TEST TME 4

SAHLI OUSSAMA

Perceptron :

On va comparer les résultats de scores obtenus à partir du perceptron de scikit-learn sur les données gen_arti (2D), avec les résultats des scores obtenue à partir du perceptron implémenté dans le tme3.

La classification_report est exécutée sur les données de test.

Perceptron de scikit-Learn :

```
Erreur moyenne : train 91.600000, test 91.700000
```

	precision	recall	f1-score	support
-1.0	0.93	0.90	0.92	500
1.0	0.90	0.93	0.92	500
micro avg	0.92	0.92	0.92	1000
macro avg	0.92	0.92	0.92	1000
weighted avg	0.92	0.92	0.92	1000

Perceptron implémenté au tme3 :

```
Erreur moyenne : train 90.700000, test 90.600000
```

	precision	recall	f1-score	support
-1.0	0.90	0.91	0.91	500
1.0	0.91	0.90	0.91	500
micro avg	0.91	0.91	0.91	1000
macro avg	0.91	0.91	0.91	1000
weighted avg	0.91	0.91	0.91	1000

On ne constate pas de grandes différences entre ces deux algorithmes.

Maintenant, on va utiliser les données USPS pour réaliser l'apprentissage des deux algorithmes :

(Rappel : Le perceptron que nous avions codé au tme3 a été construit de manière à opposer deux classes (exemple : classe 1 vs classe 8) ou, de manière à opposer une classe aux autres (exemple : 6 vs autres)).

Ici on applique classe 1 vs classe 8.

Perceptron de scikit-Learn :

```
Erreur moyenne : train 100.000000, test 98.837209
```

	precision	recall	f1-score	support
-1.0	0.98	0.99	0.99	166
1.0	1.00	0.98	0.99	264
micro avg	0.99	0.99	0.99	430
macro avg	0.99	0.99	0.99	430
weighted avg	0.99	0.99	0.99	430

Perceptron implémenté au tme3 :

```
Erreur moyenne : train 99.676794, test 97.209302
```

	precision	recall	f1-score	support
-1.0	0.95	0.98	0.96	166
1.0	0.98	0.97	0.98	264
micro avg	0.97	0.97	0.97	430
macro avg	0.97	0.97	0.97	430
weighted avg	0.97	0.97	0.97	430

On ne constate pas de grandes différences sur les résultats obtenues à partir des deux différents algorithmes.

KNN :

Avec les données 2D gen_arti :

```
Erreur moyenne : train 93.400000, test 91.800000
```

	precision	recall	f1-score	support
-1.0	0.91	0.92	0.92	500
1.0	0.92	0.91	0.92	500
micro avg	0.92	0.92	0.92	1000
macro avg	0.92	0.92	0.92	1000
weighted avg	0.92	0.92	0.92	1000

Avec les données USPS :

Classe 1 vs classe 8 :

```
Erreur moyenne : train 99.676794, test 99.302326
```

	precision	recall	f1-score	support
-1.0	0.98	1.00	0.99	166
1.0	1.00	0.99	0.99	264
micro avg	0.99	0.99	0.99	430
macro avg	0.99	0.99	0.99	430
weighted avg	0.99	0.99	0.99	430

DECISION TREE :

Avec les données 2D gen_arti :

```
Erreur moyenne : train 100.000000, test 86.400000
```

	precision	recall	f1-score	support
-1.0	0.88	0.85	0.86	500
1.0	0.85	0.88	0.87	500
micro avg	0.86	0.86	0.86	1000
macro avg	0.86	0.86	0.86	1000
weighted avg	0.86	0.86	0.86	1000

Avec les données USPS :

Classe 1 vs classe 8 :

```
Erreur moyenne : train 100.000000, test 97.209302
```

	precision	recall	f1-score	support
-1.0	0.95	0.98	0.96	166
1.0	0.98	0.97	0.98	264
micro avg	0.97	0.97	0.97	430
macro avg	0.97	0.97	0.97	430
weighted avg	0.97	0.97	0.97	430

REGULARISATION DE TIKHONOV

On intègre une pénalité sur le vecteur poids :

```python
def hinge(datax,datay,w,a=0,l=1):
    """ retourn la moyenne de l'erreur hinge """

    s=0
    n=len(datax)

    for j in range (0,len(datax)):
        p=np.dot(w,datax[j])
        p=p*(-datay[j])
        s=s+max(0,a+p) + l*math.pow(np.linalg.norm(w),2)

    return s/n
```

Par conséquence on modifie la formule du gradient associé à ce coût :

```python
def hinge_g(datax,datay,w,a=0,l=1):
    """ retourne le gradient moyen de l'erreur hinge """
    n_w=np.zeros(len(w))
    n=len(datax)

    for j in range (0,len(datax)):
        p=np.dot(w,datax[j])
        p=p*datay[j]
        if (p<0):
            n_w+=(-datay[j]*datax[j]) + 2*l*w

    n_w*=1/n

    return n_w
```

Avec les données USPS :

Classe 1 vs classe 8 :

```
Erreur moyenne : train 100.000000, test 98.372093
```

	precision	recall	f1-score	support
-1.0	0.98	0.98	0.98	166
1.0	0.99	0.98	0.99	264
micro avg	0.98	0.98	0.98	430
macro avg	0.98	0.98	0.98	430
weighted avg	0.98	0.98	0.98	430

En utilisant un SVM linéaire :

```
Erreur moyenne : train 100.000000, test 98.139535
```

	precision	recall	f1-score	support
-1.0	0.96	0.99	0.98	166
1.0	1.00	0.97	0.98	264
micro avg	0.98	0.98	0.98	430
macro avg	0.98	0.98	0.98	430
weighted avg	0.98	0.98	0.98	430

SVM ET GRID SEARCH :

Observations sur le nombre de vecteurs de support (en utilisant les données 2d gen_arti) :

On Commence tout d'abord à étudier l'évolution du nombre de vecteurs de supports en fonction du noyau et du paramétrage choisit pour le SVM.

