TP1

December 9, 2018

1 Rapport de TP1 - KNN

Réaliser par Mohamed ELFILALI et Nguyen Duc Hau

1.1 Manipulation de la base de données MNIST

Dans cette partie on va essayer de voir et comprendre les données de la base de données MNIST fourni par SKLEARN.

1.1.1 Visualisation

On commence par charger les bibliothèques dédiées pour visualiser nos données.

```
In [1]: from sklearn import datasets
        import matplotlib.pyplot as plt
        from sklearn import datasets
        from sklearn import model_selection
        from sklearn import neighbors
        import numpy as np
        from sklearn import metrics
        import warnings;
        warnings.simplefilter('ignore')

        mnist = datasets.fetch_mldata('MNIST original')
```

La fonction PRINT on lui attribuant MNIST comme paramètre permet de visualiser le contenue de notre DATASET. En revanche la fonction LEN nous donne le nombre des éléments de notre DATASET

```
In [2]: print("Les éléments de MNIST")
        print(mnist)
        print("")
        print("Les composant de Data et target")
        print (mnist.data)
        print (mnist.target)
        print("")
        print("La taille du DATASET")
        print(len(mnist.data))
```

```
        print("")
        print("L'allure du DATASET")
        print (mnist.data.shape)
        print (mnist.target.shape)
```

```
Les éléments de MNIST
{'DESCR': 'mldata.org dataset: mnist-original', 'COL_NAMES': ['label', 'data'], 'target': array
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)}

Les composant de Data et target
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
[0. 0. 0. ... 9. 9. 9.]

La taille du DATASET
70000

L'allure du DATASET
(70000, 784)
(70000,)
```

La fonction HELP nous permet de comprendre l'utilité de chaque fonction

```
In [3]: help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

On peut accéder par la suite au diffèrent entre de notre data par plusieurs manières

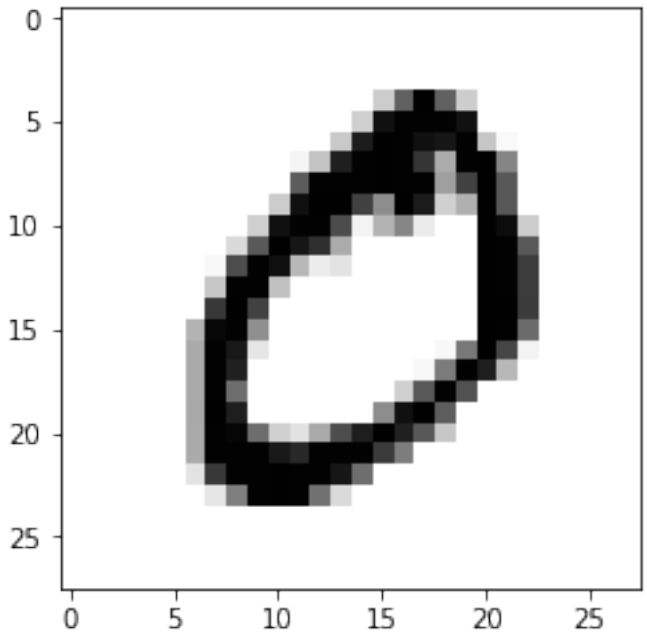
```
In [4]: print(mnist.data[0])
        print(mnist.data[0][1])
        print(mnist.data[:,1])
        print(mnist.data[:100])
```

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0  51 159 253 159  50   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0  48 238 252 252 252 237   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0  54 227 253 252 239 233 252  57   6   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0  10  60 224 252 253 252 202  84 252
 253 122   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0 163 252 252 252 253 252 252  96 189 253 167   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0  51 238 253 253 190 114 253 228
  47  79 255 168   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0  48 238 252 252 179  12  75 121  21   0   0 253 243  50   0   0   0
   0   0   0   0   0   0   0   0   0   0  38 165 253 233 208  84   0   0
   0   0   0   0 253 252 165   0   0   0   0   0   0   0   0   0   0   0
   0   7 178 252 240  71  19  28   0   0   0   0   0   0 253 252 195   0
   0   0   0   0   0   0   0   0   0   0   0  57 252 252  63   0   0   0
   0   0   0   0   0   0 253 252 195   0   0   0   0   0   0   0   0   0
   0   0   0 198 253 190   0   0   0   0   0   0   0   0   0   0 255 253
 196   0   0   0   0   0   0   0   0   0   0   0  76 246 252 112   0   0
   0   0   0   0   0   0   0   0 253 252 148   0   0   0   0   0   0   0
   0   0   0   0  85 252 230  25   0   0   0   0   0   0   0   0   7 135
 253 186  12   0   0   0   0   0   0   0   0   0   0   0  85 252 223   0
   0   0   0   0   0   0   0   7 131 252 225  71   0   0   0   0   0   0
   0   0   0   0   0   0  85 252 145   0   0   0   0   0   0   0  48 165
 252 173   0   0   0   0   0   0   0   0   0   0   0   0   0   0  86 253
 225   0   0   0   0   0   0 114 238 253 162   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  85 252 249 146  48  29  85 178 225 253
 223 167  56   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  85 252 252 252 229 215 252 252 252 196 130   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0  28 199 252 252 253 252 252 233
 145   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0  25 128 252 253 252 141  37   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0]
0
[0 0 0 ... 0 0 0]
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
```

```
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

On peut aussi visualiser d'une façon graphique les éléments de notre DATASET

```
In [5]: images = mnist.data.reshape((-1, 28, 28))
        plt.imshow(images[0],cmap=plt.cm.gray_r,interpolation="nearest")
        plt.show()
```



D'après l'image on peut visualiser le nombre 0, est-ce le cas ? Pour savoir la réponse on peut afficher la donnée descriptive convenable

```
In [6]: print(mnist.target[0])
```

```
0.0
```

1.1.2 Exploration d'autres jeux de données

A l'aide des mêmes étapes on peut visualiser d'autre DATASET par exemple le jeu de données WINE fourni par SKLEARN

In [7]:
```
from tabulate import tabulate

wine = datasets.load_wine()

values = wine.data
target = wine.target

newValues = []
newfeature = []
for i in range(5):
    newfeature.append(wine.feature_names[i])
newfeature.append("Target")

for j in range(len(values)):
    if j <= 15 :
        content = []
        for i in range(len(values[j])):
            if i < 5:
                content.append(values[j][i])
        content.append(target[j])
        newValues.append(content)

print(tabulate(newValues, headers=newfeature,tablefmt="grid"))
```

```
+-----------+-------------+-------+---------------------+-------------+----------+
|   alcohol |   malic_acid |   ash |   alcalinity_of_ash |   magnesium |   Target |
+===========+=============+=======+=====================+=============+==========+
|     14.23 |        1.71 |  2.43 |                15.6 |         127 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     13.2  |        1.78 |  2.14 |                11.2 |         100 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     13.16 |        2.36 |  2.67 |                18.6 |         101 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.37 |        1.95 |  2.5  |                16.8 |         113 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     13.24 |        2.59 |  2.87 |                21   |         118 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.2  |        1.76 |  2.45 |                15.2 |         112 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.39 |        1.87 |  2.45 |                14.6 |          96 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.06 |        2.15 |  2.61 |                17.6 |         121 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.83 |        1.64 |  2.17 |                14   |          97 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     13.86 |        1.35 |  2.27 |                16   |          98 |        0 |
+-----------+-------------+-------+---------------------+-------------+----------+
|     14.1  |        2.16 |  2.3  |                18   |         105 |        0 |
```

```
+-----------+-------------+-------+--------------------+------------+----------+
|     14.12 |        1.48 |  2.32 |               16.8 |         95 |        0 |
+-----------+-------------+-------+--------------------+------------+----------+
|     13.75 |        1.73 |  2.41 |               16   |         89 |        0 |
+-----------+-------------+-------+--------------------+------------+----------+
|     14.75 |        1.73 |  2.39 |               11.4 |         91 |        0 |
+-----------+-------------+-------+--------------------+------------+----------+
|     14.38 |        1.87 |  2.38 |               12   |        102 |        0 |
+-----------+-------------+-------+--------------------+------------+----------+
|     13.63 |        1.81 |  2.7  |               17.2 |        112 |        0 |
+-----------+-------------+-------+--------------------+------------+----------+
```

1.2 La méthode des k-nn

Dans cette partie on va commencer notre premier algorithme d'apprentissage superviser qui utilise le principe du plus proche voisin.

1.2.1 Apprentissage

Tout d'abord on va couper notre jeu donné en deux parties ; la partie qui va servir pour l'apprentissage et l'autre pour l'entrainement. Pour cela on va utiliser la méthode ń model_selection ż avec une liste d'indice aléatoire pour bien mélanger nos données

