

TP3 : Support Vector Machines

In [1]:

```
#imports
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.svm import SVC
from sklearn import metrics
from time import time
import matplotlib.pyplot as plt
import numpy as np
import warnings
```

In [2]:

```
#load dataset
mnist = fetch_openml('mnist_784')
```

[1] Echantillonnage du jeu de données et séparation en entraînement/test

In [3]:

```
#choose sample size
dataset_length = len(mnist.data)
sample_size = 7000
sample_size = min(dataset_length,sample_size)

#extract sample from dataset
sample_indexes = np.random.randint(dataset_length, size= sample_size)
data, target = np.array([mnist.data[i] for i in sample_indexes]), np.array([mnist.target[i] for i in sample_indexes])

#extract train/test according to the proportion given in the subject
train_size = 70/100 #70% de test
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
, random_state = 42)
```

[2] Premières machines

In [4]:

```
clsvm = SVC(kernel='linear')
clsvm.fit(d_train,l_train)
prediction = clsvm.predict(d_test[0:1])[0]
actual= l_test[0]
print("First prediction :",prediction,"Actual :",actual)
```

First prediction : 7 Actual : 7

[3] Analyse

Paramètre noyau

Note : Le paramètre 'precomputed' permet d'indiquer que l'input de la machine a déjà été transformée et redimensionnée, via une opération personnalisée au lieu des standards proposés. Nous n'avons pas de fonctions de transformation particulière à tester, cette option est donc ignorée dans le test. On peut également obtenir le même comportement en passant une fonction de noyau comme argument kernel, qui sera alors appliquée à l'entrée de la machine.

In [5]:

```
kernel_set = ['linear', 'poly', 'rbf', 'sigmoid']
plot = ([], [], [], [], [])
for kernel in kernel_set:
    print("Kernel :", kernel, end='')
    start = time()#TIMER START

    clsvm = SVC(kernel = kernel)
    clsvm.fit(d_train, l_train)
    prediction = clsvm.predict(d_test)

    duration = time() - start #TIMER END

    recall = metrics.recall_score(l_test, prediction, average = 'micro')
    score = metrics.precision_score(l_test, prediction, average='micro')
    error = metrics.zero_one_loss(l_test, prediction)

    plot[0].append(kernel)
    plot[1].append(100*score)
    plot[2].append(100*recall)
    plot[3].append(error)
    plot[4].append(duration)
    print(' | Done')
```

```
Kernel : linear | Done
Kernel : poly | Done
Kernel : rbf | Done
Kernel : sigmoid | Done
```