(Je réutilise ici la figure du Cours 4 pour expliquer ma façon de calculer les vecteurs de supports.) Le Gamma (γ) ici, représente la distance entre l'hyperplan (frontière) et le point proche. La distance qui sépare l'hyperplan avec chacune des deux droites parallèles est égale à 1/ ||w||.

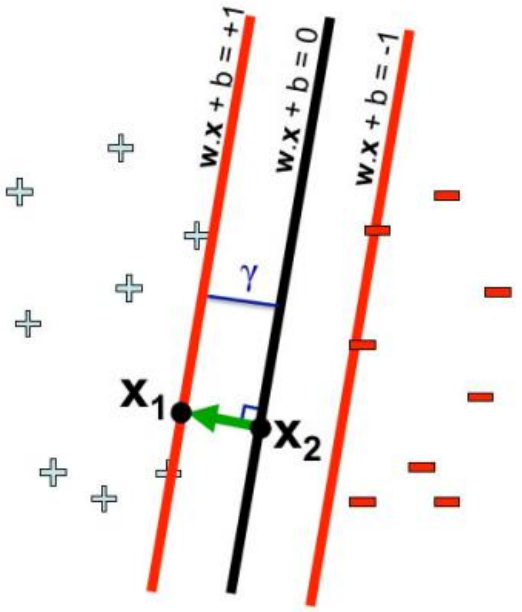
Les vecteurs de supports sont les points qui se situe sur les droites rouges de l'image ci-dessous. Ainsi, les vecteurs de supports sont les points dont la distance avec l'hyperplan est égale à 1/||w||.

La distance d'un point x à un hyperplan w, avec un biais b, est donnée par la formule |<w.x>+b| / ||w||. Le numérateur est en valeur absolue car si on doit classer par exemple un point de la classe -1, alors le résultat de <w.x>+b sera négatif.

Ainsi avec la formule |<w.x>+b| / ||w||, on peut récupérer les points qui sont à la fois sur la droite d'équation w.x+b=+1 et ceux qui sont sur la droite d'équation w.x+b=-1.

Le classifieur SVM (SVC) de scikit-learn possède une fonction decision_function. Cette fonction retourne la valeur w.x+b pour chaque point de l'ensemble de données étudié. Ainsi, les vecteurs de support seront ceux dont la valeur de |w.x +b| sera très proche de 1.

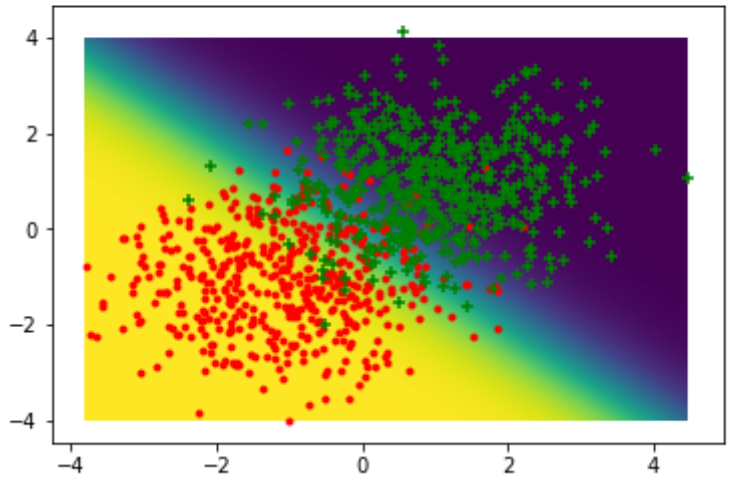
Parce que j'ai remarqué que pour aucun point on obtient une valeur de |w.x+b| exactement égale à 1. J'ai donc décidé de prendre ceux qui se rapprocher très fortement de 1, comme le montre le code ci-dessous.



```
l=svm.decision_function(trainx)
v=0
for r in l:
    if ( math.fabs(r)<1.01 and math.fabs(r)>0.99 ):
        v+=1
print("\nnombre de vecteur de supports: ",v,"\n")
```

En utilisant les données 2D (gen_arti), et un noyau linéaire, le SVM fournit les résultats suivants :

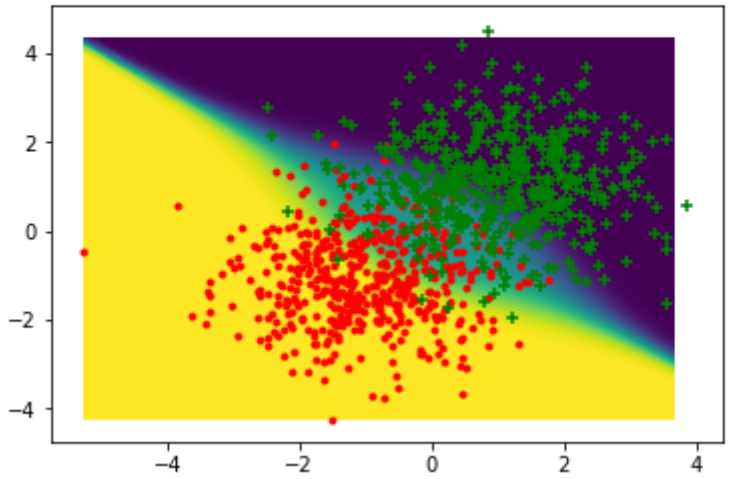
```
svm = SVC(probability=True,gamma=1,kernel="linear",C=1)
svm.fit(trainx, trainy)

Erreur moyenne : train 91.000000, test 91.600000

nombre de vecteur de supports:  6
```



En utilisant les données 2D (gen_arti), et un noyau polynomial, le SVM fournit les résultats suivants :

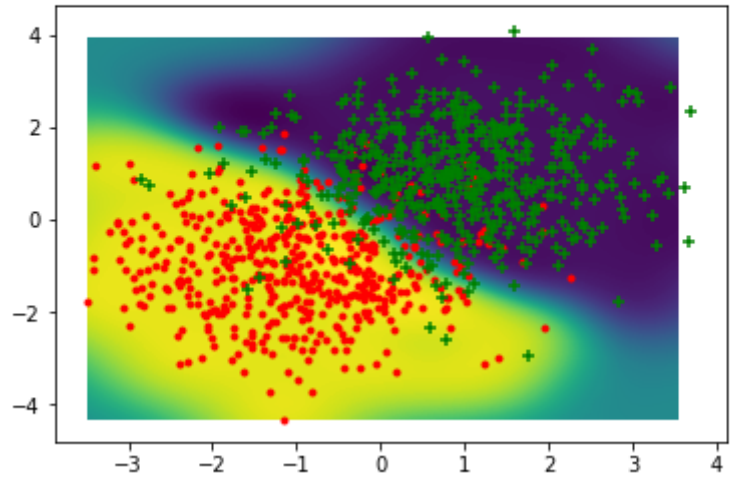
```
svm = SVC(probability=True,gamma=1,kernel="poly",C=1)
svm.fit(trainx, trainy)
```

```
Erreur moyenne : train 92.600000, test 90.400000

nombre de vecteur de supports:  6
```



En utilisant les données 2D (gen_arti), et un noyau gaussien, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=1,kernel="rbf",C=1)
svm.fit(trainx, trainy)

Erreur moyenne : train 90.200000, test 90.000000

nombre de vecteur de supports:  280
```



Observation :

On remarque que pour différents types de noyau, le nombre de vecteurs de supports peut varier fortement.

Dans le cas du noyau linéaire et du noyau polynomiale, le nombre de vecteur de supports obtenue en utilisant chacun des deux noyaux, ne varie pas beaucoup (ils sont même très proche). Sur les figures

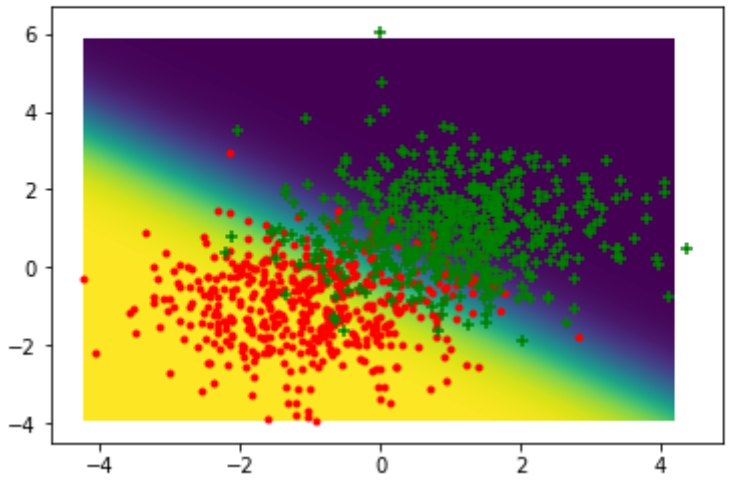
ci-dessous, on peut constater qu'en utilisant un noyau linéaire, on obtient 6 vecteurs de supports. Et en utilisant un noyau polynomial, on obtient 6 vecteurs de supports.

Par contre, si on utilise un noyau gaussien, on peut voir que le nombre de vecteurs de supports augmente fortement. En utilisant un noyau gaussien on obtient 280 vecteurs de support.

Dans nos exemples précédant, on a choisi un paramétrage du type C=1 et gamma=1 pour le SVM de scikit-learn. Mais désormais, on va modifier les valeurs de ces deux paramètres, pour observer les variations sur le nombre de vecteurs de support. On va maintenant prendre C=0,1 et gamma=0,1.

En utilisant les données 2D (gen_arti), et un noyau linéaire, le SVM fournit les résultats suivants :

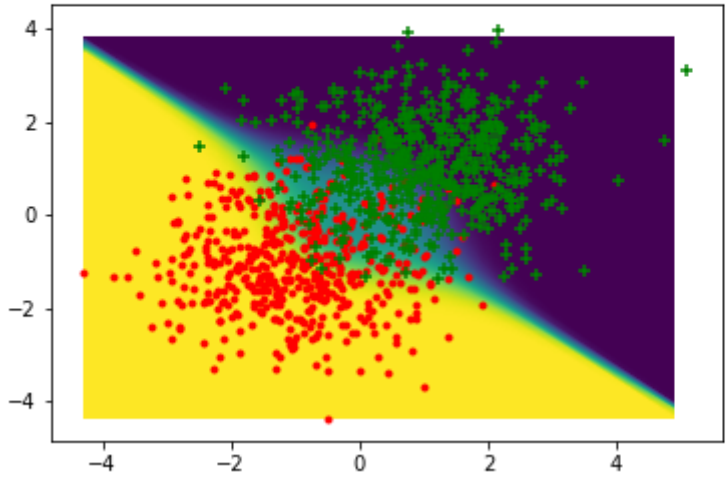
```
svm = SVC(probability=True,gamma=0.1,kernel="linear",C=0.1)
svm.fit(trainx, trainy)

Erreur moyenne : train 89.900000, test 89.600000

nombre de vecteur de supports:  8
```



En utilisant les données 2D (gen_arti), et un noyau polynomial, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=0.1,kernel="poly",C=0.1)
svm.fit(trainx, trainy)
```

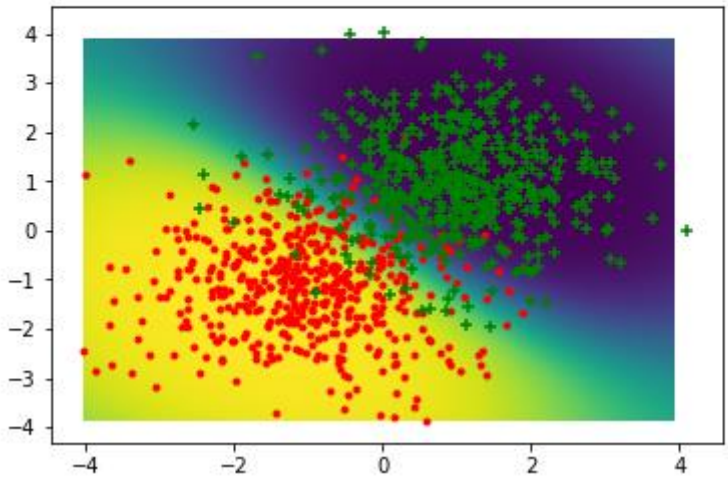
```
Erreur moyenne : train 91.400000, test 90.300000

nombre de vecteur de supports:  5
```



En utilisant les données 2D (gen_arti), et un noyau gaussien, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=0.1,kernel="rbf",C=0.1)
svm.fit(trainx, trainy)
```

```
Erreur moyenne : train 93.200000, test 91.300000

nombre de vecteur de supports:  18
```



Observations :

En ce qui concerne le noyau linéaire et le noyau polynomiale, en changeant les paramètres C et gamma, on voit qu'il n'y a pas de grand changement pour le nombre de vecteurs de supports. On reste à des valeurs toujours compris entre 0 et 10.

Par contre, lorsqu'on utilise le noyau gaussien avec C=0.1 et gamma=0.1, on observe que le nombre de vecteurs de supports a baissé. On est passé de 280 vecteurs de support à seulement 18.

Maintenant on va refaire ces mêmes tests sur les données USPS.

Observations sur le nombre de vecteurs de support (en utilisant les données USPS) :

Ici C=1 et gamma=1. On fait varier le type de noyau utilisé.

On va confronter les données de la classe 1 avec ceux de la classe 8 (classe 1 vs classe 8).

En utilisant les données USPS, et un noyau linéaire, le SVM fournit les résultats suivants :

```
num1=1
num2=8

X,Y=tme3.genere_Data(datax,datay,num1,num2)
testX,testY=tme3.genere_Data(datatestx,datatesty,num1,num2)

svm = SVC(probability=True,gamma=1,kernel="linear",C=1)
svm.fit(X, Y)

Erreur moyenne : train 100.000000, test 98.139535

nombre de vecteur de supports:  34
```

En utilisant les données USPS, et un noyau polynomial, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=1,kernel="poly",C=1)
svm.fit(X, Y)

Erreur moyenne : train 100.000000, test 99.069767

nombre de vecteur de supports:  62
```

En utilisant les données USPS, et un noyau gaussien, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=1,kernel="rbf",C=1)
svm.fit(X, Y)

Erreur moyenne : train 100.000000, test 61.395349

nombre de vecteur de supports:  928
```

Observations :

Comme dans le cas précédent avec les données 2D(gen_arti), on ne constate pas de grands écarts lorsqu'on utilise un noyau linéaire ou polynomial. Ici en utilisant un noyau linéaire, on obtient 34 vecteurs de support, et en utilisant un noyau polynomial, on obtient 62 vecteurs de supports.

Mais lorsqu'on utilise un noyau gaussien, on obtient 928 vecteurs de support. On retrouve la tendance repérée précédemment, où l'on disait que lorsque C=1 et gamma=1, avec un noyau gaussien on obtenait un plus grand nombre de vecteurs de supports.

Maintenant, on va observer le nombre de vecteurs de support obtenue avec C=0,1 et gamma=0,1.

En utilisant les données USPS, et un noyau linéaire, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=0.1,kernel="linear",C=0.1)
svm.fit(X, Y)

Erreur moyenne : train 99.935359, test 98.372093

nombre de vecteur de supports:  34
```

En utilisant les données USPS, et un noyau polynomial, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=0.1,kernel="poly",C=0.1)
svm.fit(X, Y)

Erreur moyenne : train 100.000000, test 99.069767

nombre de vecteur de supports:  62
```

En utilisant les données USPS, et un noyau gaussien, le SVM fournit les résultats suivants :

```
svm = SVC(probability=True,gamma=0.1,kernel="rbf",C=0.1)
svm.fit(X, Y)

Erreur moyenne : train 95.216548, test 92.325581

nombre de vecteur de supports:  584
```

Observations :

Avec gamma=0.1 et C=0.1, il n'y a pas de grand changement visible. Pour le noyau linéaire, on reste à 34 vecteurs de support et pour le noyau polynomial, on reste à 62 vecteurs de supports.

Par contre, pour le noyau gaussien, le nombre de vecteurs de supports à considérablement baissé. On est passé de 928 à 584 vecteurs de supports. Du avec les données USPS, on retrouve les mêmes tendances identifiées précédemment sur les données 2D (gen_arti).

GRID SEARCH :

On commence à réaliser nos expériences sur les données 2D (gen_arti).

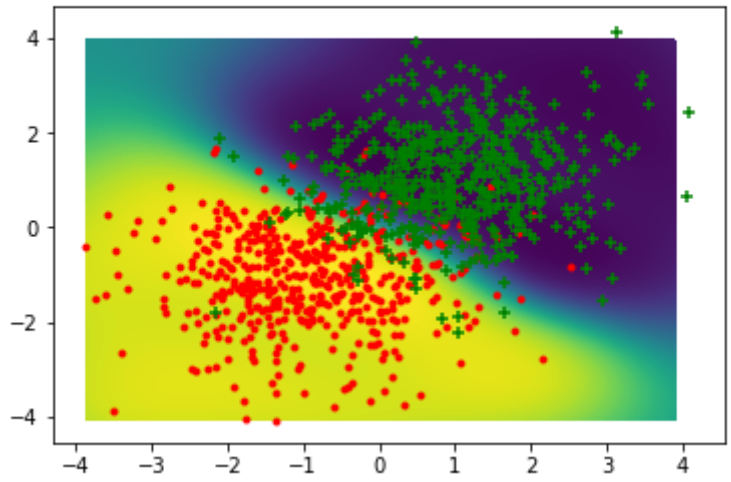
Avant de réaliser un Grid search pour trouver le meilleur estimateur, on va calculer les scores obtenus à partir du SVM (SVC) par défaut. Ça nous permettra de faire des comparaisons significatives entre les scores obtenus avec le SVM par default et le SVM obtenue avec un Grid search.

SVM par défaut :

```
clf = SVC(probability=True,gamma='scale')
clf.fit(trainx, trainy)
```

```
              precision    recall  f1-score   support

        -1.0       0.89      0.90      0.90       500
         1.0       0.90      0.89      0.90       500

   micro avg       0.90      0.90      0.90      1000
   macro avg       0.90      0.90      0.90      1000
weighted avg       0.90      0.90      0.90      1000
```

```
Erreur moyenne : train 91.300000, test 89.700000
```



<u>Maintenant on va appliquer le Grid search :</u>

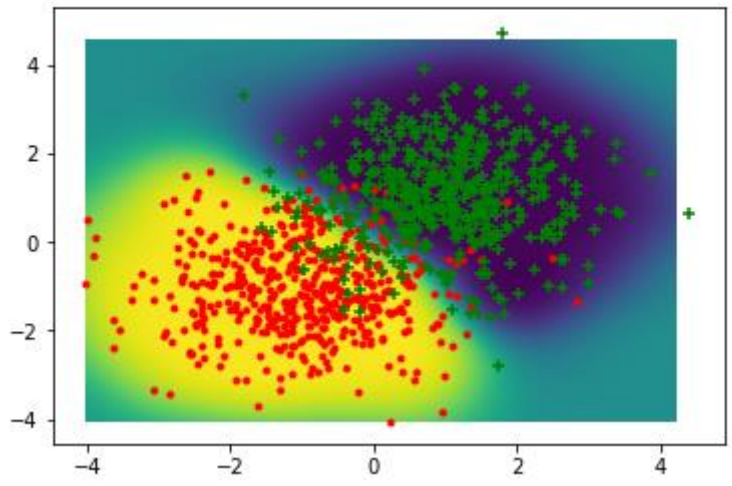
On va commencer par tester plusieurs combinaisons possibles entre les paramètres C, gamma, et kernel du SVM. Ici on va appliquer une recherche exhaustive.

```
parameters = [{'kernel': ['rbf','linear','poly'],
               'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
               'C': [0.1, 1, 10, 100, 1000]},
              ]
```

```
Best Parameters:

{'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}

best Estimator:

SVC(C=0.1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf',
  max_iter=-1, probability=True, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
              precision    recall  f1-score   support

        -1.0       0.91      0.94      0.92       500
         1.0       0.94      0.91      0.92       500

   micro avg       0.92      0.92      0.92      1000
   macro avg       0.92      0.92      0.92      1000
weighted avg       0.92      0.92      0.92      1000
```

```
Erreur moyenne : train 91.800000, test 92.200000
```



Observations :

Avec le Grid search, on constate une amélioration sur le score calculé à partir des données test.

Maintenant, au lieu de faire une recherche exhaustive des paramètres, on va réaliser une recherche aléatoire sur les paramètres, où chaque paramètre est échantillonné à partir d'une distribution sur les valeurs de paramètres possibles

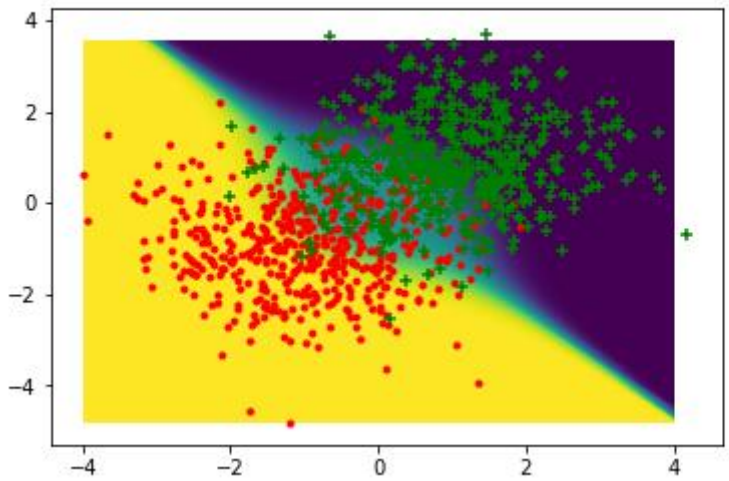
RandomizedSearchCV :

```
distributions = {'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf','linear','poly'], 'class_weight':['balanced', None]}

clf = RandomizedSearchCV(SVC(probability=True), distributions, random_state=0,cv=5)
clf.fit(trainx,trainy)
```

```
Best Parameters:

{'C': 53.556571830075626, 'class_weight': None, 'gamma': 0.02062032805354869,
'kernel': 'poly'}

best Estimator:

SVC(C=53.556571830075626, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.02062032805354869,
  kernel='poly', max_iter=-1, probability=True, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

	precision	recall	f1-score	support
-1.0	0.88	0.94	0.91	500
1.0	0.94	0.87	0.90	500
micro avg	0.90	0.90	0.90	1000
macro avg	0.91	0.90	0.90	1000
weighted avg	0.91	0.90	0.90	1000

```
Erreur moyenne : train 90.100000, test 90.300000
```



<u>Observations :</u>

On ne constate pas de grand changement ou de nette améliorations par rapport aux expérimentation précédentes.

<u>Maintenant, on va utiliser les données USPS pour réaliser nos expérimentations.</u>

On va confronter la classe 1 à la classe 8 (classe 1 vs classe 8).

```
num1=1
num2=8
X,Y=tme3.genere_Data(datax,datay,num1,num2)
testX,testY=tme3.genere_Data(datatestx,datatesty,num1,num2)
```

Avec le SVM par défaut, on obtient :

```
clf = SVC(probability=True,gamma='scale')
clf.fit(X, Y)
```

```
              precision    recall  f1-score   support

        -1.0       0.98      1.00      0.99       166
         1.0       1.00      0.98      0.99       264

   micro avg       0.99      0.99      0.99       430
   macro avg       0.99      0.99      0.99       430
weighted avg       0.99      0.99      0.99       430


Erreur moyenne : train 99.935359, test 99.069767
```

En utilisant un Grid search :

```
parameters = [{'kernel': ['rbf','linear','poly'],
               'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
               'C': [0.1, 1, 10, 100, 1000]},
              ]

clf = GridSearchCV( SVC(probability=True), parameters,refit = True,cv=5)
clf.fit(X,Y)
```

```
Best Parameters:

{'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}

best Estimator:

SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=True, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
              precision    recall  f1-score   support

        -1.0       0.98      1.00      0.99       166
         1.0       1.00      0.98      0.99       264

   micro avg       0.99      0.99      0.99       430
   macro avg       0.99      0.99      0.99       430
weighted avg       0.99      0.99      0.99       430


Erreur moyenne : train 100.000000, test 99.069767
```

Avec RandomizedSearchCV :

```
Best Parameters:

{'C': 180.8369231994153, 'class_weight': 'balanced', 'gamma': 0.009116102911900048,
'kernel': 'rbf'}

best Estimator:

SVC(C=180.8369231994153, cache_size=200, class_weight='balanced', coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.009116102911900048,
  kernel='rbf', max_iter=-1, probability=True, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

	precision	recall	f1-score	support
-1.0	0.98	1.00	0.99	166
1.0	1.00	0.98	0.99	264
micro avg	0.99	0.99	0.99	430
macro avg	0.99	0.99	0.99	430
weighted avg	0.99	0.99	0.99	430

```
Erreur moyenne : train 100.000000, test 99.069767
```

Observations :

Les résultats étaient déjà très bons en utilisant le SVM par défaut. On ne constate de grandes améliorations. Les résultats obtenus sont proches.

Pour les trois utilisations du SVM (par défaut, Grid search, RandomizedSearch), les résultats sont très similaires et proches de 100%.

APPRENTISSAGE MULTICLASSE :

Ici je vais tout d'abord appliquer un one vs one, puis un one vs all. Je vais utiliser les données USPS car ce sont des données multi classes. Il y a 10 classes différentes dans ce jeu de données. Le modèle SVM que je donne en entrée des algorithmes (one vs one, et one vs all), sera le modèle SVM qui m'a permis d'obtenir les meilleurs résultats précédemment lors de l'utilisation Grid search.

```
SVC(C=10,gamma=0.01,kernel="rbf")
```

Ainsi j'obtiens les résultats suivants pour le One vs One :

	precision	recall	f1-score	support
0	0.98	0.99	0.98	359
1	0.99	0.97	0.98	264
2	0.93	0.94	0.94	198
3	0.94	0.90	0.92	166
4	0.90	0.94	0.92	200
5	0.90	0.94	0.92	160
6	0.98	0.95	0.96	170
7	0.98	0.95	0.96	147
8	0.92	0.95	0.93	166
9	0.96	0.96	0.96	177
micro avg	0.95	0.95	0.95	2007
macro avg	0.95	0.95	0.95	2007
weighted avg	0.95	0.95	0.95	2007

Erreur moyenne : train 99.986284, test 95.266567

Résultats pour le One vs All :

	precision	recall	f1-score	support
0	0.97	0.99	0.98	359
1	0.99	0.97	0.98	264
2	0.93	0.94	0.93	198
3	0.96	0.91	0.93	166
4	0.92	0.94	0.93	200
5	0.92	0.94	0.93	160
6	0.98	0.97	0.97	170
7	0.97	0.94	0.96	147
8	0.94	0.93	0.93	166
9	0.95	0.97	0.96	177
micro avg	0.95	0.95	0.95	2007
macro avg	0.95	0.95	0.95	2007
weighted avg	0.95	0.95	0.95	2007

Erreur moyenne : train 99.986284, test 95.465869

STRING KERNEL :

J'ai récupéré un dataset contenant différents textes de différents auteurs. Les données sont des textes écrit par différents auteurs. Dans mon jeu de données, j'ai trois classes différentes correspondants aux trois auteurs différents. Ces classes sont représenté par 0,1 et 2.

Pour pouvoir passer mes données à la méthode fit du SVM, j'ai d'abord construit la matrice de similarité associé à l'ensemble de train :

```
mat=np.zeros((len(X_train),len(X_train)))
for i in range(0,len(X_train)):
    x=np.zeros(len(X_train))
    cpt=0
    for j in range(0,len(X_train)):
        x[cpt]=Kernel_String(X_train[i],X_train[j],1)
        cpt+=1
    mat[i]=x
```

La similarité entre deux textes (phrases) est calculée avec la fonction Kernel String:

```
def Kernel_String(A,B,l=0.5):
    dA={}
    for i in range (0,len(A)-1):
        for j in range (i+1,len(A)):
            dA[A[i]+A[j]]=math.pow(l,j-i+1)

    dB={}
    for i in range (0,len(B)-1):
        for j in range (i+1,len(B)):
            dB[B[i]+B[j]]=math.pow(l,j-i+1)

    d=np.unique(list(dA.keys())+list(dB.keys()))

    s=0
    for c in d:
       v1=0
       v2=0
       if (c in dA):
           v1=dA[c]
       if (c in dB):
           v2=dB[c]
       s=s+(v1*v2)
    return s
```

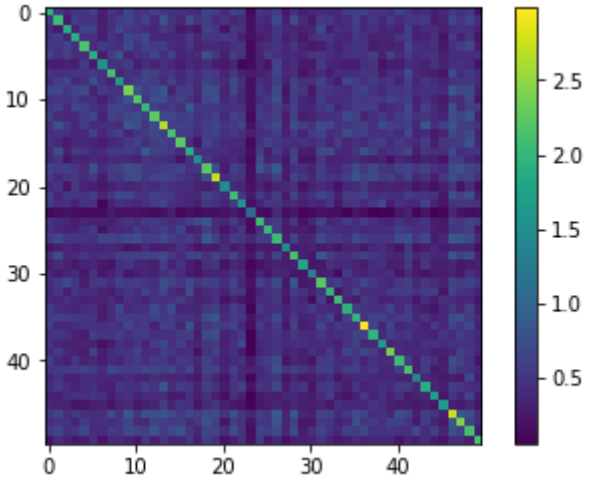
La fonction kernel_String ci-dessus calcule la similarité entre deux phrases. Pour cela elle définit toutes les sous séquences de taille 2 (contiguë et non contiguë). Ainsi ici, chaque sous-séquence u est définit tel-que $u \in \Sigma^n$

Avec n=2. Pour des tailles de sous-séquences de longueur n>4, le calcul des similarités sur des tailles de texte moyen, serait vraiment trop long. Donc, je me suis contenté de garder n=2.

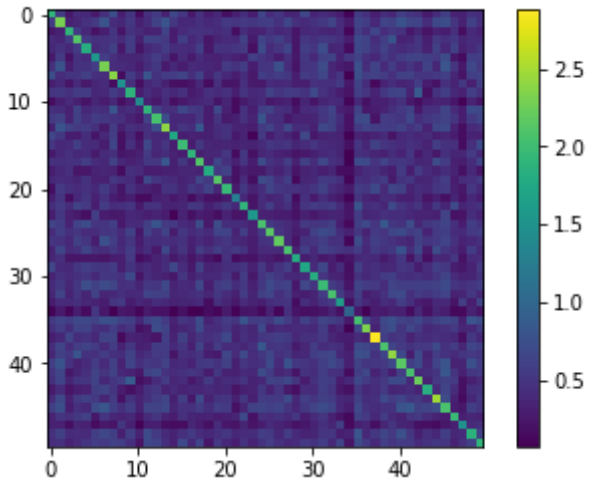
Scores obtenus quand on considère que seulement 50 exemples.

	precision	recall	f1-score	support
0	0.58	0.48	0.53	29
1	0.12	0.08	0.10	12
2	0.22	0.44	0.30	9
micro avg	0.38	0.38	0.38	50
macro avg	0.31	0.34	0.31	50
weighted avg	0.41	0.38	0.38	50

La matrice de similarité de l'ensemble de train :



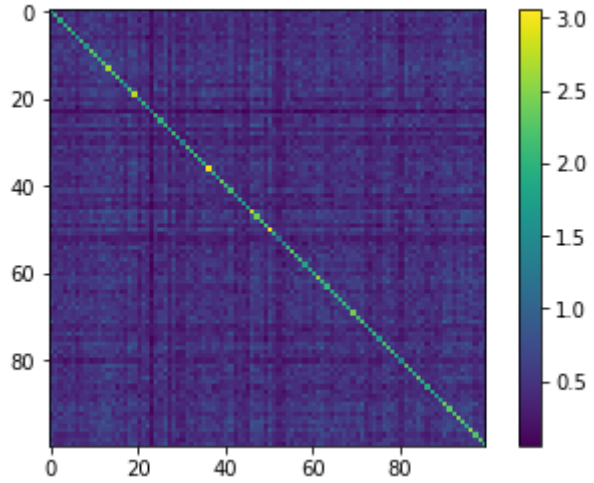
Matrice de similarité de l'ensemble de test :



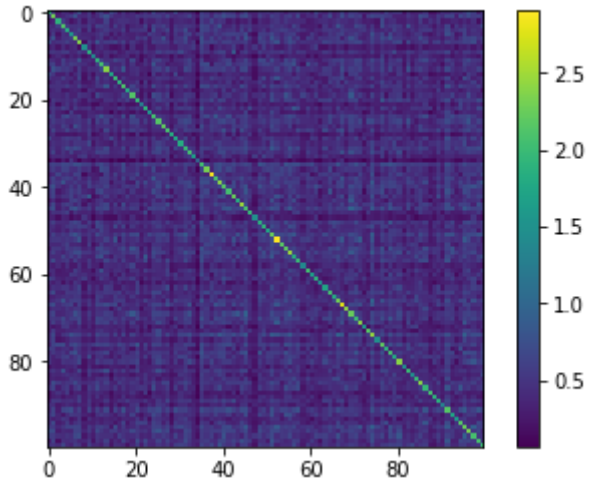
Score obtenu quand on considère seulement que 100 exemples :

	precision	recall	f1-score	support
0	0.50	0.39	0.44	56
1	0.17	0.20	0.19	25
2	0.19	0.26	0.22	19
micro avg	0.32	0.32	0.32	100
macro avg	0.29	0.29	0.28	100
weighted avg	0.36	0.32	0.33	100

Matrice de similarité de l'ensemble train :



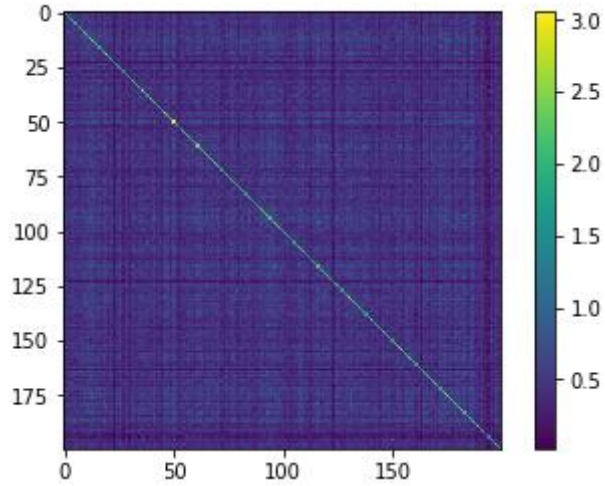
Matrice de similarité de l'ensemble test :



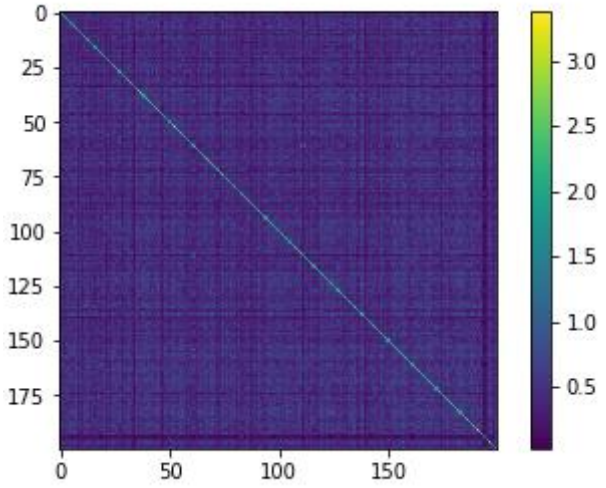
Score obtenu quand on considère seulement que 200 exemples. :

	precision	recall	f1-score	support
0	0.48	0.57	0.52	95
1	0.30	0.29	0.29	55
2	0.29	0.20	0.24	50
micro avg	0.40	0.40	0.40	200
macro avg	0.36	0.35	0.35	200
weighted avg	0.38	0.40	0.39	200

Matrice de similarité de l'ensemble train :



Matrice de similarité de l'ensemble de test :



Observations :

On remarque que quand le nombre d'exemples augmente, la prédiction sur les données s'améliorent. On remarque notamment dans la classification report que le score de prédiction pour les classes 1 et 2 s'améliorent. Car quand on considérait que très peu de données (50 par exemple), le SVM donnait de très mauvais score quand il s'agissait de prédire la classe 2.

Mais le problème est qu'avec un nombre d'exemples trop grand, le temps d'exécution est trop long.

Avec seulement 200 exemples par exemple, le temps de calcule est assez long.

Mais en dessous de 100 exemples, c'est assez rapide.

Maintenant, je vais faire varier la valeur du paramètre lambda, afin d'observer les différents scores obtenus.

J'ai choisi de ne considéré que seulement 50 exemples, et je fais varier lambda de 0.1 à 0.9 avec un pas de 0,1.

```
lambda= 0.1
```

	precision	recall	f1-score	support
0	0.58	1.00	0.73	29
1	0.00	0.00	0.00	12
2	0.00	0.00	0.00	9
micro avg	0.58	0.58	0.58	50
macro avg	0.19	0.33	0.24	50
weighted avg	0.34	0.58	0.43	50

```
lambda= 0.2
```

	precision	recall	f1-score	support
0	0.58	1.00	0.73	29
1	0.00	0.00	0.00	12
2	0.00	0.00	0.00	9
micro avg	0.58	0.58	0.58	50
macro avg	0.19	0.33	0.24	50
weighted avg	0.34	0.58	0.43	50

```
lambda= 0.30000000000000004
```

	precision	recall	f1-score	support
0	0.58	1.00	0.73	29
1	0.00	0.00	0.00	12
2	0.00	0.00	0.00	9
micro avg	0.58	0.58	0.58	50
macro avg	0.19	0.33	0.24	50
weighted avg	0.34	0.58	0.43	50

lambda= 0.4

	precision	recall	f1-score	support
0	0.62	0.83	0.71	29
1	0.00	0.00	0.00	12
2	0.20	0.22	0.21	9
micro avg	0.52	0.52	0.52	50
macro avg	0.27	0.35	0.31	50
weighted avg	0.39	0.52	0.45	50

lambda= 0.5

	precision	recall	f1-score	support
0	0.58	0.48	0.53	29
1	0.12	0.08	0.10	12
2	0.22	0.44	0.30	9
micro avg	0.38	0.38	0.38	50
macro avg	0.31	0.34	0.31	50
weighted avg	0.41	0.38	0.38	50

lambda= 0.6

	precision	recall	f1-score	support
0	0.55	0.41	0.47	29
1	0.14	0.08	0.11	12
2	0.19	0.44	0.27	9
micro avg	0.34	0.34	0.34	50
macro avg	0.29	0.31	0.28	50
weighted avg	0.38	0.34	0.35	50

lambda= 0.7000000000000001

	precision	recall	f1-score	support
0	0.57	0.41	0.48	29
1	0.12	0.08	0.10	12
2	0.19	0.44	0.27	9
micro avg	0.34	0.34	0.34	50
macro avg	0.30	0.31	0.28	50
weighted avg	0.40	0.34	0.35	50

lambda= 0.8

	precision	recall	f1-score	support
0	0.57	0.41	0.48	29
1	0.12	0.08	0.10	12
2	0.19	0.44	0.27	9
micro avg	0.34	0.34	0.34	50
macro avg	0.30	0.31	0.28	50
weighted avg	0.40	0.34	0.35	50

lambda= 0.9

	precision	recall	f1-score	support
0	0.59	0.34	0.43	29
1	0.00	0.00	0.00	12
2	0.19	0.67	0.30	9
micro avg	0.32	0.32	0.32	50
macro avg	0.26	0.34	0.24	50
weighted avg	0.38	0.32	0.31	50

<u>Observations :</u>

Avec une valeur de lambda égale à 0.1, on constate que l'on obtient un bonne prédiction sur les données de la classe 0. Par contre,sur la prédiction des données des classes 1 et 2, les résultats sont très mauvais. D'ailleurs je penses que les résultats ne sont pas très bon car le nombre d'exemple considéré pour réaliser l'apprentissage n'est pas suffisant. Il faudrait augmenter ce nombre, mais cela reviendrai à avoir un temps de calculs beaucoups trop long.

Lorsque la valeur de lambda augmente, les prédictions sur les données des classes 1 et 2 sont meilleurs, tandis que la prédiction sur la classe 0 devient moins bonnes. De plus, même si les prédictions sur les donnés des classes 1 et 2 deviennent meilleurs, elle sont toujours faibles.