TP 1 : K-Nearest-Neighbours

In [2]:

```
#imports
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from time import time
import matplotlib.pyplot as plt
import numpy as np
```

In [3]:

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

Partie 1 : Exploration du jeu de données

1.1 Inspection de la structure du jeu de données

On analyse la nature et les dimensions des données à notre disposition. Le rôle et la valeur attendue est indiquée en commentaire pour chaque propriété

In [ ]:

```
#1.1
print(mnist) #représentation des paramètres du jeu de données, les images, leurs label
s, une description textuelle, les types de données...
#La sortie de cette commande a été retirée du rapport, au vu de la place prise par la t
otalité des informations
```

In [23]:

```
#et des informations supplémentaires de numpy (9 champs en tout)

print (mnist.data) #les images(chiffres) du dataset
print (mnist.target) #les labels(chiffre représenté par l'image) du dataset, 1 pour cha
que image
print (len(mnist.data)) #le nombre d'images du dataset
print (mnist.data.shape) #la dimension des images du dataset (70000 * 784)
print (mnist.target.shape) #la dimension des labels du dataset (70000 * 1)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
['5' '0' '4' ... '4' '5' '6']
70000
(70000, 784)
(70000,)
```

In [ ]:

```
print (mnist.data[0]) #la première image du dataset
#La sortie de cette commande a été retirée du rapport, au vu de la place prise par les
 784 pixels de l'image
```

In [22]:

```
print (mnist.data[0][1]) #le deuxième pixel de la première image du dataset
print (mnist.data[:,1]) #le deuxième pixel de chaque image du dataset
print (mnist.data[:100]) #les 100 premières images du dataset
```

```
0.0
[0. 0. 0. ... 0. 0. 0.]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

1.2 Affichage d'une image et du label associé

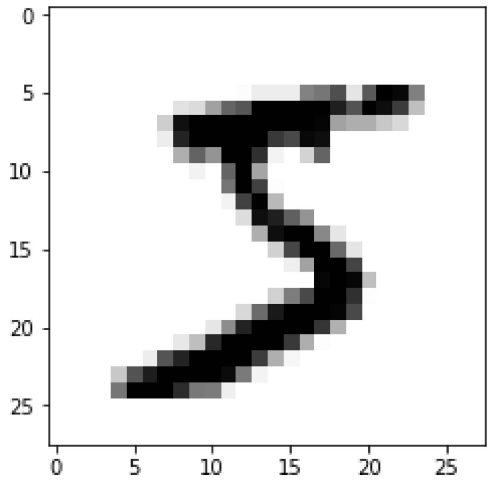
Plot d'une image :

- Puisque les images du dataset sont 'écrasées' en une dimension de 784 pixel, on les redimensionne d'abord au format ligne/colonne 28 * 28

In [5]:

```
img_index = 0
images = mnist.data.reshape((-1, 28, 28))
plt.imshow(images[img_index],cmap=plt.cm.gray_r,interpolation="nearest")
print("Associated label :",mnist.target[img_index])
```

```
Associated label : 5
```



1.3 Extraction de données aléatoires

Enfin, on utilise les fonctions numpy pour extraire un échantillon aléatoire depuis le jeu de donénes initial, qui servira pour la suite du TP

In [6]:

```
#choose sample size
dataset_length = len(mnist.data)
sample_size = 5000
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.targe
t[i] for i in sample_indexes])
```

Partie 2 : Méthode KNN

2.1 Premières classifications

Diviser la base de données à 80% pour l'apprentissage (training) et à 20% pour les tests

In [7]:

```
train_size = 0.8
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
)
```

Entrainer un classifieur k-nn avec k = 10 sur le jeu de données chargé.

In [ ]:

```
clf = KNeighborsClassifier(10)
clf.fit(d_train,l_train)
```

Afficher la classe de l'image 4 et sa classe prédite

In [9]:

```
pred_index = 3
prediction = clf.predict(([data[pred_index]]))[0] #predict method awaits a list of data
expected = target[pred_index]
print("prediction : " + str(prediction))
print("expected : " + str(expected))
```

