant représentée par 16 entiers

[3 2 2 2 2 2 2 1 1 1 2 1 2 2 3 4]
```

**C-123: Ecrire les fonctions d'apprentissage des paramètres et de calcul de la vraisemblance avec cette modélisation**

In [59]:

```

def learnGeom ( X,Y):
    # votre code ici
    #sh=np.sqrt(X.shape[1])+1
    sh=X.shape[1]
    res=np.zeros((10,sh))
    #print(X.shape[1])
    for i in range(10):
        where=np.where(Y==i)
        len=where[0].shape[0]
        temp=X[where[0],:]
        for j in range(sh):
            #estimation de la probabilité que le pixel j soit allumé en premier
            #pour la classe i par max de vraisemblance
            res[i,j]=len/np.sum(temp[:,j])
    return res
pass

def logpobsGeom(X, theta):
    # votre code ici
    res=np.zeros((1,10))
    epsilon=1e-4
    #seuillage
    seuil=np.copy(theta)
    seuil[np.where(theta<epsilon)]=epsilon
    seuil[np.where(seuil>=1-epsilon)]=1-epsilon
    for j in range(10):
        a=np.sum(np.log(seuil[j,:])) #1* 256
        b=np.dot(np.log(np.ones(seuil[j,:].shape)-seuil[j,:]),(X-np.ones(X.shape
    )))
        #res[0,j]=np.log(seuil[j,:])+np.dot(np.log(np.ones(seuil[j,:].shape)-seu
        il[j,:]),(np.ones(X.shape)-X))
        res[0,j]=a+b
    return res
pass

'''Xg_train=np.array(Xg_train)
Y_train=np.array(Y_train)'''
theta = learnGeom(Xg_train, Y_train)

'''Xg_test=np.array(Xg_test)
Y_test=np.array(Y_test)'''
print(logpobsGeom(Xg_test[1], theta))

Y_train_hat = [np.argmax(logpobsGeom(Xg_train[i], theta)) for i in range (len(Xg
_train))]
Y_test_hat  = [np.argmax(logpobsGeom(Xg_test[i], theta)) for i in range (len(Xg_
test))]

Y_train=np.array(Y_train)
Y_train_hat=np.array(Y_train_hat)
Y_test=np.array(Y_test)
Y_test_hat=np.array(Y_test_hat)
ma = matrice_confusion(Y_train, Y_train_hat)
mt = matrice_confusion(Y_test, Y_test_hat)

print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1
, 0).mean()))

plt.figure()

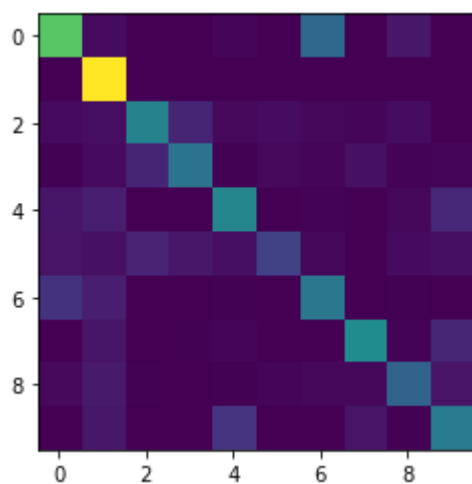
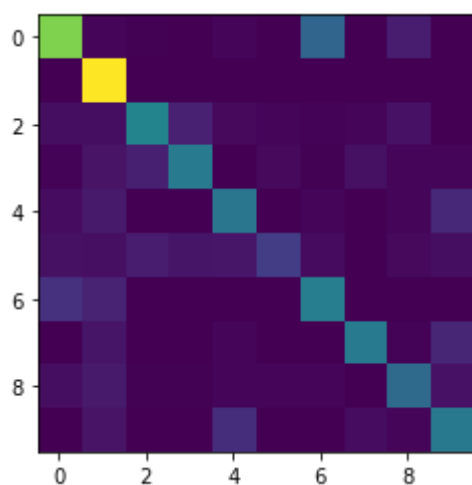
```

