# TD Info 8 - Intelligence artificielle et apprentissage
# Proposition de correction

Préambule :

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from random import uniform
from math import sqrt
```

L'intelligence artificielle est aujourd'hui une notion très employée et synonyme d'efficacité et de beaucoup de promesses. Des progrès récents spectaculaires permettant atteindre des performances sur-humaines dans de nombreux domaines (étude d'images médicales, jeux de go, étude de repliement de molécules, …).

Prenons l'exemple des jeux à deux joueurs aux règles bien définies :

- En 1996, le grand maître des échecs *Kasparov* bat le supercalculateur d'IBM, *Deep Blue*. L'année suivante, l'ordinateur prend sa revanche. Sa façon de jouer consiste à évaluer, avec de la *force brute* (et aussi beaucoup de raffinements), les différents coups pouvant être joués, et leurs conséquences à long terme, ainsi qu'on a pu le voir dans le TD précédent. Ses performances sont donc directement liées à la puissance de calcul et au temps de calcul disponible. La puissance de calcul de l'ordinateur utilisé à l'époque était de l'ordre de la puissance actuellement disponible dans une smartwatch….
- Le jeu de Go est un jeu offrant beaucoup plus de possibilités aux joueurs : ainsi, pendant longtemps, on a pensé que les joueurs humains, et leur façon de jouer *intuitive*, battrait toujours l'ordinateur. Cependant, en 2016, le programme d'intelligence artificielle *AlphaGo*, de Google, bat un des meilleurs joueur du monde *Lee Sedol*. Cette victoire marque un tournant symbolique : alors que Kasparov avait été battu par la *force brute* d'un code informatique rédigé par des informaticiens, code **figé**; Lee Sedol a été battu par un programme capable d'**apprendre** (en utilisant des réseaux neuronaux) et de **faire évoluer son comportement** et ses réponses aux coup adverses au fur et à mesure des parties jouées. La différence est fondamentale : comme l'explique l'un des créateurs d'AlphaGo, "*Bien que nous ayons programmé cette machine, nous n'avons aucune idée des coups qu'elle va jouer. Ces coups sont un phénomène émergent de son apprentissage. Nous avons seulement créé la base de données et les algorithmes d'apprentissage. Mais les coups que trouve le programme nous échappent, et sont d'ailleurs bien meilleurs que ceux que nous pourrions trouver en tant que joueurs de go*". Depuis, les grands maîtres s'inspire des coups joués par cette intelligence artificielle, coups parfois qualifiés de *créatifs* et *inspirés*.

- Aujourd'hui, les programmes d'intelligence artificielle règnent aussi sur les échecs, et de nombreux autres jeux plus complexes (en particulier quelques jeux vidéos).

L'intelligence artificielle repose, comme l'intelligence humaine, sur l'apprentissage à partir de données. Cet apprentissage pourra être **supervisé** ou **non-supervisé**.

Les algorithmes que nous allons étudier ici sont assez anciens et simples, mais permettent cependant, avec les puissances de calculs et les données disponibles aujourd'hui, d'applications intéressantes.