```
prediction : 1
expected : 1
```

Afficher le score sur l'échantillon de test

In [10]:

```
score = clf.score(d_test, l_test)
print("score = " + str(score))
```

```
score = 0.922
```

Quel est le taux d'erreur sur vos données d'apprentissage ? Est-ce normal ?

In [11]:

```
score = clf.score(d_train, l_train)
print("score = " + str(score))
```

```
score = 0.93025
```

On obtient un taux d'erreur non-nul (score < 100%) C'était à attendre, puisque le modèle ne n'ajuste pas pour vérifier chaque échantillon du jeu de test, seulement un maximum d'entre eux

Faire varier le nombre de voisins (k) de 2 jusqu'à 15 et afficher le score.

Quel est le k optimal ?

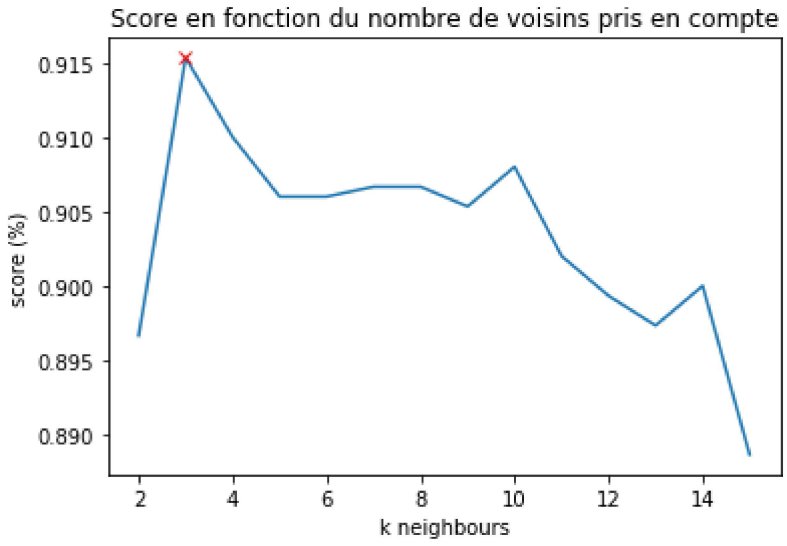
In [14]:

```
best_score = 0
best_nb = 0 #k optimal
train_size = 0.7
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
)
n_nb_set = range(2,16)
plot=([],[])

start = time() #TIMER START

for n_nb in n_nb_set:
    #print("Computing for",n_nb,train_size)
    clf = KNeighborsClassifier(n_nb)
    clf.fit(d_train     ,l_train)
    score = clf.score(d_test, l_test)
    #print("Score :",score)
    if score>best_score:
        best_nb = n_nb
        best_score = score
    plot[0].append(n_nb)
    plot[1].append(score)

duration = time() - start #TIMER END
```

In [15]:

```
fig,ax = plt.subplots()
ax.set_xlabel("k neighbours")
ax.set_ylabel("score (%)")
ax.set_title("Score en fonction du nombre de voisins pris en compte")
ax.plot(plot[0],plot[1])
ax.plot([best_nb],[best_score], marker= 'x', color='r')

print("Optimal k :", best_nb, "( score :",best_score,')')
print("Executed in",duration,'s')
```