```
In [27]: data = np.random.randint(70000,size=5000)
         xtrain,xtest,ytrain,ytest = model_selection.train_test_split(mnist.data[data],mnist.ta
```

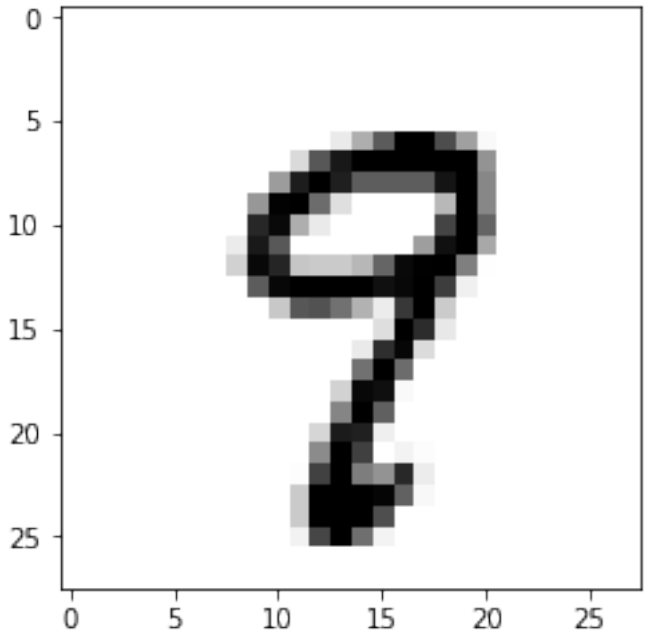
Maintenant on va créer un classifieur de type KNeighbors avec 10 voisins et ensuite on va l'entrainer.

```
In [9]: clf = neighbors.KNeighborsClassifier(n_neighbors=10)

        clf.fit(xtrain,ytrain)

Out[9]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=10, p=2,
                     weights='uniform')
```

Ensuite on va prédire l'image 4 de note dataset et la comparer avec la valeur cible.

```
In [10]: predicted = clf.predict(xtest[4].reshape(1, -1))

         print(predicted)

         images = xtest.reshape((-1, 28, 28))

         plt.imshow(images[4],cmap=plt.cm.gray_r,interpolation="nearest")
         plt.show()
```

```
[9.]
```



On affiche le score de notre classifieur et aussi son taux d'erreur. Le taux d'erreur est de 6% ce qui est logique car on n'a pas fourni un grand nombre d'entré pour l'apprentissage.

```
In [11]: print("Score :",clf.score(xtest, ytest))

         ypredTest = clf.predict(xtest)

         print("Erreur :",metrics.zero_one_loss(ytest, ypredTest))
```

```
Score : 0.932
Erreur : 0.06799999999999995
```

1.2.2 Evaluation

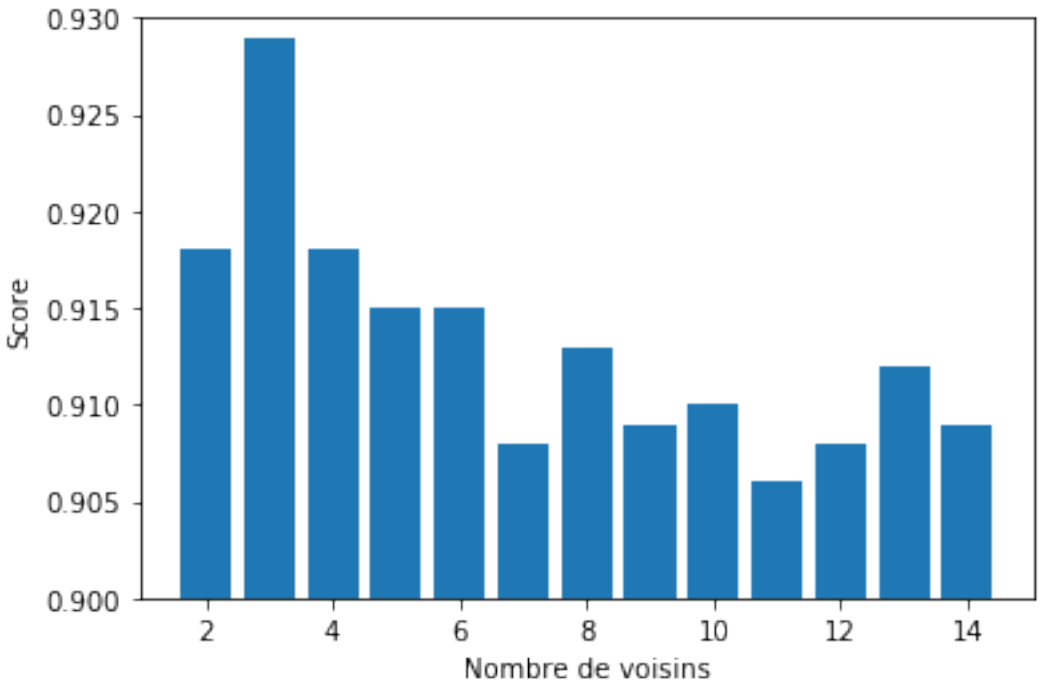
Pour voire l'impact du nombre des voisins sur la précision du classifieur on effectue le code suivant.

```
In [28]: scores = []

         for k in range(2,15):
             clf = neighbors.KNeighborsClassifier(n_neighbors=k)

             clf.fit(xtrain,ytrain)
```

```
        score = clf.score(xtest, ytest)

        scores.append(score)

        print("Score for ",k,' : ',score)
```

```
Score for  2  :  0.918
Score for  3  :  0.929
Score for  4  :  0.918
Score for  5  :  0.915
Score for  6  :  0.915
Score for  7  :  0.908
Score for  8  :  0.913
Score for  9  :  0.909
Score for  10  :  0.91
Score for  11  :  0.906
Score for  12  :  0.908
Score for  13  :  0.912
Score for  14  :  0.909
```

In [30]:
```
plt.bar(range(2,15),scores)
plt.xlabel('Nombre de voisins')
plt.ylabel('Score')
plt.ylim(0.9, 0.93)
plt.show()
```



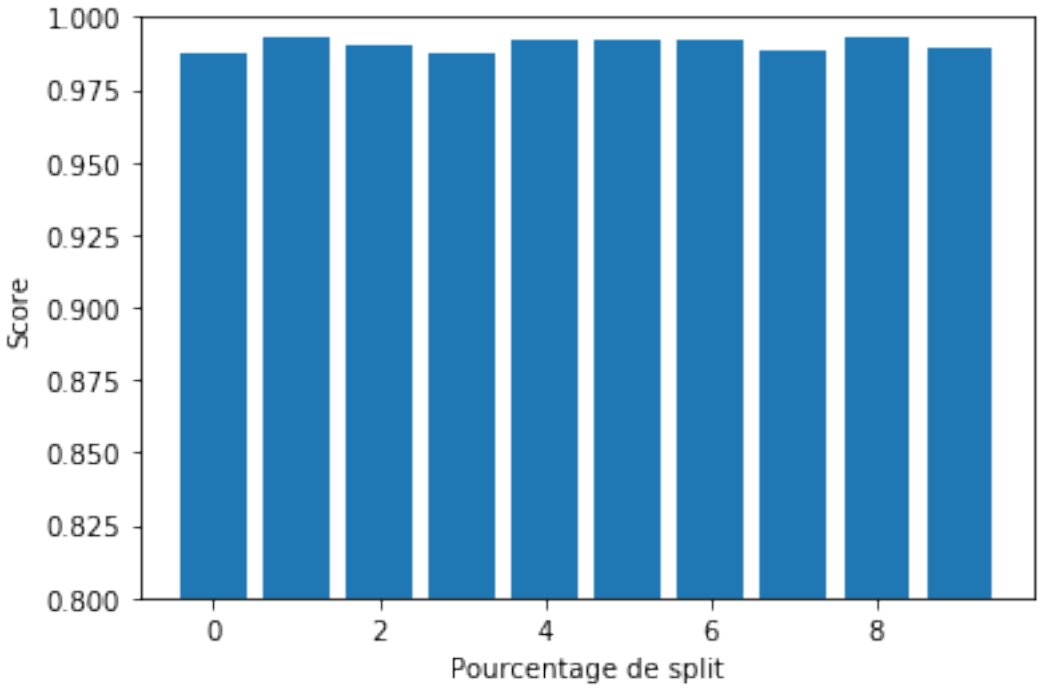
On va voir maintenant est ce que la manière avec la quelle on découpe notre dataset influe sur précision.

```
In [14]: KF = model_selection.KFold(n_splits=10,shuffle=True)
         clf = neighbors.KNeighborsClassifier(n_neighbors=7)

         data = np.random.randint(70000,size=15000)
         X = mnist.data[data]

         scoresKfold = []
         for train_index, test_index in KF.split(X):
             clf.fit(mnist.data[train_index],mnist.target[train_index])
             score = clf.score(mnist.data[test_index], mnist.target[test_index])

             scoresKfold.append(score)

             print("Score for ",train_index,' : ',score)
```

```
Score for  [    0     1     2 ... 14996 14998 14999]  :  0.9873333333333333
Score for  [    0     2     3 ... 14997 14998 14999]  :  0.9926666666666667
Score for  [    0     1     2 ... 14997 14998 14999]  :  0.9906666666666667
Score for  [    0     1     2 ... 14996 14997 14999]  :  0.9873333333333333
Score for  [    0     1     2 ... 14997 14998 14999]  :  0.992
Score for  [    1     2     3 ... 14997 14998 14999]  :  0.992
Score for  [    0     1     3 ... 14997 14998 14999]  :  0.992
Score for  [    0     1     2 ... 14997 14998 14999]  :  0.9886666666666667
Score for  [    0     1     2 ... 14997 14998 14999]  :  0.9933333333333333
Score for  [    0     1     2 ... 14995 14997 14998]  :  0.9893333333333333
```

On peut constater que la manier avec la quelle on coupe ne joue pas un grand rôle.

```
In [15]: plt.bar(range(10),scoresKfold)
         plt.xlabel('Pourcentage de split')
         plt.ylabel('Score')
         plt.ylim(0.8, 1)
         plt.show()
```



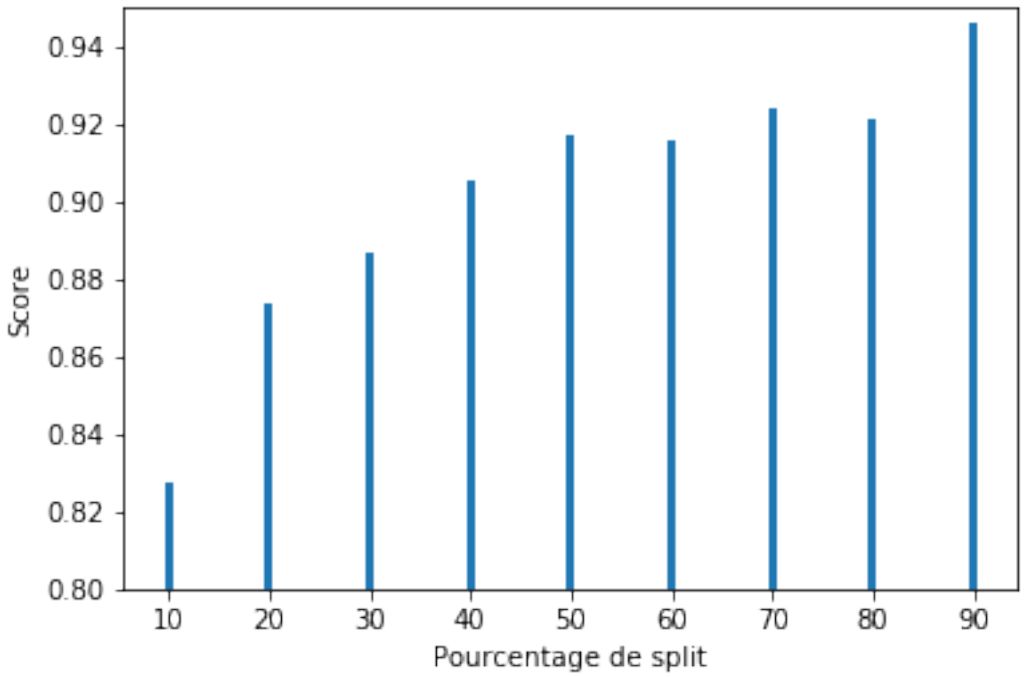
D'après ce qui précède on a pu voir qu'avec un nombre de voisinage de 3 on a un bon score. Maintenant avec cette même valeur on va varier le pourcentage des échantillant.

```
In [31]: data = np.random.randint(70000,size=5000)

         scoresTraine = []
         for percentage in np.arange(0.1, 1.0, 0.1):
             xtrain,xtest,ytrain,ytest = model_selection.train_test_split(mnist.data[data],mnis
             clf = neighbors.KNeighborsClassifier(n_neighbors=3)
             clf.fit(xtrain,ytrain)
             score = clf.score(xtest, ytest)

             scoresTraine.append(score)

             print("Score for ",percentage,' : ',score)
```

```
Score for  0.1  :  0.8275555555555556
Score for  0.2  :  0.8735
Score for  0.30000000000000004  :  0.8868571428571429
Score for  0.4  :  0.9053333333333333
Score for  0.5  :  0.9168
Score for  0.6  :  0.9155
Score for  0.7000000000000001  :  0.924
Score for  0.8  :  0.921
Score for  0.9  :  0.946
```

Tant qu'on ajoute un grand nombre d'entrées pour le test plus on obtient de meilleures précisions. On remarque que le pourcentage 90% donne un meilleur résultat.

```
In [33]: plt.bar(np.arange(10, 100, 10),scoresTraine)
         plt.xlabel('Pourcentage de split')
         plt.ylabel('Score')
         plt.ylim(0.8, 0.95)
         plt.show()
```



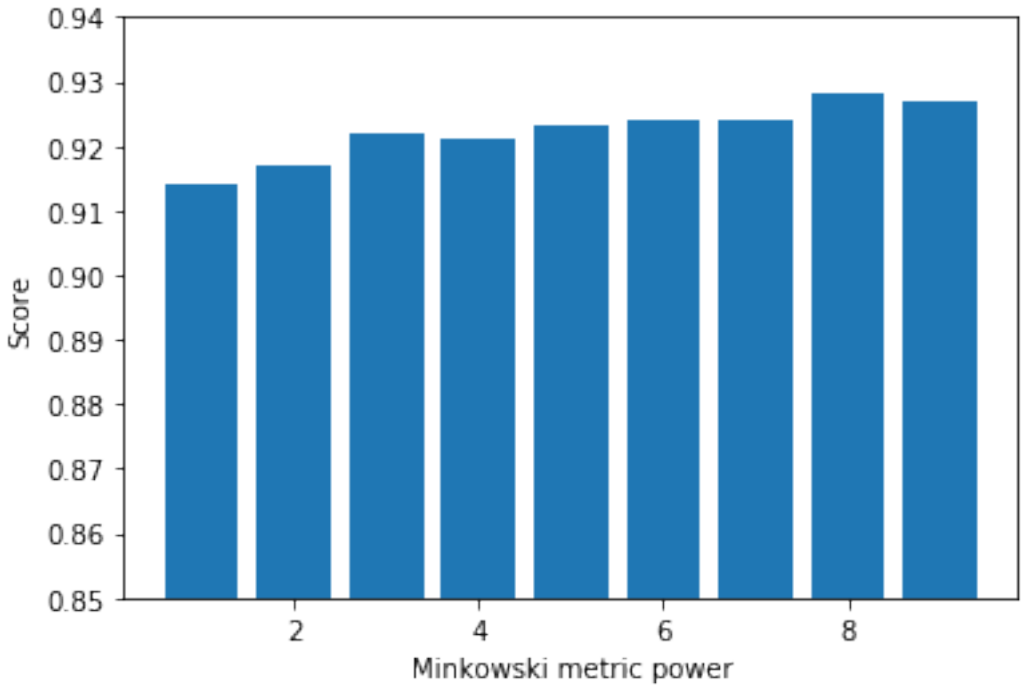
Maintenant on fait varier la valeur de la métrique de Minkowski.

```
In [18]: data = np.random.randint(70000,size=5000)
         xtrain,xtest,ytrain,ytest = model_selection.train_test_split(mnist.data[data],mnist.ta

         scoresMetric = []

         for m in range(1,10):
             clf = neighbors.KNeighborsClassifier(n_neighbors=3,p=m)

             clf.fit(xtrain,ytrain)

             score = clf.score(xtest, ytest)

             scoresMetric.append(score)

             print("Score for ",m,' : ',score)
```

```
Score for  1  :  0.914
Score for  2  :  0.917
Score for  3  :  0.922
Score for  4  :  0.921
Score for  5  :  0.923
Score for  6  :  0.924
Score for  7  :  0.924
Score for  8  :  0.928
Score for  9  :  0.927
```

Cette métrique permet de donner un bon résultat pour ses différentes grandes valeurs. Au plus de 8 on a toujours le même score.

```
In [19]: plt.bar(range(1,10),scoresMetric)
         plt.xlabel('Minkowski metric power')
         plt.ylabel('Score')
         plt.ylim(0.85, 0.94)
         plt.show()
```



Maintenant on fait varier la valeur de l'option n_job.