# 1 Apprentissage supervisé

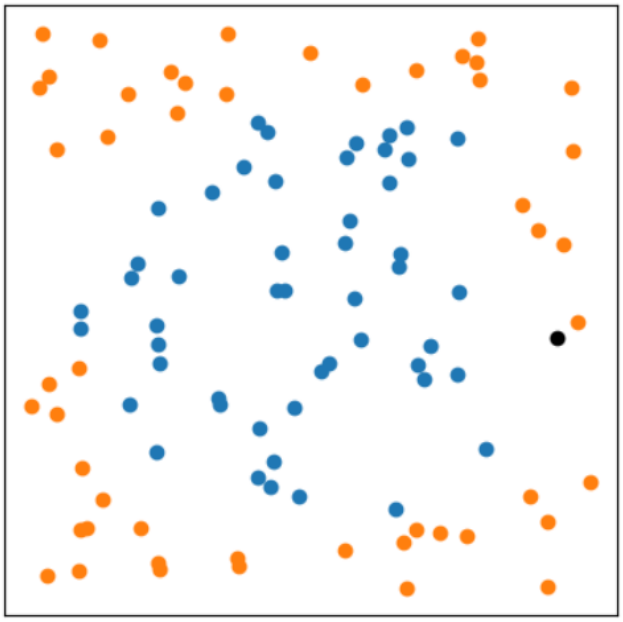
On distingue deux type d'apprentissage : le plus fréquent, l'apprentissage supervisé, consiste pour un opérateur à montrer à la machines des milliers voire des millions d'exemples étiquetés avec leur catégorie, qui permettront à la machine de déterminer elle-même les paramètres pertinents pour classer chaque objet dans la catégorie qui lui correspond. Une fois cette phase d'apprentissage terminée, la machine doit être capable de généraliser à des objets pas encore vu.

## 1.1 Présentation de l'exemple d'étude

Commençons avec un exemple simple : nous allons considérer un ensemble de points, qui peuvent appartenir à deux catégories :

- soit le point est de couleur bleu
- soit le point est de couleur orange

On présente ces points (ici 100 points bleus ou oranges) :



A ces 100 points, on a ajouté un dernier point, noir. La question à laquelle doit répondre l'algorithme est la suivante : ce point doit-il être colorié en bleu ou en orange ?

Les 100 points bleus ou oranges sont les données "d'entrainement" : on a fourni à l'algorithme des points pour lesquels on connait la couleur à attribuer. Il doit lui en déduire, à partir de ces données

d'entraînement, la couleur à donner au point noir.

**Question 1 :** Quelle couleur voulez-vous attribuer à ce point ? Pourquoi ? Comment raisonnez-vous ?

En réalité, les points bleus sont coloriés en bleu car ils appartiennent tous à l'intérieur d'un disque. Plus précisement, chaque point possède des coordonnées $(x, y)$ telles que $-1 < x < 1$ et $-1 < y < 1$ (chaque point est à l'intérieur du carré de côté 2, centré sur l'origine), et :

- si le point est à l'intérieur du cercle de centre $O$, de rayon $0,8$; on lui attribue la couleur bleu,
- sinon le point est en-dehors de ce cercle, et on lui attribue la couleur orange.

Ces points seront tirés aléatoirement, et constitueront les données d'entraînements : nous allons les fournir au programme d'intelligence artificielle, en étiquetant la catégorie (à l'intérieur ou à l'extérieur du cercle).

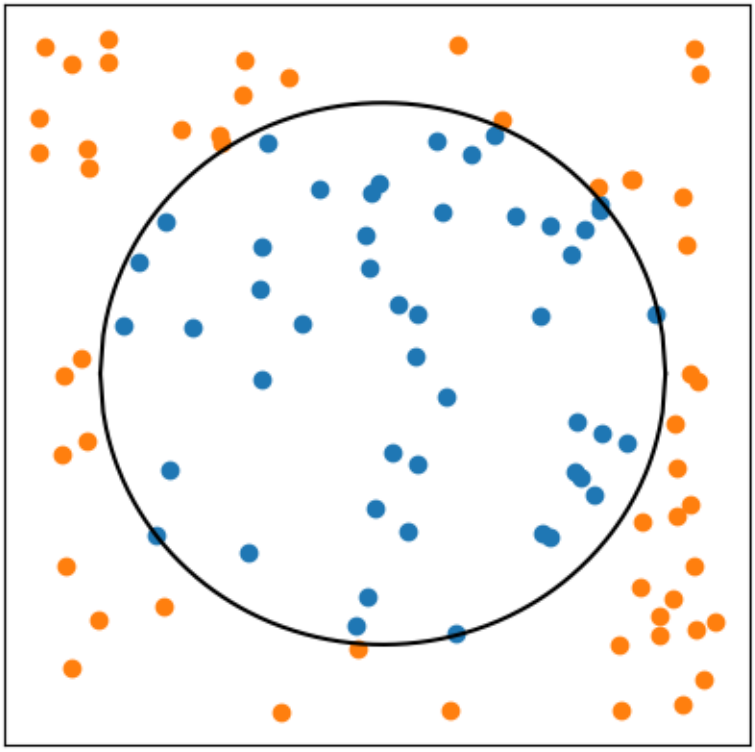
**Question 2 :** Écrire une fonction `donnees_ent(n_samp)` générant `n_samp` données d'entraînement. Cette fonction retournera un dictionnaire ayant pour clefs les tuples `(x, y)` des coordonnées des `n_samp` points, et pour valeur associée un booléen `True` ou `False` selon si le point associé est à l'intérieur du cercle ou non. On utilisera la fonction `uniform(-1,1)` pour générer un nombre aléatoire compris entre $-1$ et 1.

[2]:
```
def donnees_ent(n_samp):
    Dict_ent = dict()
    for i in range(n_samp):
        x, y =  uniform(-1,1), uniform(-1,1)
        if x**2 + y**2 < 0.8**2 :
            Dict_ent[(x, y)] = True
        else :
            Dict_ent[(x, y)] = False
    return Dict_ent
```

Représentons quelques données :

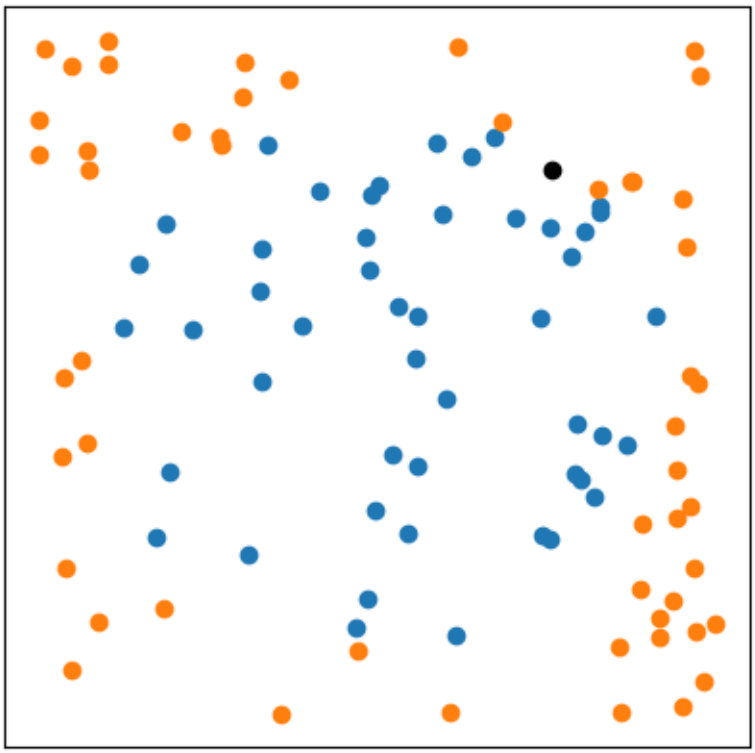
[3]:
```
Dict_ent = donnees_ent(100)

plt.figure(figsize=(5,5))

# Cercle :
x = np.linspace(-0.8, 0.8, 200)
plt.plot(x, np.sqrt(0.8**2 - x**2), color='k')
plt.plot(x, -np.sqrt(0.8**2 - x**2), color='k')

# Points :
X_in, X_out, Y_in, Y_out = [], [], [], []
for clef in Dict_ent :
    if Dict_ent[clef] :
        X_in.append(clef[0])
        Y_in.append(clef[1])
    else :
```

```
        X_out.append(clef[0])
        Y_out.append(clef[1])

plt.scatter(X_in, Y_in)
plt.scatter(X_out, Y_out)
plt.xticks([])
plt.yticks([])
plt.show()
```



Le but de notre algorithme sera d'étiqueter correctement un nouveau point généré aléatoirement :

[8]:
```
plt.figure(figsize=(5,5))
# Données entraînement :
plt.scatter(X_in, Y_in)
plt.scatter(X_out, Y_out)
plt.xticks([])
plt.yticks([])
# Nouveau point à étiquet :
x, y = uniform(-1, 1), uniform(-1, 1)
plt.scatter(x, y, color = 'k')
plt.show()
```



Écrire un algorithme permettant de déterminer la couleur réelle du point noir repose sur un apprentissage supervisé car on possède déjà des points d'entraînement dont on connaît la couleur. Une intervention humaine initial, pour le *bon étiquetage*, est ici indispensable !

**Attention, il est ici important de bien comprendre que pour déterminer la couleur du point noir, la règle sous-jacente d'étiquetage (bleu si dans le disque de rayon $0,8$) est inconnue !**

### 1.2 Algorithme des k plus proches voisins

Cet algorithme repose sur une idée intuitive : pour attribuer une couleur au point noir de test, il peut être utile de regarder la couleur des points d'entraînements qui lui sont voisin. Nous allons donc attribuer la couleur majoritaire des k plus proches voisins au point de test.

**Question 3 :** Écrire une fonction `dist_eucl(pointA, pointB)` qui retourne la distance euclidienne entre les points A et B, `pointA` et `pointB` étant des tuples contenant l'abscisse et l'ordonnée de ces points.

[9]:
```
def dist_eucl(pointA, pointB) :
    xA, yA = pointA
    xB, yB = pointB
    return sqrt((xB - xA)**2 + (yB - yA)**2)
```

**Question 4 :** Compléter le code de la fonction `coul_ppv(point_test, Dict_ent, k)`, qui, à partir d'un point test représenté par un tuple `point_test` (abscisse, ordonnée) et d'un dictionnaire de données d'entraînement tel que celui retourné par la fonction `donnees_ent`; retourne la couleur majoritaire (via une variable booléenne : `True` si bleu, `False` sinon) dans les k points d'entraînements

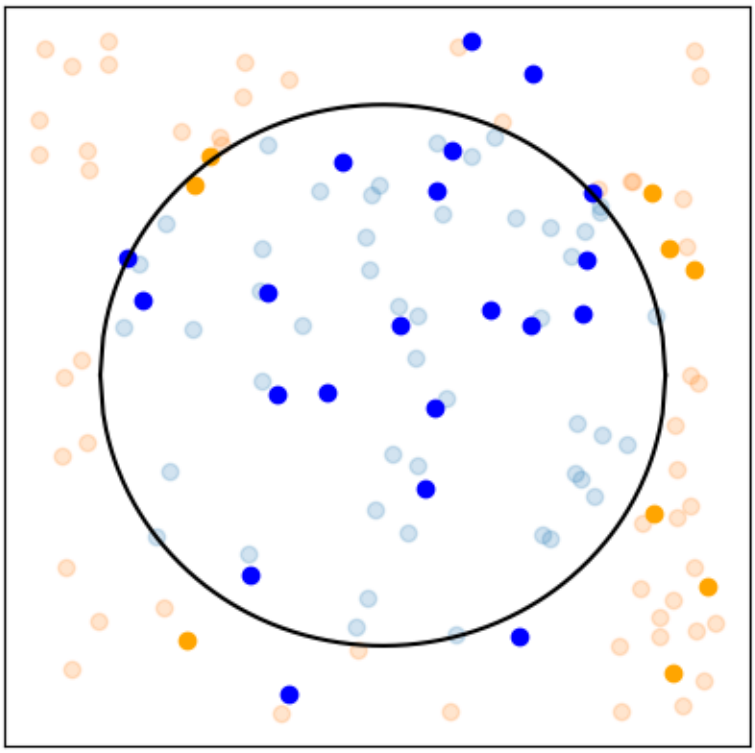
plus proches voisins du point test.

[10]:
```
def coul_ppv(point_test, Dict_ent, k) :
    # on demande un nombre k impair, pourqu'une couleur majoriataire se dégage !
    assert k%2 == 1

    # on construit le dictionnaire {distance eucl au point test : couleur}
    Dict_dist_coul = dict()
    for point in Dict_ent :
        Dict_dist_coul[dist_eucl(point,point_test)] = Dict_ent[point]

    # Liste des distances triées par ordre croissant :
    Dist_t = sorted(Dict_dist_coul.keys())

    # Détermination du nombre de point bleu parmi les k plus proches voisins :
    nb_bleu = 0
    for i in range(k) :
        coul = Dict_dist_coul[Dist_t[i]]
        if coul == True :
            nb_bleu += 1

    # Détermination de la couleur majoritaire :
    if nb_bleu > k//2 :
        return True
    else :
        return False
```

Vérifions le bon fonctionnement avec les 5 plus proches voisins considérés, pour 30 points test :

[12]:
```
plt.figure(figsize=(5,5))

# Cercle :
x = np.linspace(-0.8, 0.8, 200)
plt.plot(x, np.sqrt(0.8**2 - x**2), color='k')
plt.plot(x, -np.sqrt(0.8**2 - x**2), color='k')

# Points d'entraînement (en transparence) :
plt.scatter(X_in, Y_in, alpha = 0.2)
plt.scatter(X_out, Y_out, alpha = 0.2)
plt.xticks([])
plt.yticks([])

# 30 points test :
for i in range(30) :
    point_test = (uniform(-1, 1), uniform(-1, 1))
    if coul_ppv(point_test, Dict_ent, 5) == True :
        plt.scatter(point_test[0], point_test[1], color = 'b')
    else :
```

```
        plt.scatter(point_test[0], point_test[1], color = 'orange')
plt.show()
```



**Question 5 :** Commenter l'aspect de la figure ci-dessus, en particulier justifier les *erreurs* d'étiquetage faites par l'algorithme.

On peut bien évidemment tester d'autres configurations, avec plus de données de tests et/ou d'autres valeurs de k.

### 1.3 Évaluation et optimisation

Comme on a pu le remarquer, cet algorithme ne donne pas de résultats exacts, on parle alors d'**heuristique** : il s'agit d'une méthode pour solutionner un problème complexe de façon *simple* et *rapide*, mais pas nécessairement optimale et/ou exacte.

Évaluons la précision de notre algorithme des plus proches voisins : nous nommerons erreur le pourcentage de points test auxquels l'algorithme attribue la *mauvaise* couleur.

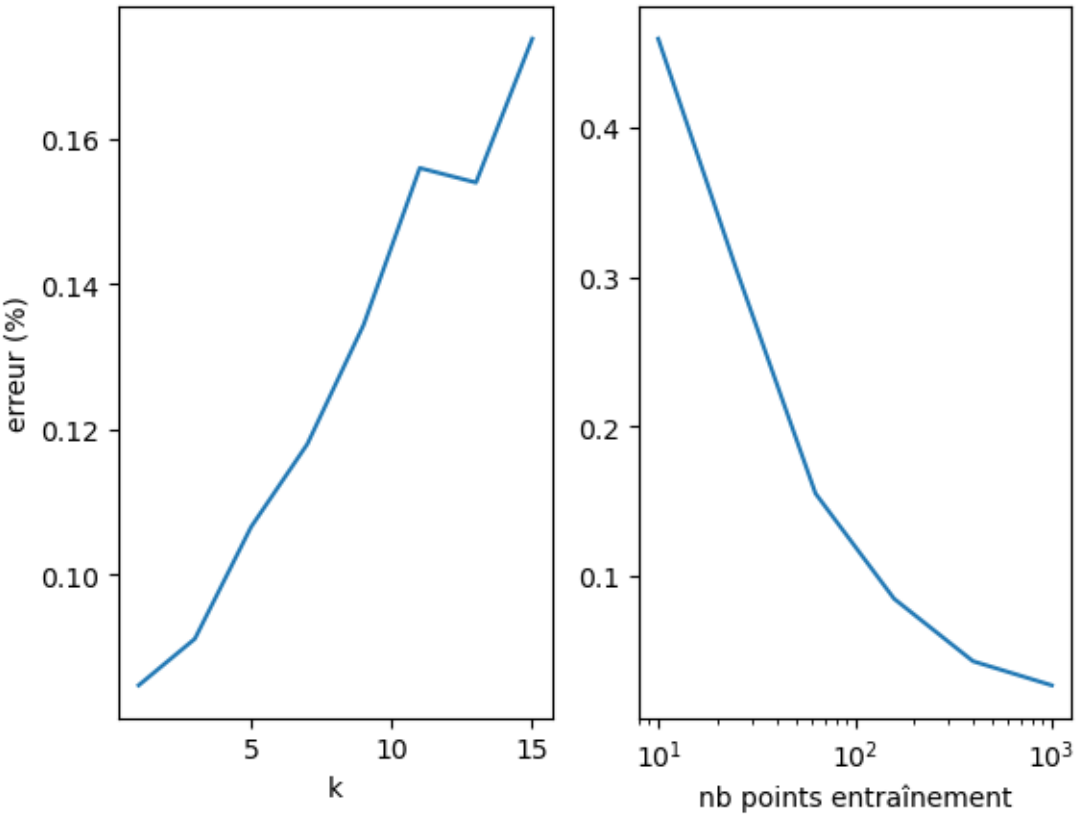
**Question 6 :** Compléter le code de la fonction `erreur_ppv(n_samp, n_ test, k)` retournant l'erreur pour `n_samp` points d'apprentissage, `n_test` points test et `k voisins considérés`.

[19]:
```
def erreur_ppv(n_samp, n_test, k) :

    erreur = 0

    Dict_ent = donnees_ent(n_samp)
```

```
    for i in range(n_test) :
        # Création du point test, placé aléatoirement :
        point_test = (uniform(-1, 1), uniform(-1, 1))
        # Int variable booléenne : True si point test à l'intérieur du disque :
        Int = point_test[0]**2 + point_test[1]**2 < 0.8**2
        # Test si erreur :
        if coul_ppv(point_test, Dict_ent, k) != Int :
            erreur += 1 / n_test
    return erreur
```

Évaluons alors l'effet du nombre de données de test et de k :

[20]:
```
plt.figure()

plt.subplot(121)
K = []
E = []

for k in range(1, 16, 2) :
    K.append(k)
    erreur_k = 0
    for i in range(20):
        erreur_k += erreur_ppv(100, 500, k)/20
    E.append(erreur_k)

plt.plot(K,E)
plt.xlabel('k')
plt.ylabel('erreur (%)')

plt.subplot(122)
N_samp = []
E = []

for n_samp in np.logspace(1,3,6):
    N_samp.append(int(n_samp))
    erreur_n = 0
    for i in range(20):
        erreur_n += erreur_ppv(int(n_samp), 500, 5)/50
    E.append(erreur_n)

plt.plot(N_samp,E)
plt.xlabel('nb points entraînement')
plt.xscale('log')
plt.show()
```




Pour évaluer la précision de notre algorithme de façon un peu plus précise, on peut définir la **matrice de confusion** : il s'agit de la matrice présentant dans notre exemple 2 lignes et 2 colonnes. Dans la première ligne, on présentera le pourcentage de points étiquetés bleu par notre algorithme : ceux à raison dans la première colonne, et ceux à tord dans la deuxième. Dans la deuxième ligne, on présentera le pourcentage de points étiquetés orange par notre algorithme : ceux à tord dans la première colonne, et ceux à raison dans la deuxième.

**Question 7 :** Idéalement, à quoi matrice doit ressembler cette matrice de confusion ? Utiliser le code fourni pour la calculer, et conclure.

Cette matrice doit posséder les valeurs les plus importantes possibles dans la diagonale.

[13]:
```
Mat_conf = [[0, 0], [0, 0]]
n_test = 500
n_samp = 100
k = 5

Dict_ent = donnees_ent(n_samp)

for i in range(n_test) :
    point_test = (uniform(-1, 1), uniform(-1, 1))
    Int = point_test[0]**2 + point_test[1]**2 < 0.8**2
    Int_ppv = coul_ppv(point_test, Dict_ent, k)
    if  Int == True and Int_ppv == True :
```

```
        Mat_conf[0][0] += 1 / n_test
    elif Int == False and Int_ppv == True :
        Mat_conf[0][1] += 1 / n_test
    elif Int == False and Int_ppv == False :
        Mat_conf[1][1] += 1 / n_test
    else :
        Mat_conf[1][0] += 1 / n_test
display(Mat_conf)
```

```
[[0.4260000000000003, 0.08200000000000006],
 [0.04600000000000003, 0.44600000000000034]]
```

**Question 8 :** Quels améliorations de cet algorithme des plus proches voisins pouvez-vous proposer ?

On peut proposer de raffiner la règle d'étiquetage, par exemple en :

- faisant une moyenne pondérée par la distance des couleurs des k plus proches voisins;
- considérant la couleur majoritaire des voisins situés à une distance maximale du point de test.

Remarque : si les données entraînement sont pas exactes ou biaisées (par l'opérateur), les résultats le seront aussi ! C'est une problématique actuelle des IA, pour lesquelles le choix des données d'entraînement est crucial.

### 1.4 Application : reconnaissance de chiffres manuscrits

Grâce au module `scikit-learn`, on dispose de 1797 images de 8px par 8px, en 256 nuances de gris, représentant des chiffres manuscrits de 0 à 9.

**Question 9 :** Exécuter le code fourni pour afficher 10 échantillons de chaque chiffre. Ces données ont aussi partagées : 80% ont été gardées pour l'apprentissage, et les 20% restant pour les tests. On affiche alors un échantillon de test pour chaque chiffre.

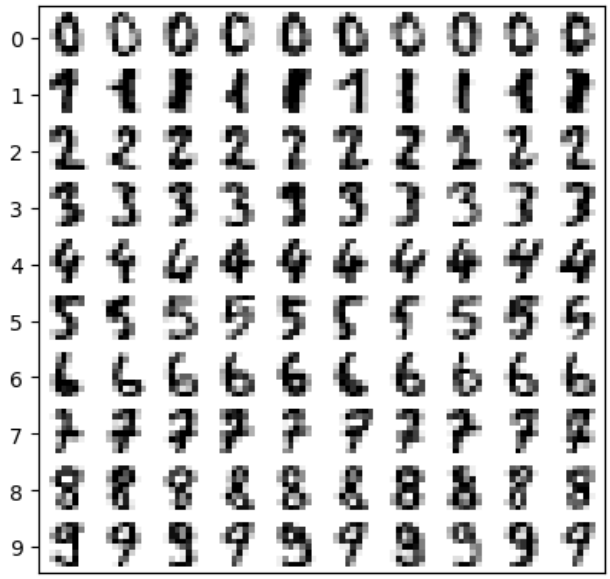
[14]:
```
digits = load_digits()
X, y = digits.data, digits.target
```

Visualisons 10 échantillons de chaque chiffre :

[15]:
```
def montrer_chiffres(X, y, limit_max=10):
    '''Montre des chiffres.'''
    labels, nombres = np.unique(y, return_counts=True)
    nombre_max = min(np.max(nombres), limit_max)
    img = np.zeros((100, nombre_max*10))
    for i in range(10):
        index_label = np.where(y == i)[0][:limit_max]
        for j, echantillon in enumerate(index_label):
            img[i*10+1:i*10+9,j*10+1:j*10+9] = X[echantillon].reshape((8, 8))
    plt.imshow(img, cmap='binary')
    plt.xticks([])
    plt.yticks(5 + 10*np.arange(10), np.arange(10))
```

[16]:
```
def obtenir_echantillons(X, y, n_echantillons=10):
    '''Donne une sélection aléatoire d'échantillons pour chaque classe.'''
    index = []
    for label in np.unique(y):
        index_label = np.where(y == label)[0]
        replace = len(index_label) > n_echantillons
        index += list(np.random.choice(index_label, size=n_echantillons,␣
 ↪replace=replace))
    return index
```

[17]:
```
index = obtenir_echantillons(X, y)
montrer_chiffres(X[index], y[index])
```




Partageons toutes ces données : gardons-en 80% pour l'apprentissage, et les 20% restant pour les tests :

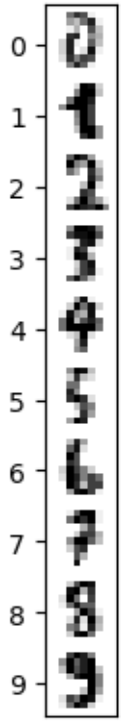
[18]:
```
def partager_apprentissage_test(X, y, ratio_test=0.2):
    '''Partage un jeu de données entre apprentissage et test.
    La répartition entre classes est conservée.'''
    index = []
    for label in np.unique(y):
        index_label = np.where(y==label)[0]
        size = int(ratio_test * len(index_label))
        replace = size < len(index_label)
        index += list(np.random.choice(index_label, size=size, replace=replace))
    X_test = X[index]
    y_test = y[index]
    index_ = np.ones(len(y), dtype=bool)
    index_[index] = False
    X_app = X[index_]
```

```
    y_app = y[index_]
    return X_app, X_test, y_app, y_test
```

[19]:
```
X_app, X_test, y_app, y_test = partager_apprentissage_test(X, y)
```

Observons un des échantillons à tester :

[20]:
```
index = obtenir_echantillons(X_test, y_test, 1)
montrer_chiffres(X_test[index], y_test[index])
```




**Question 10 :** Exécuter le code fourni pour la recherche des plus proches voisins de cet échantillon de test. Les 5 plus proches seront alors affichés.

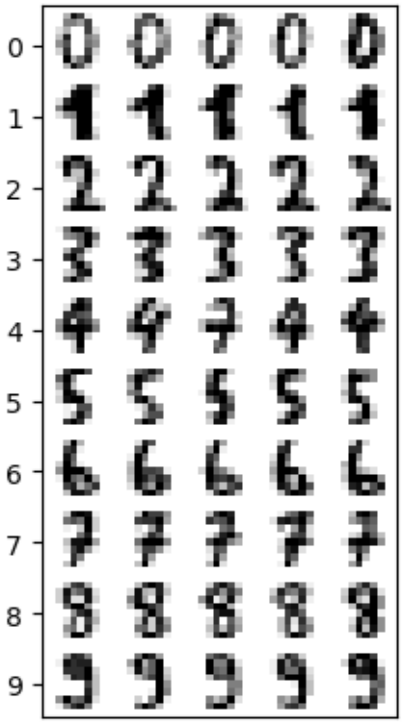
Recherchons alors les plus proches voisins de cet échantillon de test :

[21]:
```
def rechercher_plus_proches_voisins(X_test, X_app, n_voisins=5):
    '''Recherche des plus proches voisins.
    Retourne une matrice de taille (n_test, n_voisins).'''
    voisins = []
    for x in X_test:
        distances = np.linalg.norm(X_app - x, axis=1) # Calcul de la distance␣
↪carrée (norme 2) entre tous les points d'apprentissage et le point de test,␣
↪retourne un vecteur
        voisins.append(np.argpartition(distances, n_voisins)[:n_voisins]) # tri␣
↪et retourne que les n_voisins premiers voisins pour le point de test
    return np.array(voisins)
```

[22]:
```
voisins = rechercher_plus_proches_voisins(X_test[index], X_app)
```

Voici ces 5 plus proches voisins :

[23]:
```
index = voisins.flatten()
montrer_chiffres(X_app[index], np.repeat(np.arange(10), voisins.shape[1]))
```




**Question 11 :** Exécuter le code fourni pour étiqueter toutes les images de test, puis observer les erreurs d'étiquettage : l'affichage montre les chiffres et les étiquettes que le programme leurs attribués. Interpréter l'affichage final.

Étiquetons toutes les images de test :

[24]:
```
def classifier_plus_proches_voisins(X_test, X_app, y_app, n_voisins=5):
    '''Classification par plus proches voisins.
    Retourne la classe majoritaire.'''
    y_pred = []
    for x in X_test:
        distances = np.linalg.norm(X_app - x, axis=1)
        voisins = np.argpartition(distances, n_voisins)[:n_voisins]
        labels = y_app[voisins] # vecteur des classes à touver pour les données␣
↪d'apprentissage
        labels_unique, compteurs = np.unique(labels, return_counts=True)
        y_pred.append(labels_unique[np.argmax(compteurs)])
    return np.array(y_pred)
```

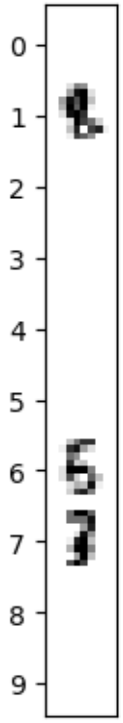
[25]:
```
y_pred = classifier_plus_proches_voisins(X_test, X_app, y_app)
```

Puis observons les erreurs d'étiquetage: :

[26]:
```
index = np.where(y_pred != y_test)[0]
```

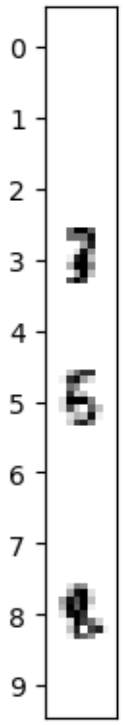
Commençons par les chiffres qu'on a mal classés ou étiqueté :

[28]:
```
montrer_chiffres(X_test[index], y_pred[index])
```




On peut alors comparer à leur "réelle" étiquette :

[27]:
```
montrer_chiffres(X_test[index], y_test[index])
```




**Question 12 :** A votre avis, comment est définie la distance euclidienne pour ces données ?

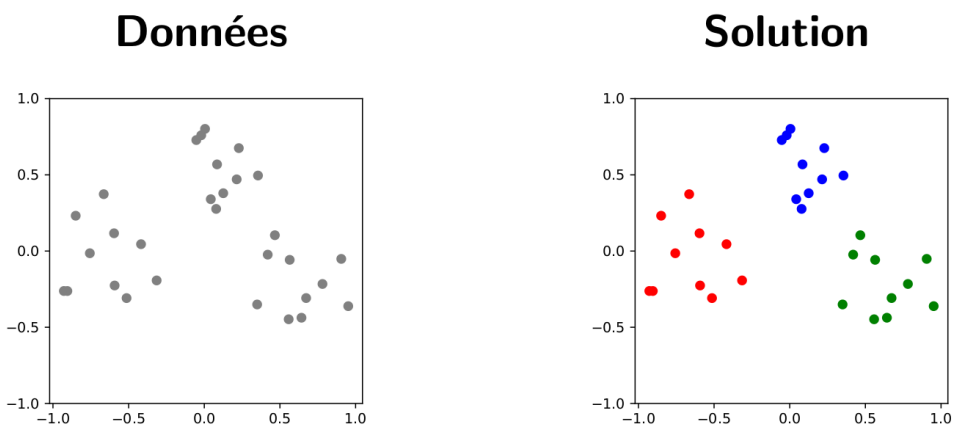
# 2 Apprentissage non-supervisé

L'apprentissage non-supervisé est plus ambitieux que l'apprentissage supervisé, mais il est aussi plus proche de notre propre modèle d'apprentissage, basé sur l'observation. Il consiste à faire en sorte qu'à partir d'un ensemble de données la machine soit capable de créer ses propres catégories, et si possible que ces catégories soient pertinentes pour nous !

Par exemple, dans l'exemple précédent des chiffres manuscrits, les données étaient déjà étiquetées. Comment faire si on ne veut pas le faire nous-même ? Comment faire en sorte que "la machine" classe elle-même les images en 10 catégories ?

## 2.1 Présentation de l'exemple d'étude

Commençons par un exemple visuellement plus simple, en considérant à nouveau des points répartis dans un plan, d'abscisse et ordonnée comprises entre -1 et 1 :

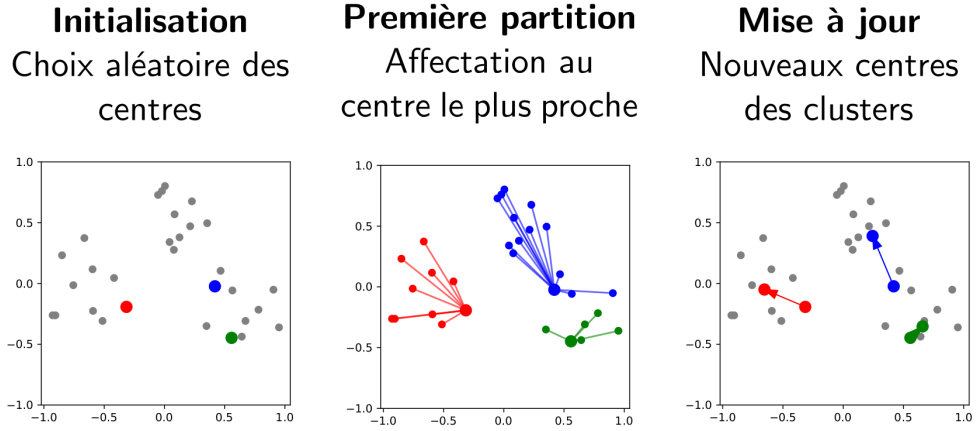



Les points ne sont pas répartis ici au hasard, on distingue clairement, grâce à notre “intelligence”, 3 groupes de points, comme montré sur la deuxième image. Le but de cette partie est de présenter un algorithme pouvant *partitionner* les points en 3 groupes.

### 2.2 Algorithme des k-moyennes

Voici le fonctionnement de cet algorithme, avec l'exemple précédent, pour lequel $ k =3$ ($k$ est le nombre de groupes) :

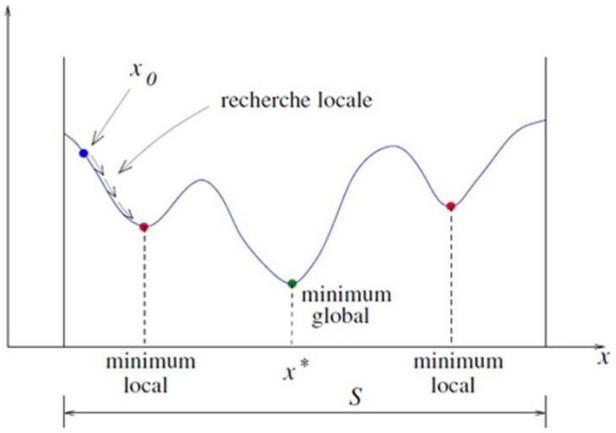
- la première étape, l'**initialisation**, consiste à placer aléatoirement 3 centres, un par groupe;
- la deuxième étape, la **partition** affecte chaque point au groupe du centre le plus proche;
- enfin la troisième étape, la **mise à jour**, met à jour la position des centres des groupes.




Il s'agit ici d'une itération de l'algorithme : on reprend alors les étapes de partition puis de mise à jour de façon répétée. On espère alors que l'algorithme converge, c'est-à-dire que les positions des centres convergent vers des positions fixes. Les 3 groupes sont alors finalement déterminés.

**Question 13 :** Tester cet algorithme à l'adresse suivante http://mpechaud.fr/scripts/kmeans/kmeans.html, disponible dans le fichier “I8_IA.py”. Arrive-t-on toujours à la solution optimale ? L'algorithme retourne-t-il toujours les mêmes groupes ? Expliquer !

C'est encore une fois méthode heuristique, qui ne converge pas forcement vers la meilleur solution, et la solution proposée peut dépendre de la position initiale aléatoire des centres.




C'est un problème tout à fait analogue à la recherche du minimum global d'une fonction : selon le point de départ de la recherche, on peut se retrouver *piégé* dans le voisinage d'un minimum local.

### 2.3 Application à la reconnaissance des chiffres manuscrits

On donne les fonctions permettant de coder l'algorithme k-moyenne.

[37]:
```
def initialiser_centres(X, n_clusters):
    '''Initialise les centres des clusters de façon aléatoire.'''
    replace = n_clusters < len(X)
    index = np.random.choice(len(X), size=n_clusters, replace=replace)
    centres = X[index]
    return centres
```

[38]:
```
def trouver_clusters(X, centres):
    '''Trouve les clusters à partir des centres (en fonction des distances).'''
    distances = []
    for x in centres:
        distances.append(np.linalg.norm(X - x, axis=1))
    distances = np.array(distances)
    y = np.argmin(distances, axis=0)
    return y
```

[39]:
```
def trouver_centres(X, y):
    '''Trouve les centres à partir des clusters (indiqués par les labels).'''
    centres = []
    for label in np.unique(y):
        centre = np.mean(X[y==label], axis=0)
        centres.append(centre)
    return np.array(centres)
```

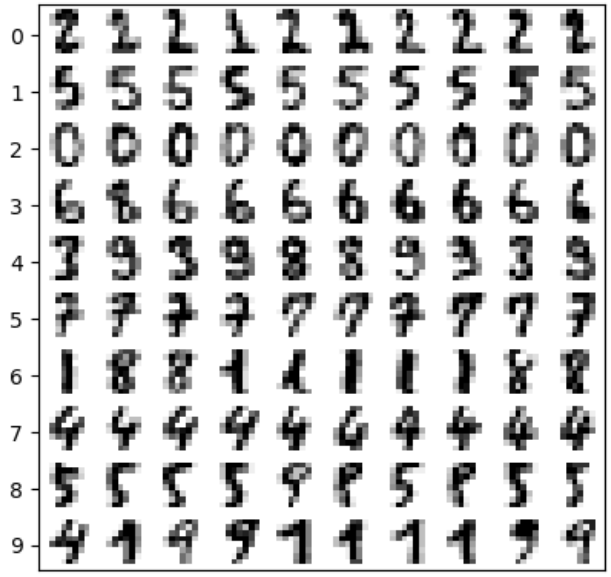
[40]:
```
def partitionner(X, n_clusters):
    '''Partitionne les données par l'algorithme des k-moyennes.'''
    centres = initialiser_centres(X, n_clusters)
    y = trouver_clusters(X, centres)
    y_prev = None
    while not np.all(y == y_prev):
        y_prev = y.copy()
        centres = trouver_centres(X, y)
        y = trouver_clusters(X, centres)
    return y
```

**Question 14 :** Expliquer le fonctionnement de la fonction `partitionner`. Exécuter le code : l'affichage montre 10 échantillons du "partionnement" pour les 10 groupes. Interpréter et commenter. Existe-t-il un avantage à choisir un nombre de groupe différent de celui attendu ?

Faisons agir la fonction `partitionner`, et observons le résultat :

[48]:
```
n_cluster = 10
y_pred = partitionner(X, n_cluster)
```

[49]:
```
index = obtenir_echantillons(X, y_pred)
montrer_chiffres(X[index], y_pred[index])
```




*Remarque : Le point déterminant de cet algorithme est le choix des positions initiales des centres. Un "bon choix" permet une convergence vers le minimum global en peu d'étapes. On peut grandement améliorer le choix de cette situation initiale en supervisant "un peu" : on peut étiqueter "à la main" quelques données, pour connaître a minima le nombre de groupes et donner des centres de groupes cohérents, puis laisser l'algorithme étiqueter les autres données.*

Pour aller plus loin sur la reconnaissance de caractères : http://mpechaud.fr/scripts/kmeans/kmeansdigits.html

## 2.4 Autre application : compression d'images

Voir http://mpechaud.fr/scripts/kmeans/kimage.html