```
Optimal k : 3 ( score : 0.9153333333333333 )
Executed in 145.55909419059753 s
```



Interprétation

Le nombre de voisins(k) optimal semble être 3, avec de bonnes performances jusqu'à 10 voisins. En répétant les analyses, l'optimum exact variait sur cette plage, mais on observe ensuite toujours une dégradation. 3 voisins sera donc la valeur retenue pour les prochains tests où l'on devra fixer ce paramètre.

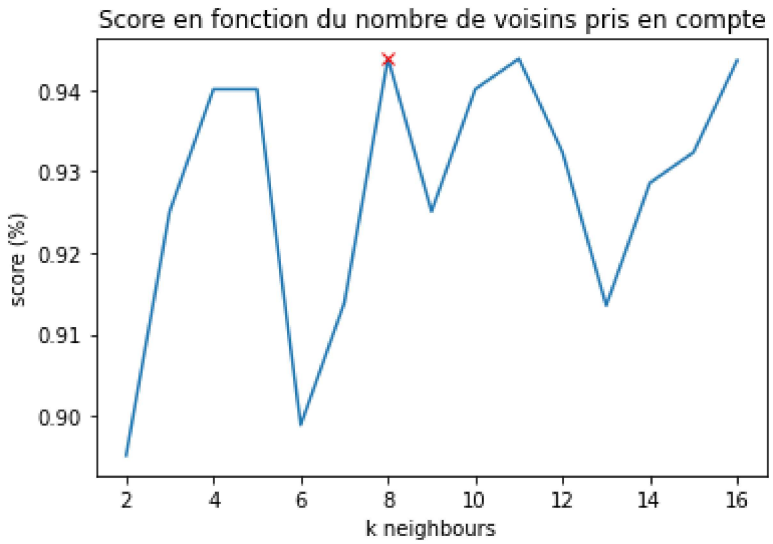
In [ ]:

```
best_score_fold = 0
best_nb_fold = 0 #k optimal
train_size = 0.8
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
)
plot_fold=([],[])
kf = KFold(n_splits=15, shuffle=True)

start = time() #TIMER START

n_nb =2
for train_index, test_index in kf.split(d_train, l_train):
    clf = KNeighborsClassifier(n_nb)
    clf.fit(d_train[train_index], l_train[train_index])
    score = clf.score(d_train[test_index], l_train[test_index])
    if score>best_score_fold:
        best_nb_fold = n_nb
        best_score_fold = score
    plot_fold[0].append(n_nb)
    plot_fold[1].append(score)
    n_nb += 1

duration = time() - start #TIMER END
```

In [27]:

```
fig,ax = plt.subplots()
ax.set_xlabel("k neighbours")
ax.set_ylabel("score (%)")
ax.set_title("Score en fonction du nombre de voisins pris en compte")
ax.plot(plot_fold[0],plot_fold[1])
ax.plot([best_nb_fold],[best_score_fold], marker= 'x', color='r')

print("Optimal k :", best_nb_fold, "( score :",best_score_fold,')')
print("Executed in",duration,'s')
```

```
Optimal k : 8 ( score : 0.9438202247191011 )
Executed in 26.24063539505005 s
```



Interprétation

La validation croisée confirme globalement les résultats précédents, même si le score 3 voisins apparaît désormais légèrement inférieur aux résultats de la plage autour de 10 voisins. On conservera le chiffre de 3 voisins.

Faites varier le pourcentage des échantillons (training et test) et affichez le score.

Quel est le pourcentage remarquable ?

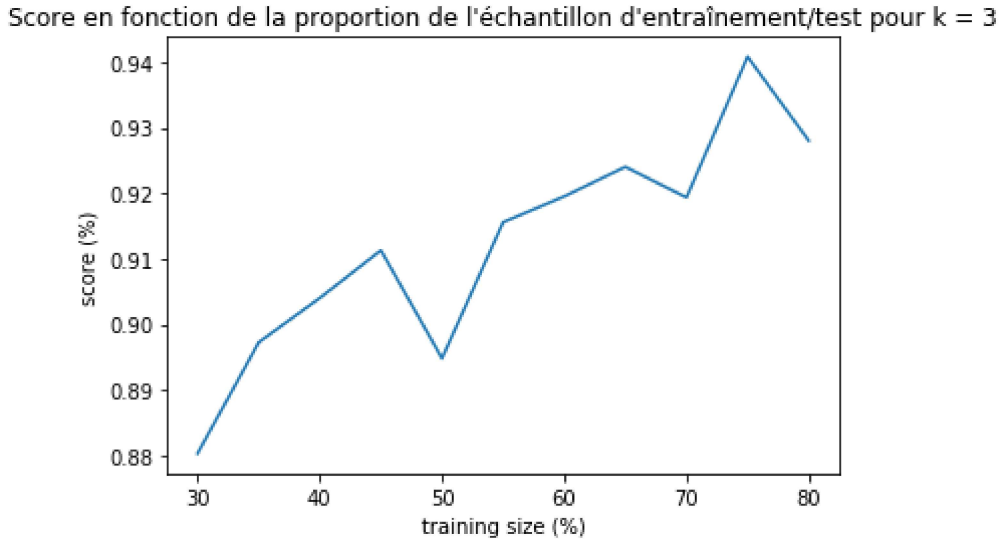
In [16]:

```
best_score = 0
best_size = 0 #training sample size
train_size_set = [i*0.05 for i in range(6,17)] #variation between 30-80 by 5% steps
n_nb = 3 #from previous estimations
plot=([],[])

start = time() #TIMER START
clf = KNeighborsClassifier(n_nb)
for train_size in train_size_set:
    d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_
size)
    clf.fit(d_train     ,l_train)
    score = clf.score(d_test, l_test)
    if score>best_score:
        best_score = score
        best_size = train_size
    plot[0].append(100*train_size)
    plot[1].append(score)

duration = time() - start #TIMER END
```

In [17]:

```
fig,ax = plt.subplots()
ax.set_xlabel("training size (%)")
ax.set_ylabel("score (%)")
ax.set_title("Score en fonction de la proportion de l'échantillon d'entraînement/test p
our k = 3")
ax.plot(plot[0],plot[1])

print("Optimal training proportion :",best_size, "( score :",best_score,')')
print("Executed in",duration,'s')
```

```
Optimal training proportion : 0.75 ( score : 0.9408 )
Executed in 126.49846696853638 s
```



On observe une croissance nette du score jusqu'à 65% du dataset utilisé pour l'entraînement. Le score croit ensuite de nouveau passé 75%, mais le pourcentage de données alors utilisées pour le test perd en signification : si l'on prenait 99% de données d'entraînement, le score serait probablement idéal mais le modèle serait moins pertinent une fois appliqué sur d'autres données. Un bon compromis efficacité/pertinence semble donc être 65% d'entraînement.

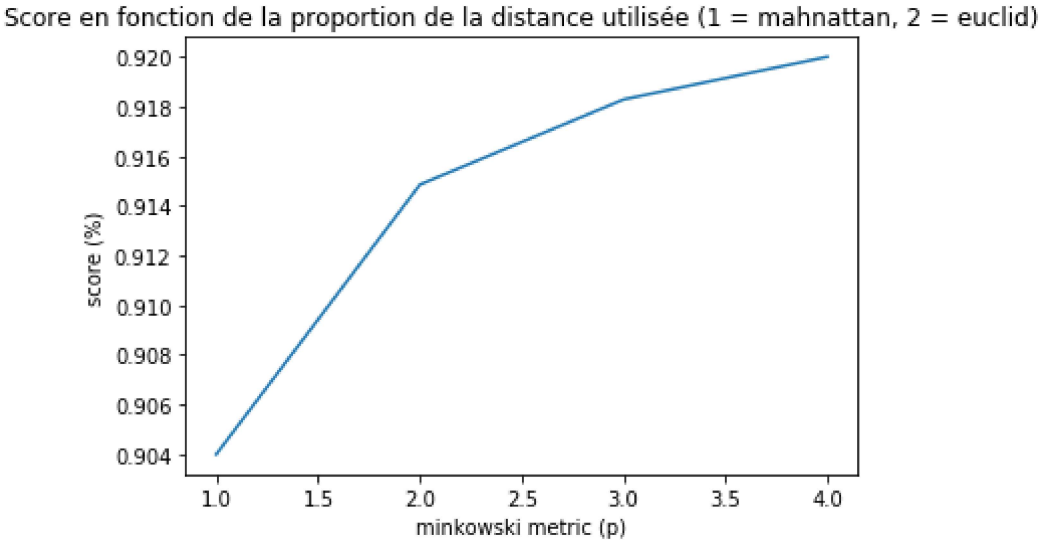
Faites varier les types de distances (p).

Quelle est la meilleure distance ?

In [19]:

```
best_dist = 0
best_score = 0
dist_set = [1,2,3,4]
train_size = 0.65 #from previous tests
n_nb = 3 #from previous tests
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
)
plot=([],[])

start = time() #TIMER START

for dist in dist_set:
    start_iter = time()
    clf = KNeighborsClassifier(n_nb, p = dist)
    clf.fit(d_train     ,l_train)
    score = clf.score(d_test, l_test)
    duration_iter = time()-start_iter
    print("Execution for metric",dist,":",duration_iter,"s")
    if score>best_score:
        best_score = score
        best_dist = dist
    plot[0].append(dist)
    plot[1].append(score)

duration = time() - start #TIMER END

fig,ax = plt.subplots()
ax.set_xlabel("minkowski metric (p)")
ax.set_ylabel("score (%)")
ax.set_title("Score en fonction de la proportion de la distance utilisée (1 = mahnatta
n, 2 = euclid)")
ax.plot(plot[0],plot[1])
print("Optimal distance :",best_dist, "( score :",best_score,')')
print("Executed in",duration,'s')
```

```
Execution for metric 1 : 10.08952808380127 s
Execution for metric 2 : 11.254179954528809 s
Execution for metric 3 : 121.73934960365295 s
Execution for metric 4 : 119.02021265029907 s
Optimal distance : 4 ( score : 0.92 )
Executed in 262.1052532196045 s
```



Le meilleur score est donc oobtenu pour la distance de minkowski 4, et la tendance est croissante avec la métrique utilisée. En revanche, les temps d'exécution explosent (x10) lorsque l'on sort des deux métriques classiques manhattan et euclidienne (1 et 2 respectivement). La métrique pertinente semble donc être la distance euclidienne, compromis entre efficacité et rapidité.

Fixez n_job à 1 puis à -1 et calculez le temps de chacun.

In [20]:

```
train_size = 0.65 #from previous tests
n_nb = 3 #from previous tests
dist = 2
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
)


clfA = KNeighborsClassifier(n_nb, p = dist, n_jobs = 1)
clfB = KNeighborsClassifier(n_nb, p = dist, n_jobs = -1)

start = time()
clfA.fit(d_train, l_train)
score = clf.score(d_test, l_test)
duration = time() - start
print("Execution for n_jobs = 1:",duration,"s")

start = time()
clfB.fit(d_train, l_train)
score = clf.score(d_test, l_test)
duration = time() - start
print("Execution for n_jobs = -1:",duration,"s")
```

```
Execution for n_jobs = 1: 159.75563859939575 s
Execution for n_jobs = -1: 156.8400616645813 s
```

Interprétation :

Malheureusement, la fonctionnalité de répartir la prédiction sur plusieurs coeur n'offrait aucun avantage sur la machine à notre disposition sur ce TP, les temps de calcul étaient donc similaires. Dans le cas général, la majeure partie du temps d'exécution de KNN provient du calcul des distances entre les points à prédire et les points du jeu de données d'entraînement. Ces calculs étant indépendant, ils se prêtent parfaitement à l'optimisation par parallélisation sur une machine disposant effectivement de plusieurs coeurs ou sur plusieurs machines par grid computing.

Paramètres retenus

On retiendra donc les paramètres de 3 voisins et une métrique de distance euclidienne pour les algorithmes KNN.

Conclusion sur KNN

Les k-nn peuvent être de bons classifieurs mais dépendent lourdement de la nature du jeu de données et offrent des performances peu satisfaisantes en terme de temps de calcul, d'autant plus qu'il 'y a pas relement de division entre apprentissage et prédiction : l'apprentissage consistant seulement à stocker le jeu de données de référence, la complexité de l'algorithme est totalement dédiée à la prédiction.

Il est nécessaire de définir un jeu d'entraînement suffisamment petit et néanmoins représentatif. En effet, plus le jeu d'entraînement sera grand, plus le nombre de mesure de distance à effectuer pour une prédiction sera élevé.

De plus, la définition de la métrique de distance est cruciale et doit également être un compromis pertinence/temps de calcul car elle sera exécutée de nombreuse fois et représente la quasi totalité de la complexité de l'algorithme.

FIN TP 1