```
plt.imshow(ma)
plt.figure()
plt.imshow(mt)
```

```
[[ -8.59383056 -30.91829135 -21.49092763 -29.19427401 -25.74819885
  -24.9698666  -20.39913243 -32.67620668 -24.64316047 -28.58230496]]
Taux de bonne classification: 0.6448354512870642
```


Out[59]:

<matplotlib.image.AxesImage at 0x7efc02b0ec88>



## D. Maximum a posteriori

Etant donné les distributions non uniformes de classes observées sur le jeu de donnée:

 Distribution des classes

Calculer les maxima a posteriori avec les différentes modélisations et vérifier s'il y a un gain en performance avec cette modélisation.

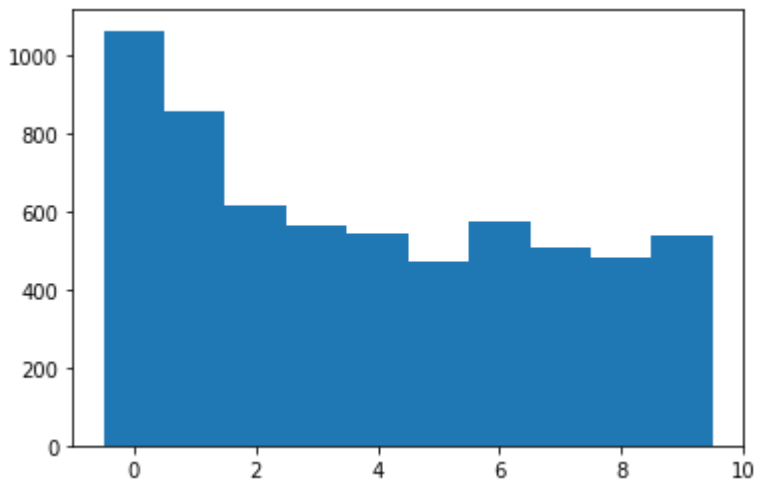
In [60]:

```
# récupération des probabilités a priori sur les données d'apprentissage:
p= np.histogram(Y_train, np.linspace(-0.5,9.5,11))
p = p[0] / p[0].sum()
print(p)
plt.hist(Y_train, np.linspace(-0.5,9.5,11))
```

```
[0.17097447 0.13758228 0.09921336 0.09054423 0.08765452 0.0757746
 0.09198908 0.0818751  0.07770108 0.08669128]
```

Out[60]:

```
(array([1065.,  857.,  618.,  564.,  546.,  472.,  573.,  510.,  48
4.,
        540.]),
 array([-0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5,
9.5]),
 <a list of 10 Patch objects>)
```



In [74]:

```
print(p.shape)
```

```
(10,)
```



In [86]:

```

#Modélisation Loi normale
mu,sig = learnML_parameters ( X_train, Y_train )
def classify_image_prior(img, mu, sig, p, defeps=1e-5):
    # votre code
    proba=np.ones((10,1))
    for i in range (10):
        proba[i]=log_likelihood(img, mu[i], sig[i], defeps )+np.log(p[i])
    return np.argmax(proba)
def classify_all_images_prior(X, mu, sig, p,defeps=1e-5):
    # votre code
    sh=X.shape[0]
    res=np.zeros(sh)
    for i in range(X.shape[0]):
        res[i]=classify_image_prior(X[i], mu, sig,p, defeps)
    return res

Y_train_hat = classify_all_images_prior(X_train, mu, sig,p, -1)
m = matrice_confusion(Y_train, Y_train_hat)
#matrice_confusion(Y_train, Y_train_hat)
#taux_bern=np.where(Y_train == Y_train_hat, 1, 0).mean()
print("Taux de bonne classification: {}".format(np.where(Y_train == Y_train_hat,
1, 0).mean()))

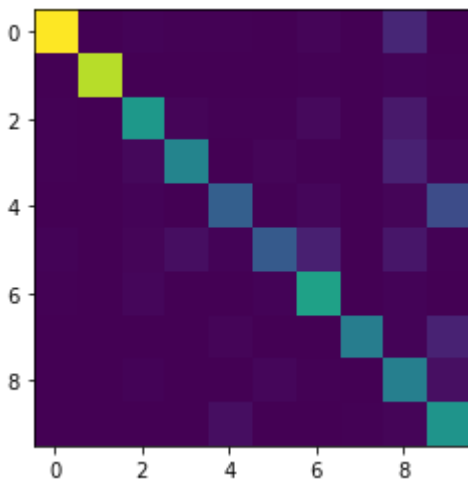
plt.figure()
plt.imshow(m)

```

Taux de bonne classification: 0.8107240327500401

Out[86]:

&lt;matplotlib.image.AxesImage at 0x7efc05bcc908&gt;



In [91]:

```
Y_test_hat_normal=classify_all_images_prior(X_test, mu, sig,p, -1)
m2=matrice_confusion(Y_test, Y_test_hat_normal)
taux_normal=np.where(Y_test == Y_test_hat, 1, 0).mean()
print(taux_normal)
print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1
, 0).mean()))

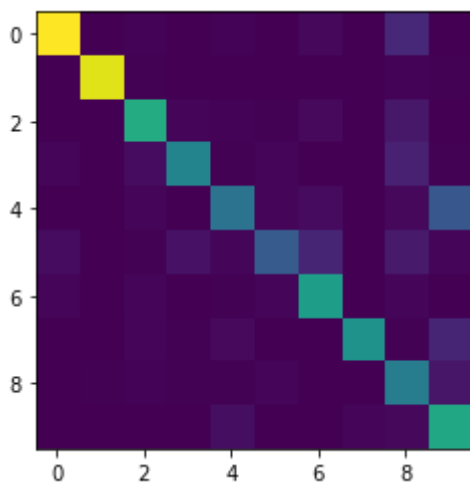
plt.figure()
plt.imshow(m2)
```

0.6184424894102314

Taux de bonne classification: 0.6184424894102314

Out[91]:

<matplotlib.image.AxesImage at 0x7efc06203780>



In [89]:

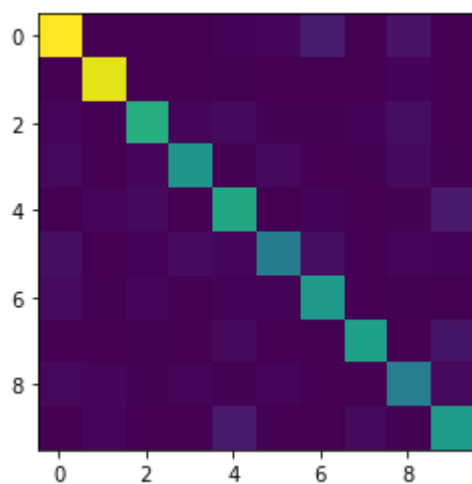
```
# calcul des maximas à posteriori
#Modélisation Bernouilli
theta = learnBernoulli ( Xb_train,Y_train )
Y_test_hat = [np.argmax(logpobsBernoulli(Xb_test[i], theta)+np.log(p)) for i in
range (len(Xb_test))]
Y_test=np.array(Y_test)
Y_test_hat_bern=np.array(Y_test_hat)
m = matrice_confusion(Y_test, Y_test_hat_bern)
taux_bern=np.where(Y_test == Y_test_hat, 1, 0).mean()
print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1
, 0).mean()))

plt.figure()
plt.imshow(m)
```

Taux de bonne classification: 0.8536982730531117

Out[89]:

<matplotlib.image.AxesImage at 0x7efc05a562b0>



**meilleur taux de classification pour Bernouilli**

In [92]:

```

# Modélisation Loi géom
theta = learnGeom(Xg_train, Y_train)
Y_train_hat = [np.argmax(logpobsGeom(Xg_train[i], theta)+np.log(p)) for i in range (len(Xg_train))]
Y_test_hat = [np.argmax(logpobsGeom(Xg_test[i], theta)+np.log(p)) for i in range (len(Xg_test))]

Y_train=np.array(Y_train)
Y_train_hat=np.array(Y_train_hat)
Y_test=np.array(Y_test)
Y_test_hat_geom=np.array(Y_test_hat)
ma = matrice_confusion(Y_train, Y_train_hat)
mt = matrice_confusion(Y_test, Y_test_hat_geom)
taux_geom=np.where(Y_test == Y_test_hat, 1, 0).mean()
print("Taux de bonne classification: {}".format(np.where(Y_test == Y_test_hat, 1, 0).mean()))