In [6]:

```
fig=plt.figure()
ax1=fig.add_subplot(111, label="precision")
ax2=ax1.twinx()

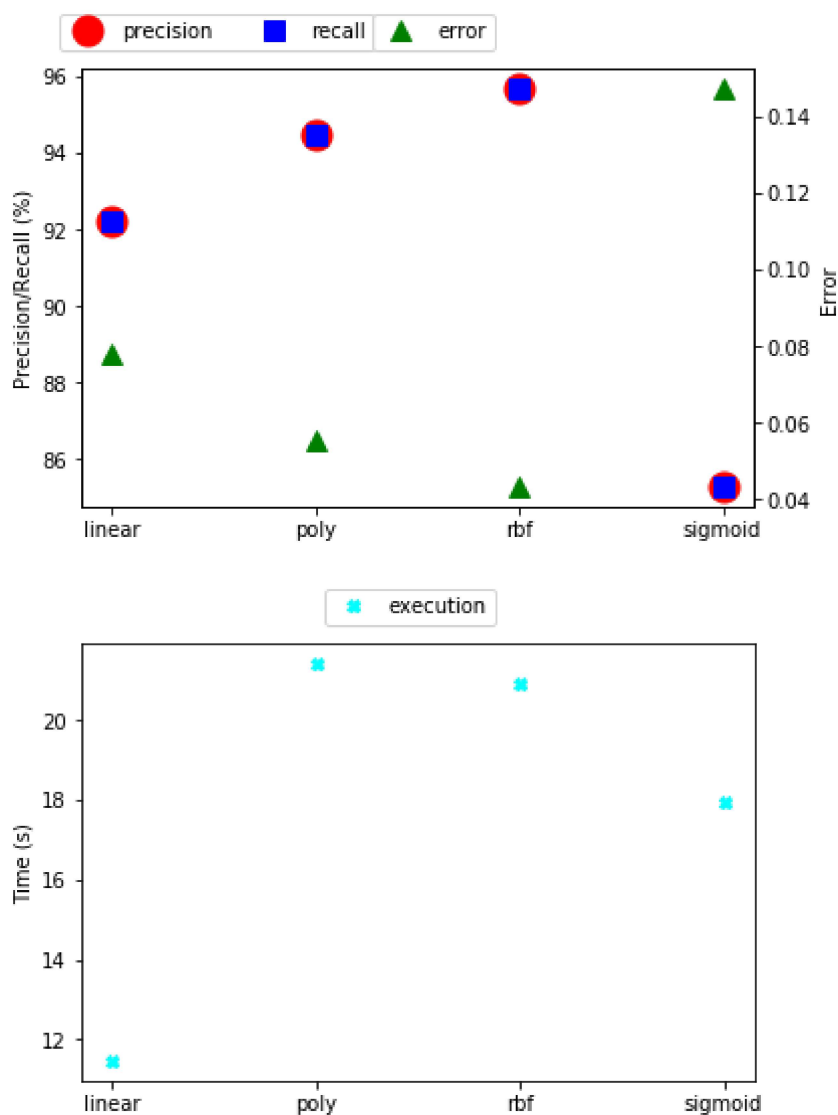
ax1.plot(plot[0],plot[1],linestyle='None',marker='o',ms=15.0,color='r', label='precision')
ax1.plot(plot[0],plot[2],linestyle='None',marker='s',ms=10,color='b', label='recall')
ax2.plot(plot[0],plot[3],linestyle='None',marker='^',ms=10,color='g', label='error')

ax1.set_ylabel("Precision/Recall (%)")
ax2.set_ylabel("Error")
ax1.legend(ncol = 2, bbox_to_anchor=(0.45,1.15))
ax2.legend(bbox_to_anchor = (0.63, 1.15))

fig2=plt.figure()
ax3=fig2.add_subplot(111, label="error")
ax3.plot(plot[0],plot[4],linestyle='None',marker='X',color='cyan',label='execution')
ax3.set_ylabel("Time (s)")
ax3.legend(bbox_to_anchor = (0.63, 1.15))
```

Out[6]:

<matplotlib.legend.Legend at 0x1bc235b0588>



Interprétation

La fonction rbf obtient les meilleurs résultats, surpassant les trois autres en terme de précision, recall, erreur. Cependant, elle est également la plus lente. La fonction linear est plus rapide au prix d'une dégradation de rappel et précision. Les deux autres fonctions semblent moins pertinentes pour ce cas : la fonction poly obtient de bons scores mais qui demeurent inférieurs à ceux de la fonction rbf, pour un temps d'exécution proche. La fonction sigmoïde obtient les pires scores pour un temps d'exécution là aussi comparable à rbf, à éviter donc.

Tolérance aux erreurs

In [7]:

```
kernel = 'rbf'
plot = ([],[],[],[],[])
c_set = [0.1,0.25,0.50,0.75,1.0,1.25,1.50,3,3.50]

for c in c_set:
    print("C :",c, end='')
    start = time()#TIMER START

    clsvm = SVC(kernel = kernel, C = c)
    clsvm.fit(d_train,l_train)
    prediction = clsvm.predict(d_test)

    duration = time() - start #TIMER END

    recall = metrics.recall_score(l_test, prediction, average = 'micro')
    score = metrics.precision_score(l_test, prediction, average='micro')
    error = metrics.zero_one_loss(l_test, prediction)

    plot[0].append(c)
    plot[1].append(100*score)
    plot[2].append(100*recall)
    plot[3].append(error)
    plot[4].append(duration)
    print(' | Done')
```

```
C : 0.1 | Done
C : 0.25 | Done
C : 0.5 | Done
C : 0.75 | Done
C : 1.0 | Done
C : 1.25 | Done
C : 1.5 | Done
C : 3 | Done
C : 3.5 | Done
```

In [8]:

```
fig=plt.figure()
ax1=fig.add_subplot(111, label="precision")
ax2=ax1.twinx()

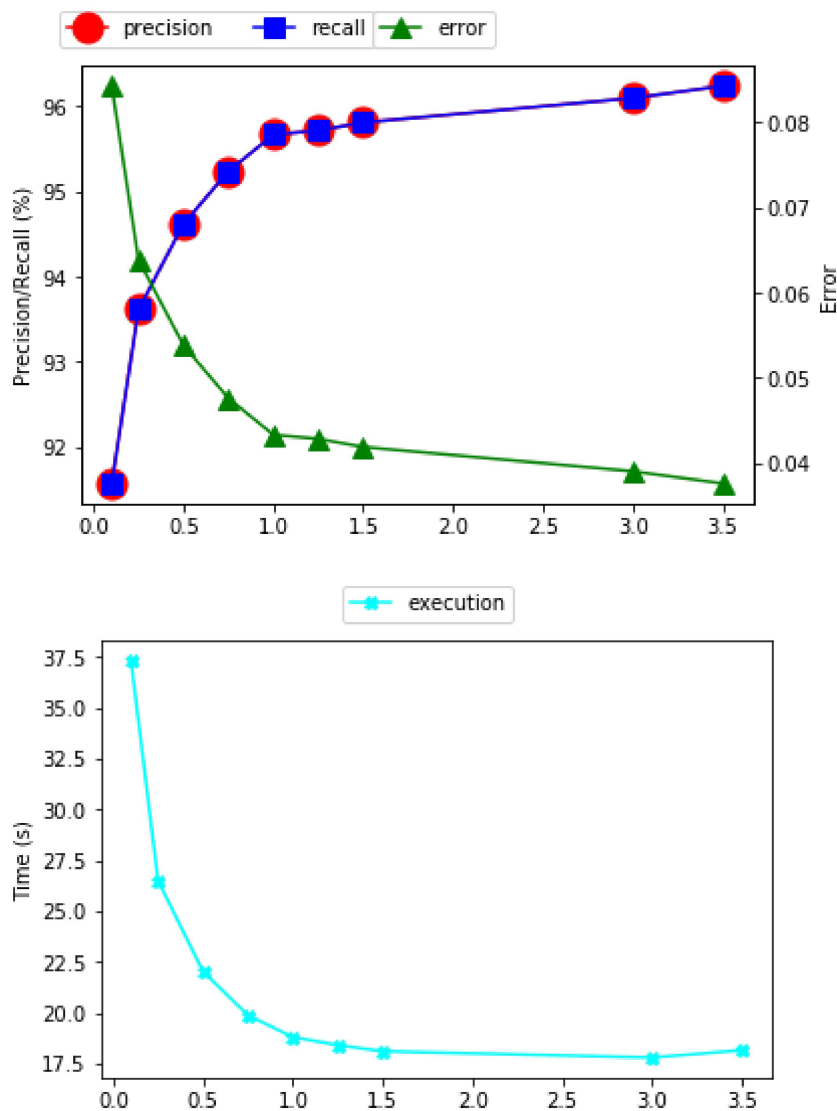
ax1.plot(plot[0],plot[1],marker='o',ms=15.0,color='r', label='precision')
ax1.plot(plot[0],plot[2],marker='s',ms=10,color='b', label='recall')
ax2.plot(plot[0],plot[3],marker='^',ms=10,color='g', label='error')

ax1.set_ylabel("Precision/Recall (%)")
ax2.set_ylabel("Error")
ax1.legend(ncol = 2, bbox_to_anchor=(0.45,1.15))
ax2.legend(bbox_to_anchor = (0.63, 1.15))

fig2=plt.figure()
ax3=fig2.add_subplot(111, label="error")
ax3.plot(plot[0],plot[4],marker='X',color='cyan',label='execution')
ax3.set_ylabel("Time (s)")
ax3.legend(bbox_to_anchor = (0.63, 1.15))
```

Out[8]:

<matplotlib.legend.Legend at 0x1bc214ba4e0>



Interprétation

Il apparaît que les scores obtenus vont croissant avec la marge. Cela s'accompagne de plus d'un temps d'exécution réduit jusqu'à 1.25, où la qualité de prédiction commence à stagner et les temps d'exécution à augmenter de nouveau. Pour une SVM, on cherche à minimiser la valeur de ce paramètre pour obtenir la marge la plus importante, soit une séparation nette des hyperplans. On choisira donc le paramètre 1.25, qui obtient des résultats comparables aux valeurs supérieures et a de plus l'avantage d'un temps d'exécution minimal.

Paramètres retenus

Pour le classifieur SVC, nos résultats indiquent donc qu'il est optimal d'utiliser la fonction noyau RBF et un paramètre de marge C de 1.0. On retiendra également l'alternative d'utiliser la fonction linear comme noyau si l'on souhaite gagner en temps d'exécution. En effet, cette fonction joue à la fois sur les temps d'entraînement et de prédiction, puisque son rôle est de formater les données en entrée du modèle.

Conclusion sur SVC

Les classifieurs SVC semblent obtenir de bons résultats (> 94% de précision), mais le temps d'exécution est important (>17s pour 7000 échantillons).

FIN TP3