```
In [34]: import time

         data = np.random.randint(70000,size=5000)
         xtrain,xtest,ytrain,ytest = model_selection.train_test_split(mnist.data[data],mnist.ta
```

```
start = time.time()
clf = neighbors.KNeighborsClassifier(n_neighbors=3,n_jobs=1)
clf.fit(xtrain,ytrain)
end = time.time()
print("Pour 1 : ",end - start,"s")

start = time.time()
clf = neighbors.KNeighborsClassifier(n_neighbors=3,n_jobs=-1)
clf.fit(xtrain,ytrain)
end = time.time()
print("Pour -1 : ",end - start,"s")
```

```
Pour 1 :  0.23331975936889648 s
Pour -1 :  0.21465277671813965 s
```

1.2.3 Conclusion

D'après notre analyse de la méthode avec ses différentes options on remarque que cette dernière a un temps d'apprentissage qui est rapide et aussi simple à réaliser et manipuler.

En revanche la méthode est sensible vis-à-vis à la taille des jeux données, aussi que la prédiction prend un temps énorme.

TP2

December 9, 2018

1 Rapport de TP2 - ANN

Réaliser par Mohamed ELFILALI et Nguyen Duc Hau

1.1 Tester performance de Réseau de Neuronne avec différents paramètres

L'Objectif du TP est de trouver les paramètres les plus pertinents de ANN (**Artificial Neural Network** - Réseau de Neuronne Artificiel)

```
In [1]: from sklearn import datasets, model_selection
        from sklearn.metrics import precision_score
        import numpy as np
        from sklearn.neural_network import MLPClassifier
        import matplotlib.pyplot as plt
        from time import time
        from sklearn.metrics import accuracy_score

        import warnings;
        warnings.simplefilter('ignore')

        mnist = datasets.fetch_mldata('MNIST original')
```

Tout d'abord on va couper notre jeu donné en deux parties ; la partie qui va servir pour l'apprentissage et l'autre pour l'entrainement. Pour cela on va utiliser la méthode ń model_selection ż avec une liste d'indice aléatoire pour bien mélanger nos données

```
In [2]: index = np.random.randint(70000, size=49000)
        data = mnist.data[index]
        target = mnist.target[index]
```

Par la suite, on essaie de tester de manière exhautive les valeurs possibles de chaque paramètre. Chaque paramètre prends en compte la valeur optimum en temps d'exécution de paramtère analysé précédent pour but d'accélérer l'expérimentation.

1.1.1 Les couches de ANN

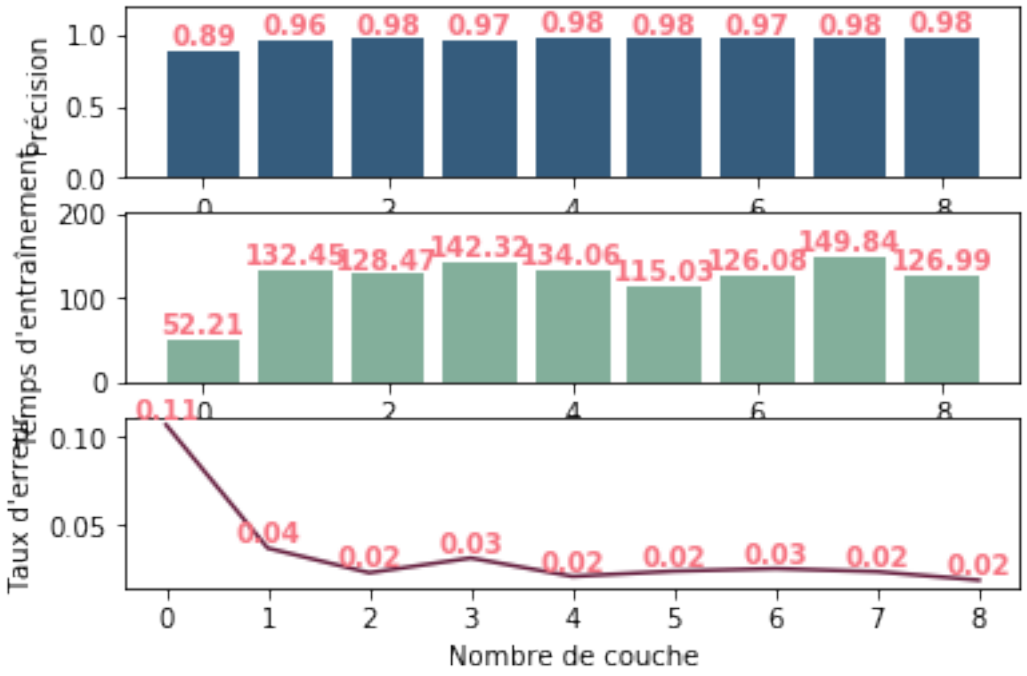
Pour des raisons de limite de calcul de machine personnelle, on se limite par tester le nombre de couche entre 1 et 10. Chaque couche dispose 50 neurone.

```
In [8]: max_l = 10
        min_l = 1
        precisions = np.zeros(max_l - min_l)
        error_rate = np.zeros(max_l - min_l)
        temps_execs = np.zeros(max_l - min_l)
        hidden_layer = (50,)*(max_l - min_l+1)

        # split donnée
        x_train, x_test, y_train, y_test = model_selection.train_test_split(
            data, target, train_size=0.8, test_size=0.2)

        for n_layer in range (max_l - min_l):


            # Init modèle
            MLP_model = MLPClassifier(hidden_layer_sizes = hidden_layer[0:n_layer])
            t_before = time()
            MLP_model.fit(x_train, y_train)
            t_after = time()

            y_predict = MLP_model.predict(x_test)

            temps_execs[n_layer] = t_after - t_before
            precisions[n_layer] = precision_score(y_test, y_predict, average='micro')
            error_rate[n_layer] = 1 - accuracy_score(y_test, y_predict)
            print('.', end='')
```

...

```
In [19]: plt.subplot(3, 1, 1)
         plt.bar(range(max_l - min_l) , precisions, color='#355C7D')
         plt.ylim(0,1.2)
         for i in range(max_l - min_l):
             plt.text(i, precisions[i],
                      "%.2f"%precisions[i],
                      fontweight='bold',
                      color='#F67280',
                      ha='center', va='bottom')
         plt.ylabel("Précision")

         plt.subplot(3, 1, 2)
         plt.bar(range(max_l - min_l) , temps_execs, color='#83AF9B')
         plt.ylim(0, max(temps_execs) + min(temps_execs))
         for i in range(max_l - min_l):
             plt.text(i, temps_execs[i],
                      "%.2f"%temps_execs[i],
                      fontweight='bold',
                      color='#F67280',
```

```
                ha='center',
                va='bottom')
plt.ylabel("Temps d'entraînement")

plt.subplot(3, 1, 3)
plt.plot(range(max_l - min_l) , error_rate, color='#651E3E')
for i in range(max_l - min_l):
    plt.text(i, error_rate[i],
                "%.2f"%error_rate[i],
                fontweight='bold',
                color='#F67280',
                ha='center',
                va='bottom')
plt.ylabel("Taux d'erreur")

plt.xlabel("Nombre de couche")
plt.show()

print("Meilleur nombre de couche pour la précision : ", np.argmax(precisions))
print("Meilleur nombre de couche pour le temps d'entraînement : ", np.argmin(temps_exe
```




```
Meilleur nombre de couche pour la précision :  8
Meilleur nombre de couche pour le temps d'entraînement :  0
```

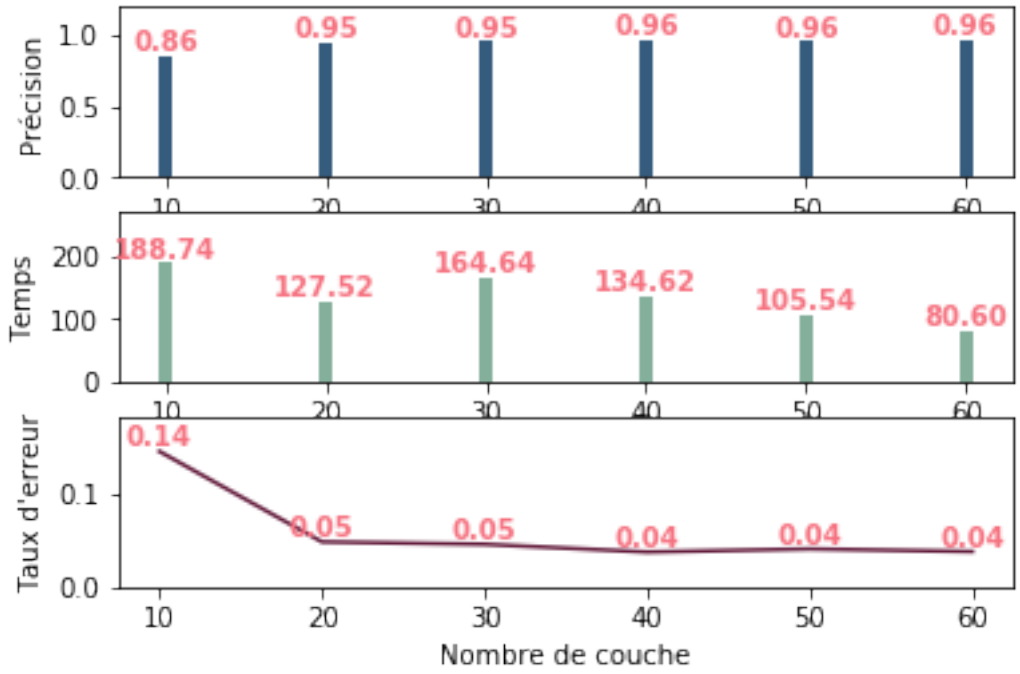
Testons maintenant les tailles différents des couches.

```
In [22]: couche_optimal = 2
         tailles = np.arange(10, 70, 10)
         size = tailles.size
         precisions = np.zeros(size)
         temps_execs = np.zeros(size)
         error_rate = np.zeros(size)

         # split donnée
         x_train, x_test, y_train, y_test = model_selection.train_test_split(
             data, target, train_size=0.8, test_size=0.2)

         for i in range(size):

             hidden_layer = (tailles[i],)*(couche_optimal)
             t_before = time()

             # Init modèle
             MLP_model = MLPClassifier(hidden_layer_sizes = hidden_layer)
             MLP_model.fit(x_train, y_train)

             t_after = time()
             y_predict = MLP_model.predict(x_test)

             temps_execs[i] = t_after - t_before
             precisions[i] = precision_score(y_test, y_predict, average='micro')
             error_rate[i] = 1 - accuracy_score(y_test, y_predict)
             print('.', end='')
```

...

```
In [28]: plt.subplot(3, 1, 1)
         plt.bar(tailles , precisions, color='#355C7D')
         plt.ylim(0,1.2)
         for i in range(size):
             plt.text(tailles[i], precisions[i],
                      "%.2f"%precisions[i], fontweight='bold',
                      color='#F67280', ha='center', va='bottom')
         plt.ylabel("Précision")

         plt.subplot(3, 1, 2)
         plt.bar(tailles , temps_execs, color='#83AF9B')
         plt.ylim(0, max(temps_execs) + min(temps_execs))
         for i in range(size):
             plt.text(tailles[i], temps_execs[i],
                      "%.2f"%temps_execs[i], fontweight='bold',
                      color='#F67280', ha='center', va='bottom')
```

```
plt.ylabel('Temps')

plt.subplot(3, 1, 3)
plt.plot(tailles , error_rate, color='#651E3E')
plt.ylim(0, max(error_rate) + min(error_rate))
for i in range(size):
    plt.text(tailles[i], error_rate[i],
             "%.2f"%error_rate[i], fontweight='bold',
             color='#F67280', ha='center', va='bottom')
plt.ylabel('Taux d\'erreur')

plt.xlabel("Nombre de couche")
plt.show()

print("Meilleur nombre de couche pour la précision : ",
      tailles[ np.argmin(precisions) ])
print("Meilleur nombre de couche pour le temps d'entraînement : ",
      tailles[ np.argmin(temps_execs) ])
```




```
Meilleur nombre de couche pour la précision :  10
Meilleur nombre de couche pour le temps d'entraînement :  60
```

1.1.2 Les algorithmes d'optimisation

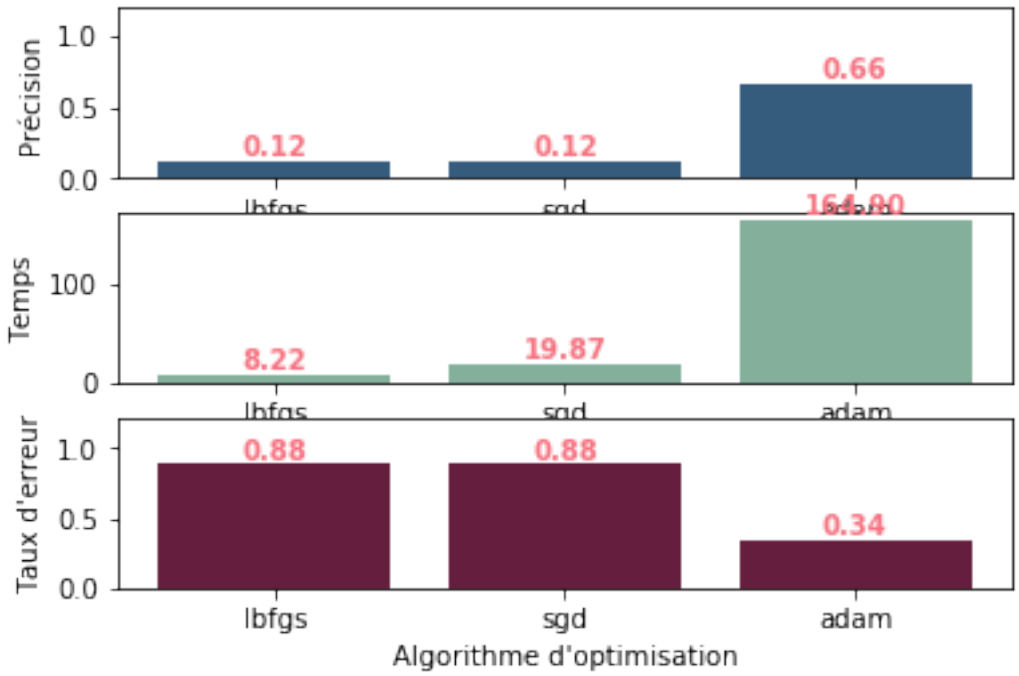
Le choix des algorithme d'optimisation est générallement pour le temps d'entraînement. Certain algorithme permets le temps de convergence vers l'optimum plus vite, mais risque de tomber dans l'optimum local ou impossible d'approcher le "vrai" optimum à partir de certain nombre de pas. D'autre convergente moins vite mais on est sûr d'avoir le moindre d'erreur.

```
In [31]: couche_optimal = 2
         size_optimal = 60
         hidden_layer_optimal = (couche_optimal)*(couche_optimal)

         solvers = ['lbfgs', 'sgd', 'adam']
         size = len(solvers)
         temps_execs = np.zeros(size)
         precisions = np.zeros(size)
         error_rate = np.zeros(size)

         # split donnée
         x_train, x_test, y_train, y_test = model_selection.train_test_split(
             data, target, train_size=0.8, test_size=0.2)

         for i in range(size):

             t_before = time()

             MLP_model = MLPClassifier(hidden_layer_sizes = hidden_layer_optimal,
                                       solver = solvers[i])
             MLP_model.fit(x_train, y_train)

             t_after = time()
             y_predict = MLP_model.predict(x_test)

             temps_execs[i] = t_after - t_before
             precisions[i] = precision_score(y_test, y_predict, average='micro')
             error_rate[i] = 1 - accuracy_score(y_test, y_predict)
             print('.', end='')
```

...

```
In [35]: plt.subplot(3, 1, 1)
         plt.bar(solvers , precisions, color='#355C7D')
         plt.ylim(0,1.2)
         for i in range(size):
             plt.text(solvers[i], precisions[i],
                      "%.2f"%precisions[i], fontweight='bold',
                      color='#F67280', ha='center', va='bottom')
         plt.ylabel("Précision")

         plt.subplot(3, 1, 2)
```

```
plt.bar(solvers , temps_execs, color='#83AF9B')
plt.ylim(0, max(temps_execs) + min(temps_execs))
for i in range(size):
    plt.text(solvers[i], temps_execs[i],
             "%.2f"%temps_execs[i], fontweight='bold',
             color='#F67280', ha='center', va='bottom')
plt.ylabel('Temps')

plt.subplot(3, 1, 3)
plt.bar(solvers , error_rate, color='#651E3E')
plt.ylim(0, max(error_rate) + min(error_rate))
for i in range(size):
    plt.text(solvers[i], error_rate[i],
             "%.2f"%error_rate[i], fontweight='bold',
             color='#F67280', ha='center', va='bottom')
plt.ylabel('Taux d\'erreur')

plt.xlabel("Algorithme d'optimisation")
plt.show()

print("Meilleur solver pour la précision : ",
      solvers[ np.argmax(precisions) ])
print("Meilleur solver pour le temps d'entraînement : ",
      solvers[ np.argmin(temps_execs) ])
```



```
Meilleur solver pour la précision :  adam
Meilleur solver pour le temps d'entraînement :  lbfgs
```

1.1.3 Fonction d'activation

Les fonctions d'activation n'affecteront pas beaucoup sur le temps de calcul mais surtout sur la précision du résultat final

```
In [37]: couche_optimal = 2
         size_optimal = 60
         hidden_layer_optimal = (couche_optimal)*(couche_optimal)
         solver_optimal = 'adam'

         activations = ['identity', 'logistic', 'tanh', 'relu']
         size = len(activations)
         temps_execs = np.zeros(size)
         precisions = np.zeros(size)
         error_rate = np.zeros(size)

         # split donnée
         x_train, x_test, y_train, y_test = model_selection.train_test_split(
             data, target, train_size=0.8, test_size=0.2)

         for i in range(size):

             MLP_model = MLPClassifier(hidden_layer_sizes = hidden_layer_optimal,
                                       solver=solver_optimal,
                                       activation = activations[i])
             t_before = time()
             MLP_model.fit(x_train, y_train)

             t_after = time()
             y_predict = MLP_model.predict(x_test)

             temps_execs[i] = t_after - t_before
             precisions[i] = precision_score(y_test, y_predict, average='micro')
             error_rate[i] = 1 - accuracy_score(y_test, y_predict)
             print('.', end='')
```

...

```
In [39]: plt.subplot(3, 1, 1)
         plt.bar(activations , precisions, color='#355C7D')
         plt.ylim(0,1.2)
         for i in range(size):
             plt.text(i, precisions[i],
                      "%.2f"%precisions[i], fontweight='bold',
                      color='#F67280', ha='center', va='bottom')
```

```
plt.ylabel('Précision')

plt.subplot(3, 1, 2)
plt.bar(activations , temps_execs, color='#83AF9B')
plt.ylim(0, max(temps_execs) + min(temps_execs))
for i in range(size):
    plt.text(activations[i], temps_execs[i],
             "%.2f"%temps_execs[i], fontweight='bold',
             color='#F67280', ha='center', va='bottom')
plt.ylabel('Temps')

plt.subplot(3, 1, 3)
plt.bar(activations , error_rate, color='#651E3E')
plt.ylim(0, max(error_rate) + min(error_rate))
for i in range(size):
    plt.text(activations[i], error_rate[i],
             "%.2f"%error_rate[i], fontweight='bold',
             color='#F67280', ha='center', va='bottom')
plt.ylabel('Taux d\'erreur')

plt.xlabel("Fonction d'activation")
plt.show()

print("Meilleur activateur pour la précision : ",
      activations[ np.argmax(precisions) ])
print("Meilleur activateur pour le temps d'entraînement : ",
      activations[ np.argmin(temps_execs) ])
```

```
Meilleur activateur pour la précision :  identity
Meilleur activateur pour le temps d'entraînement :  tanh
```

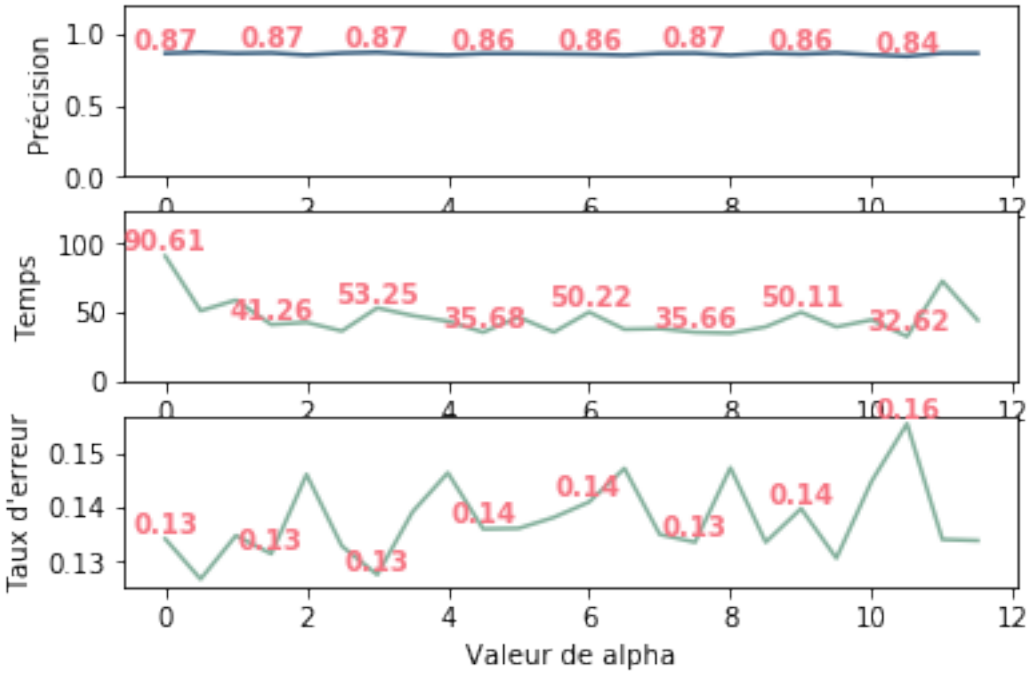
1.1.4 La régularisation L2 (paramètre)

```
In [67]: couche_optimal = 2
         size_optimal = 60
         hidden_layer_optimal = (couche_optimal)*(couche_optimal)
         solver_optimal = 'adam'
         activateur_optimal = 'identity'

         x_train, x_test, y_train, y_test = model_selection.train_test_split(
             data, target, train_size=0.8, test_size=0.2)
         alphas      = np.arange(0, 12, 0.5)
         size        = alphas.size
         temps_execs = np.zeros(size)
         precisions  = np.zeros(size)
         error_rate  = np.zeros(size)

         # split donnée
         x_train, x_test, y_train, y_test = model_selection.train_test_split(
             data, target, train_size=0.8, test_size=0.2)

         for i in range(size):

             t_before = time()

             MLP_model = MLPClassifier(hidden_layer_sizes = hidden_layer_optimal,
                                       solver = solver_optimal,
                                       activation = activateur_optimal,
                                       alpha = alphas[i])
             MLP_model.fit(x_train, y_train)

             t_after = time()
             y_predict = MLP_model.predict(x_test)

             precisions[i] = precision_score(y_test, y_predict, average='micro')
             temps_execs[i] = t_after - t_before
             error_rate[i] = 1 - accuracy_score(y_test, y_predict)
             print('.', end='')
```

...

```
In [71]: size = alphas.size
```

```
plt.subplot(3, 1, 1)
plt.plot(alphas , precisions, color='#355C7D')
plt.ylim(0,1.2)
for i in range(size):
    if(i % 3 == 0):
        plt.text(alphas[i], precisions[i],
                 "%.2f"%precisions[i], fontweight='bold',
                 color='#F67280', ha='center', va='bottom')
plt.ylabel('Précision')

plt.subplot(3, 1, 2)
plt.plot(alphas , temps_execs, color='#83AF9B')
plt.ylim(0, max(temps_execs) + min(temps_execs))
for i in range(size):
    if(i % 3 == 0):
        plt.text(alphas[i], temps_execs[i],
                 "%.2f"%temps_execs[i], fontweight='bold',
                 color='#F67280', ha='center', va='bottom')
plt.ylabel('Temps')

plt.subplot(3, 1, 3)
plt.plot(alphas , error_rate, color='#83AF9B')
for i in range(size):
    if(i % 3 == 0):
        plt.text(alphas[i], error_rate[i],
                 "%.2f"%error_rate[i], fontweight='bold',
                 color='#F67280', ha='center', va='bottom')
plt.ylabel('Taux d\'erreur')

plt.xlabel("Valeur de alpha")
plt.show()

print("Meilleur alpha pour la précision : ",
      alphas[ np.argmax(precisions) ])
print("Meilleur alpha pour le temps d'entraînement : ",
      alphas[ np.argmin(temps_execs) ])
```




```
Meilleur alpha pour la précision :  0.5
Meilleur alpha pour le temps d'entraînement :  10.5
```

1.2 Modèle le plus otimisé

Pour le problème de MNIST, les paramètre pour obtenir un compromise entre temps de calcul et précision sont: - Nombre de couche: 2 - 4 - Size de chaque couche: 50 - 60 - Algorithme d'optimisation: adam - Fonction d'activation: identity - Régularisation L2: 0.5 <= alpha <= 2

Un modèle favorise un seul axe est très facile à décider: pour favoriser le temps de calcul on met en place le moindre des paramètres, pour favoriser la précision, on tente à mettre un grande nombre de couche et toutes en grande taille. Mais ces choix ne sont pas pratiques.

1.3 Conclusion

ANN a ses avantages et ses inconvénients:

1.3.1 Avantages:

- Cette méthode permets de modéliser n'importe quelle problème (en théorie, si on dispose les ressources à l'infinie, les données à l'infinie et capacité calcul à l'infinie).
- Elle permets également d'obtenir un résultat plus précis par rapport aux autres méthodes.
- On peut passer à l'échelle cette méthode. Les neuronne dans réseau accepte le calcul indépendant, donc favorisé pour déployer au système distribué.

1.3.2 Inconvénients:

- **ANN** demande un capacité de calcul très coûteux, parfois moins efficaces par rapport à 2 autres méthodes. (Pour le même niveau de précision, ANN a besoin plus de temps pour l'apprentissage)
- Le choix des paramètres de **ANN** n'est pas évident, on ne peut pas non plus faire un test exhautive faute de son calcul gourmand.
- Pour l'apprentissage en temps réel ou en adaptation au changement du comportement, cette méthode n'est pas un choix idéal, vu impossible. C'est dû à la conception d'algorithmique: dans la phase d'apprentissage chaque couche doit attendre le résultat de celle précédente. Et ensuite dans l'étape de backpropagation, il faut calcul à chaque couche. Donc même avec les calculs distribués, on ne bénéfice pas grandement le parallélisme afin de booster le temps de calcul.

TP3

December 9, 2018

1 Rapport du TP3

Réalisé par : Mohamed ELFILALI et Nguyen Duc Hau

1.1 Apprentissage

Dans ce TP on va utiliser le classifieur SVM de la bibliothèque scikit-learn comme outil d'apprentissage superviser.

```
In [1]: from sklearn import datasets
        import matplotlib.pyplot as plt
        from sklearn import datasets
        from sklearn import model_selection
        from sklearn.svm import SVC
        import numpy as np
        from sklearn import metrics

        import warnings;
        warnings.simplefilter('ignore')

        mnist = datasets.fetch_mldata('MNIST original')
```

Tout d'abord on va couper notre jeu donné en deux parties ; la partie qui va servir pour l'apprentissage et l'autre pour l'entrainement. Pour cela on va utiliser la méthode ń model_selection ż avec une liste d'indice aléatoire pour bien mélanger nos données

```
In [2]: data = np.random.randint(70000,size=5000)
        xtrain,xtest,ytrain,ytest = model_selection.train_test_split(mnist.data[data],mnist.ta
```

Maintenant on va créer un classifieur de type SVM avec un noyau "Linear" et ensuite on va l'entrainer.

```
In [8]: model = SVC(kernel='linear')

        model.fit(xtrain,ytrain)

        score = model.score(xtest,ytest)

        print("Score :",score)
```

```
Score 0.8993333333333333
```

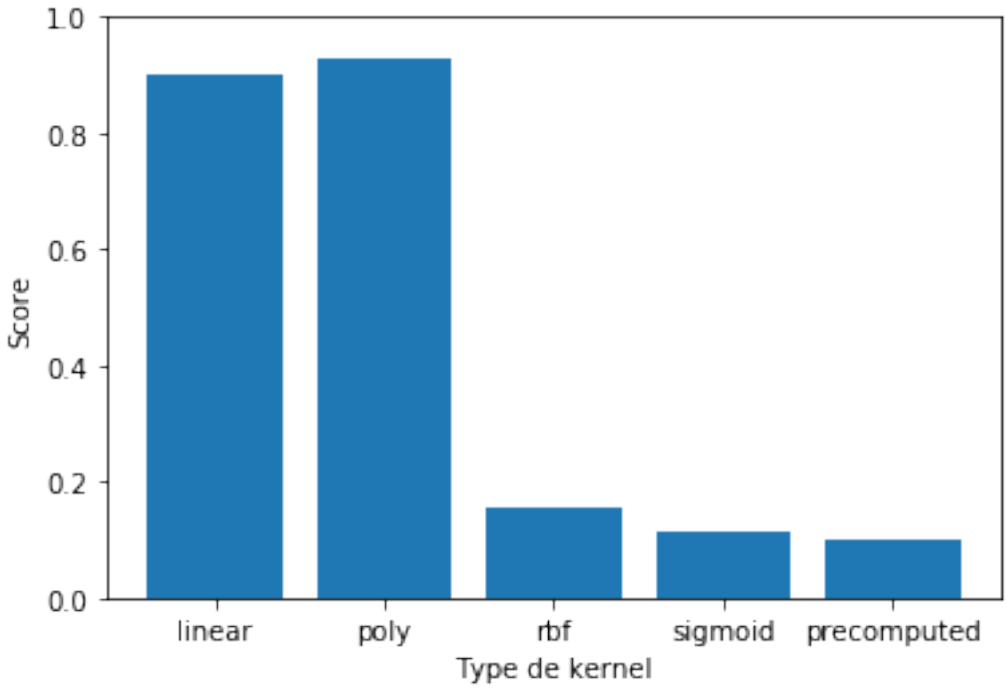
Pour bien visualiser l'impact du noyau choisit sur le classifieur on exécute le code suivant.

```
In [11]: scores = []

         for n in ['linear','poly','rbf','sigmoid']:
             model = SVC(kernel=n)

             model.fit(xtrain,ytrain)

             score = model.score(xtest,ytest)

             scores.append(score)

             print("Score avec",n,":",score)

         model = SVC(kernel='precomputed')

         kernel_train = np.dot(xtrain, xtrain.T)

         model.fit(kernel_train, ytrain)

         kernel_test = np.dot(xtest, xtrain.T)

         score = model.score(kernel_test,ytest)

         scores.append(score)

         print("Score avec precomputed :",score)
```

```
Score avec linear : 0.8993333333333333
Score avec poly : 0.9286666666666666
Score avec rbf : 0.15333333333333332
Score avec sigmoid : 0.11533333333333333
Score avec precomputed : 0.102
```

On remarque que les noyaux ń linear ż et ń poly ż donnent un bon score. Le noyau Poly est le meilleur entre eux.

```
In [12]: plt.bar(['linear','poly','rbf','sigmoid','precomputed'],scores)
         plt.xlabel('Type de kernel')
         plt.ylabel('Score')
         plt.ylim(0, 1.0)
         plt.show()
```

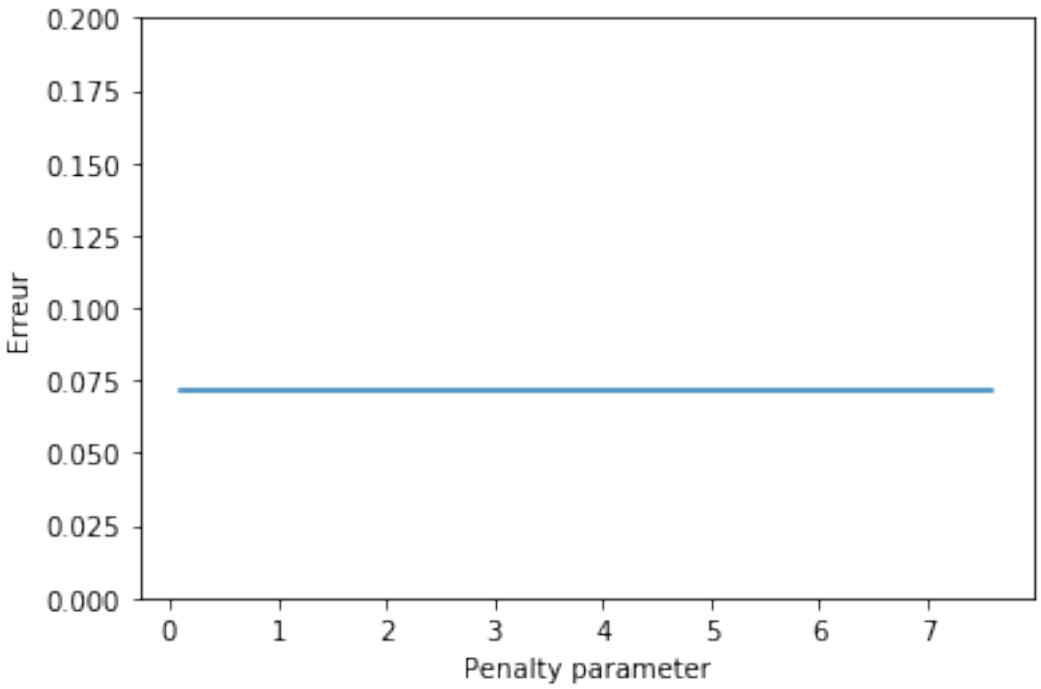


Maintenant on va faire varier le paramètre de tolérance aux erreurs et voir son impact

```
In [16]: errorsC = []

         for n in np.arange(0.1, 10, 2.5):
             model = SVC(C=n,kernel='poly')

             model.fit(xtrain,ytrain)

             ypredTest = model.predict(xtest)

             error = metrics.zero_one_loss(ytest, ypredTest)

             errorsC.append(error)

             print("Erreur avec c=",n,":",error)

Erreur avec c= 0.1 : 0.07133333333333336
Erreur avec c= 2.6 : 0.07133333333333336
Erreur avec c= 5.1 : 0.07133333333333336
Erreur avec c= 7.6 : 0.07133333333333336
```

D'après le graphe du résultat on constat que ce paramètre n'influence pas sur l'erreur du classifieur.

```
In [17]: plt.plot(np.arange(0.1, 10, 2.5),errorsC)
         plt.xlabel('Penalty parameter')
         plt.ylabel('Erreur')
         plt.ylim(0, 0.2)
         plt.show()
```



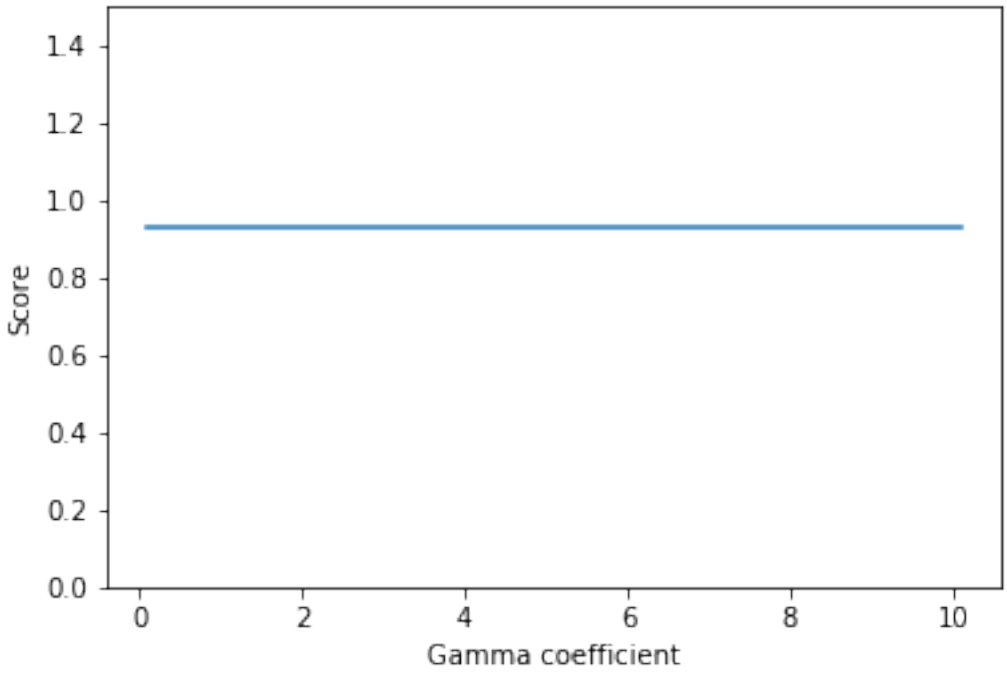
On fait la même chose pour le paramètre gamma et on constate qu'il n'a pas un impact sur l'apprentissage.

```
In [18]: scoresGama = []

         for n in np.arange(0.1, 12.5, 2.5):
             model = SVC(gamma=n,kernel='poly')

             model.fit(xtrain,ytrain)

             score = model.score(xtest,ytest)

             scoresGama.append(score)

             print("Score avec gamma=",n,":",score)
```

```
Score avec gamma= 0.1 : 0.9286666666666666
Score avec gamma= 2.6 : 0.9286666666666666
Score avec gamma= 5.1 : 0.9286666666666666
Score avec gamma= 7.6 : 0.9286666666666666
```

```
Score avec gamma= 10.1 : 0.9286666666666666
```

```
In [19]: plt.plot(np.arange(0.1, 12.5, 2.5),scoresGama)
         plt.xlabel('Gamma coefficient')
         plt.ylabel('Score')
         plt.ylim(0, 1.5)
         plt.show()
```



A ce point-là on a testé les différents paramètres du classifieur et comparé les résultats obtenus, maintenant à l'aide de la fonction GridSearchCV on va essayer de trouver les meilleures options.

```
In [3]: parameters = {'kernel':('linear','poly','rbf','sigmoid'), 'C':[1, 10],'gamma':[1,10]}
        model = SVC()
        clf = model_selection.GridSearchCV(model, parameters)

        clf.fit(xtrain,ytrain)
```

```
Out[3]: GridSearchCV(cv='warn', error_score='raise-deprecating',
              estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
          kernel='rbf', max_iter=-1, probability=False, random_state=None,
          shrinking=True, tol=0.001, verbose=False),
              fit_params=None, iid='warn', n_jobs=None,
              param_grid={'kernel': ('linear', 'poly', 'rbf', 'sigmoid'), 'C': [1, 10], 'gamma
              pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
              scoring=None, verbose=0)
```

On affiche maintenant les meilleurs paramètres.

```
In [4]: clf.best_estimator_
```

```
Out[4]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape='ovr', degree=3, gamma=1, kernel='poly',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

Pour avoir une meilleure vision sur l'ensemble des prédictions on affiche la matrice de confusion. Le résultat idéal est d'avoir une matrice diagonale

```
In [5]: ypredTest = clf.predict(xtest)
        cm = metrics.confusion_matrix(ytest,ypredTest)
```

```
In [6]: print(cm)
```

```
[[144   1   0   1   0   0   2   1   0   0]
 [  0 163   0   0   0   0   0   0   1   0]
 [  1   6 151   0   0   0   2   2   1   1]
 [  2   1   3 144   0   1   0   0   3   1]
 [  0   3   1   0 140   0   0   0   1   0]
 [  1   2   0   3   1 123   1   0   0   4]
 [  0   2   0   0   0   2 153   0   0   0]
 [  1   1   2   2   0   0   0 134   1   4]
 [  0   3   2   4   0   0   1   0 125   1]
 [  0   4   1   1   0   0   0   2   0 142]]
```

2 Conclusion

SVM a ses avantages et ses inconvénients:

2.0.1 Avantages:

- Cette méthode permets d'un taux d'erreur stable.
- Elle nécessite pas le temps de calcul important (les séparateurs sont hyperplans).
- Elle est efficace lors de la mise en production.
- **SVM** fonctionne correctement même pour les jeux de données ayant une grande dimension (beaucoup de caractéristiques - features), tandis que le nombre d'observation reste limité.

2.0.2 Inconvénients:

- **SVM** demande une connaissance profonde en statistique, des connaissances métiers afin de pouvoir choisir la fonction de noyau (kernel) adapté à la forme des données.
- Cette méthode a difficulté lors qu'on a un grand nombre d'observation: Elle a besoin le mémoire en cache pour résoudre le problème d'optimisation.
- Elle ne donne pas l'indice sur la pertinence des features.

Comparaison

December 10, 2018

1 Comparaison des 3 méthodes: k-NN, ANN et SVM

Réaliser par Mohamed ELFILALI et Nguyen Duc Hau

1.1 Contexte

Ce travail est réalisé dans le cadre du cours Apprentissage, ou on va expérimentées trois méthodes réparties en trois TP. Le jeu de donnée utilisé est **MNIST original**.

1.2 Comparaison

A partir de l'expérimentation (cf. **TP1.pdf**, **TP2.pdf**, **TP3.pdf**), nous en déduisons le tableau ci-dessous.

Critère	k-NN	ANN	SVM
Vitesse de convergence vers l'optimum	relativement vite	très lente	moyen
Réglementation pour bias et variance	paramètre **k**	paramètre **alpha**	paramètre **C**
Complexité de développement	Très facile à concevoir un modèle manuel	Très compliqué, surtout l'algorithme de backpropagation	Compliqué et demande une compréhension de données et de compétence mathématique pour implémenter les fonctions kernels
Complexité en production	Facile à suivre et debugger	Le fonctionnement dans les couches intermédiaire est une boîte noire donc obscure pour analyser le résultat	Si la nature des données est connue, cette méthode devient facile à ajuster les paramètres

Critère	k-NN	ANN	SVM
Intérêt	Facile à comprendre et mettre en place, haute vitesse de calcul	Apdaté aux problèmes de natures différents, donc intéressant pour les problèmes complexes	Permets d'obtenir un modèle ajusté très fin au problème, robust aux données bruitées et/ou à grande dimension
Inconvénient	Gourmande en espace de mémoire, prédiction lente, sensible aux features non pertinentes ou corrélés	Difficile à interpréter le processus d'apprentissage comme son fonctionnement reste un boîte noire	Difficile à traiter un jeu de données en grande nombre d'observation (car la matrice de Gram doit être stockée), difficile à choisir une fonction noyau sans avoir une expertise de métiers
Exemple d'application	Les problèmes convexes, données bruitées, pas besoin une précision élevée (système de recommandation, etc.)	Les problème très complexes pour modéliser, un grand jeu de données, bruitées possible, nécessité d'adaptation pour la prédiction soit traitée en temps réel, identification des features pertinents.	Les problèmes typiques pharmaceutiques: grande dimension alors que le nombre d'observation est peu, besoin d'un modèle sur mesure.

A noter que, l'intérêt et l'inconvénient de chaque méthode ont été détaillé dans les rapports spécifiques (cf. **TP1.pdf**, **TP2.pdf**, **TP3.pdf**). Ce tableau récapitule les grandes lignes de ces champs.

2 Annexe

Vous trouverez le code exécutable sur le lien github suivant : https://github.com/Kihansi95/apprentissage/tree/master/superviser