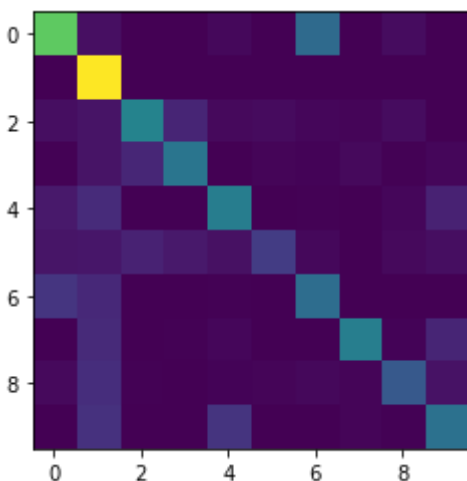
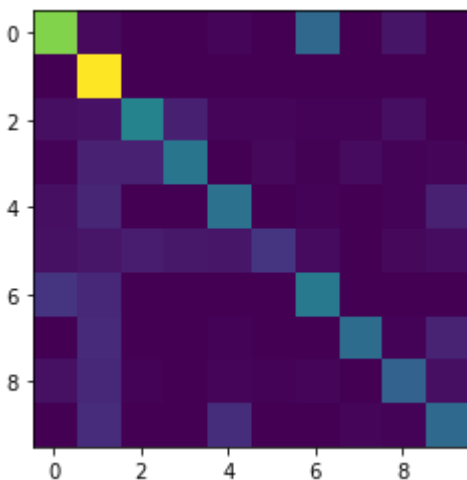
plt.figure()
plt.imshow(ma)
plt.figure()
plt.imshow(mt)

```

Taux de bonne classification: 0.6184424894102314

Out[92]:

&lt;matplotlib.image.AxesImage at 0x7efc062e8ac8&gt;



**moins bon taux de classification pour Binomiale**

## **E. Fusion de modèle**

Réussirez-vous à fusionner les sorties des modèles précédents pour améliorer la performance globale en test?

- En faisant voter les classifieurs
- En pondérant ces votes par leurs performances en apprentissage
- En fusionnant les vraisemblances

In [104]:

```

# faire voter les classifieurs revient à savoir ce qu'ils ont donné comme résultat d'affectation de classe
# pour chaque image, donc c'est les Y obtenus pour chaque loi
# on décide de pondérer sur Y_test_hat_normal, Y_test_hat_bern, Y_test_hat_geom car donne des résultats plus pertinents
a=X_test.shape
print(a)
Y_final=[0 for i in range(a[0])]
taux_list=np.array([taux_normal, taux_bern, taux_geom])
max_taux=np.argmax(taux_list)
print(max_taux)
law_list=[Y_test_hat_normal,Y_test_hat_bern,Y_test_hat_geom ]
for i in range(a[0]):
    if (Y_test_hat_normal[i]== Y_test_hat_bern[i] and Y_test_hat_bern[i]==Y_test_hat_geom[i]):
        Y_final[i]=(Y_test_hat_normal[i])
    if (Y_test_hat_normal[i]!= Y_test_hat_bern[i] and Y_test_hat_bern[i]!=Y_test_hat_geom[i] and Y_test_hat_normal[i]!= Y_test_hat_geom[i]):
        temp=law_list[max_taux]
        Y_final[i]=temp[i]
    else:
        if( Y_test_hat_normal[i]==Y_test_hat_bern[i]):
            pond=(taux_normal+taux_bern)/2
            if(pond>taux_geom):
                Y_final[i]=Y_test_hat_normal[i]
            else:
                Y_final[i]=Y_test_hat_geom[i]
        if( Y_test_hat_normal[i]==Y_test_hat_geom[i]):
            pond=(taux_normal+taux_geom)/2
            if(pond>taux_bern):
                Y_final[i]=Y_test_hat_normal[i]
            else:
                Y_final[i]=Y_test_hat_bern[i]
        if( Y_test_hat_bern[i]==Y_test_hat_geom[i]):
            pond=(taux_bern+taux_geom)/2
            if(pond>taux_normal):
                Y_final[i]=Y_test_hat_bern[i]
            else:
                Y_final[i]=Y_test_hat_normal[i]

print("Taux de bonne classification: {}".format(np.where(Y_test == Y_final, 1, 0).mean()))

```

(3069, 256)

1

Taux de bonne classification: 0.8536982730531117

## F. Proposer une modélisation en 16 niveaux de gris basées sur une loi multinomiale

In [ ]: