Université de Montpellier **HMM308**
2019/2020 **Alexandre Gramfort, Joseph Salmon, Claire Vernade**

Correction TP n° 1 : k-plus proches voisins

Les fichiers `tp_knn_source.py` et `tp_knn_script.py` sont disponibles sur le site pédagogique du cours. Ils contiennent le code et les fonctions utiles pour la partie sur les k-plus proches voisins.

- Découverte de Python -

Consulter les pages suivantes pour démarrer ou bien trouver quelques rappels de Python :

⋆⋆⋆ https://github.com/agramfort/liesse_telecom_paristech_python/blob/master/1-Intro-Python.ipynb

⋆⋆⋆ https://github.com/agramfort/liesse_telecom_paristech_python/blob/master/2-Numpy.ipynb

⋆⋆⋆ https://github.com/agramfort/liesse_telecom_paristech_python/blob/master/3-Scipy.ipynb

⋆⋆⋆ http://scikit-learn.org/stable/index.html

⋆⋆ http://www.loria.fr/~rougier/teaching/matplotlib/matplotlib.html

⋆⋆ http://jrjohansson.github.io/

- Rappels de classification -

Définitions et notations

On rappelle ici le cadre de la classification supervisée, et l'on présente les notations que l'on utilisera dans la suite.

— $\mathcal{Y}$ est l'ensemble des étiquettes des données (*labels* en anglais). Ici on raisonne avec un nombre L quelconque de classes, et l'on choisit $\mathcal{Y} = \{1, \dots, L\}$ pour représenter les L étiquettes possibles (le cas de la classification binaire est le cas où $L = 2$).

— $\mathbf{x} = (x_1, \dots, x_p)^\top \in \mathcal{X} \subset \mathbb{R}^p$ est une observation, un exemple, un point (ou un *sample* en anglais). La $j$ème coordonnée de $\mathbf{x}$ est la valeur prise par la $j$ème variable (*feature* en anglais).

— $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \dots n\}$ est l'ensemble d'apprentissage contenant les n exemples et leurs étiquettes.

— Il existe un modèle probabiliste qui gouverne la génération de nos observations selon des variables aléatoires X et Y : $\forall i \in \{1, \dots, n\}, (\mathbf{x}_i, y_i) \overset{i.i.d}{\sim} (X, Y)$.

— On cherche à construire à partir de l'ensemble d'apprentissage $\mathcal{D}_n$ une fonction appelée classifieur, $\hat{f} : \mathcal{X} \mapsto \mathcal{Y}$ qui à un nouveau point $\mathbf{x}_{\text{new}}$ associe une étiquette $\hat{f}(\mathbf{x}_{\text{new}})$.

Génération artificielle de données

On considère dans cette partie que les observations sont décrites en deux dimensions (afin de pouvoir les visualiser facilement) à savoir $p = 2$ dans le formalisme ci-dessus.

1) **Étudiez les fonctions `rand_tri_gauss`, `rand_clown` et `rand_checkers`. Que renvoient ces fonctions ? À quoi correspond la dernière colonne ?**

   La fonction `rand_tri_gauss` permet de générer un mélange de trois gaussiennes en présisant le nombre de points, la moyenne et la variance pour chacune d'entre elles. La dimension des données générées est indiquée par celle des paramètres d'entrée (taille de μ et Σ). La fonction `rand_clown`

permet de générer un mélange de deux distributions : la première forme une sorte de parabole $y = x^2$ (le sourire du clown) et la deuxième est une gaussienne de moyenne nulle et de variance `sigma2` (le visage du clown). La fonction `rand_checkers` génère un damier aléatoire : à partir d'une

distribution uniforme, on génère un damier dans la partie surpérieure droite du plan. La structure en damier n'est pas flagrante lorsque le nombre de points générés est petit mais elle devient évidente pour 1000 points.

2) **Utilisez la fonction `plot_2d` afin d'afficher quelques jeux de données, et définir des bons paramètres pour les 4 jeux de données.**

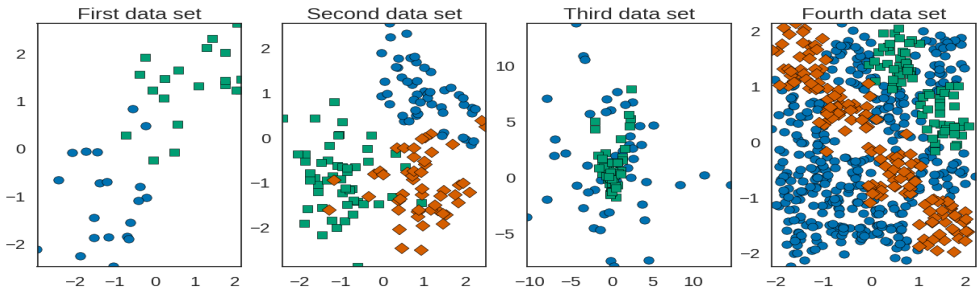


FIGURE 1 – Les quatre jeux de données, visualisés grâce à la fonction `plot2D`

- LA MÉTHODE DES k-PLUS PROCHES VOISINS -

Approche intuitive

L'algorithme des k-plus proches voisins (k-nn : pour *k-nearest neighbors* en anglais) est un algorithme intuitif, aisément paramétrable pour traiter un problème de classification avec un nombre quelconque d'étiquettes.

Le principe de l'algorithme est particulièrement simple : pour chaque nouveau point $\mathbf{x}$ on commence par déterminer l'ensemble de ses k-plus proches voisins parmi les points d'apprentissage que l'on note $V_k(\mathbf{x})$ (bien sûr on doit choisir $1 \leq k \leq n$ pour que cela ait un sens). La classe que l'on affecte au nouveau point $\mathbf{x}$ est alors la classe majoritaire dans l'ensemble $V_k(\mathbf{x})$. Une illustration de la méthode est donnée en Figure 2 pour le cas de trois classes.

1) **Proposez une version adaptée de cette méthode pour la régression, *i.e.*, quand les observations sont à valeurs réelles : $\mathcal{Y} = \mathbb{R}$.**

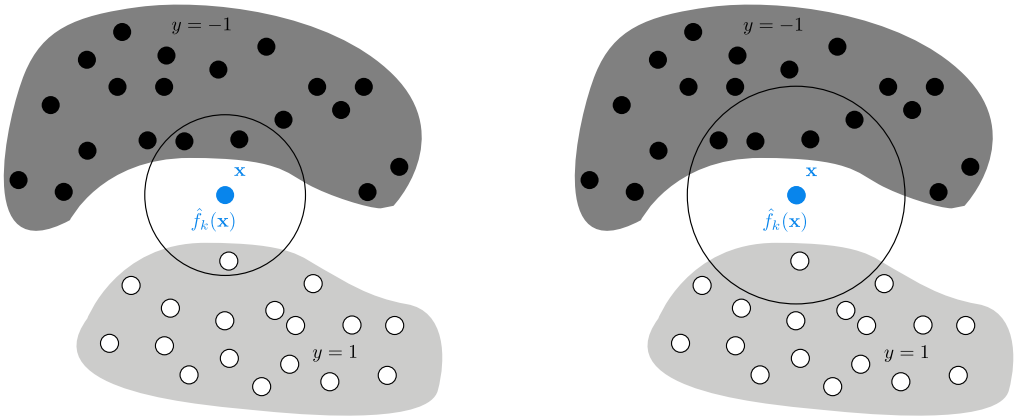
Pour la régression, on peut par exemple affecter à x la valeur moyenne des k plus proches voisins.

Approche formelle

Pour définir précisément la méthode, il faut commencer par choisir une distance $d : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$. Pour un nouveau point $\mathbf{x}$, on définit alors l'ensemble de ses k-plus proches voisins $V_k(\mathbf{x})$ au sens de de cette distance. On peut procéder de la manière suivante : pour chaque $\mathbf{x} \in \mathbb{R}^d$ et pour chaque $i = 1, \ldots, n$, on note $d_i(\mathbf{x})$ la distance entre $\mathbf{x}$ et $\mathbf{x}_i$: $d_i(\mathbf{x}) = d(\mathbf{x}_i, \mathbf{x})$. On définit la première statistique de rang $r_1(\mathbf{x})$ comme l'indice du plus proche voisin de $\mathbf{x}$ parmi $\mathbf{x}_1, \ldots, \mathbf{x}_n$, c'est-à-dire

$$r_1(\mathbf{x}) = i^* \qquad \text{si et seulement si} \qquad d_{i^*}(\mathbf{x}) = \min_{1 \leq i \leq n} d_i(\mathbf{x}).$$

Remarque 1. *S'il y a plusieurs candidats pour la minimisation ci-dessus, on ordonne les ex-æquos de manière arbitraire (généralement aléatoirement).*



(a) $k = 5$ (b) $k = 11$

FIGURE 2 – Exemple de fonctionnement de la méthode des k-plus proches voisins pour des valeurs du paramètres $k = 5$ et $k = 11$. On considère trois classes, $L = 3$, représentées respectivement en noir ($y = 1$), en gris ($y = 2$) et en blanc ($y = 3$).

Par récurrence on peut ainsi définir le rang $r_k(\mathbf{x})$ pour tout entier $1 \leq k \leq n$:

$$r_k(\mathbf{x}) = i^* \quad \text{si et seulement si} \quad d_{i^*}(\mathbf{x}) = \min_{\substack{1 \leq i \leq n \\ i \neq r_1, \ldots, r_{k-1}}} d_i(\mathbf{x}). \tag{1}$$

L'ensemble de k-plus proches voisins de $\mathbf{x}$ s'exprime alors par $V_k(\mathbf{x}) = \{\mathbf{x}_{r_1}, \ldots, \mathbf{x}_{r_k}\}$. Pour finir, la décision pour classifier le point $\mathbf{x}$ se fait par vote majoritaire, en résolvant le problème suivant :

$$\hat{f}_k(\mathbf{x}) \in \arg\max_{y \in \mathcal{Y}} \left(\sum_{j=1}^{k} \mathbb{1}_{\{Y_{r_j} = y\}} \right). \tag{2}$$

Le module `sklearn.neighbors` de scikit-learn, (*cf.* http://scikit-learn.org/stable/modules/neighbors.html) implémente les méthodes de classification et régression à base de k-plus proches voisins.

2) Complétez la classe `KNNClassifier`. Vérifier la validité des résultats en les comparant à ceux de la classe `KNeighborsClassifier` de scikit-learn. Vous proposerez votre propre méthode de comparaison.

Un classe en Python (comme en Java) est une 'grosse' fonction qui a ses propres attributs (paramètres donnés en entrée) et ses propres méthodes. Lorsque l'on appelle la fonction `KNNClassifier` grace à la ligne

```
my_classifier = KNNClassifier(n_neighbors=5)
```

, on crée en fait un *objet* 'classifieur k-plus-proches-voisins pour $k = 5$'. Cet objet pourra faire appel à sa fonction `fit` pour apprendre sur la base d'entraînement qu'on lui donnera en appelant

```
my_classifier.fit(X_train,Y_train)
```

et pourra prédire un label pour un (ou plusieurs) points(s) `X_test` par la méthode des 5- plus proches voisins avec la ligne

```
my_classifier.predict(X_test).
```

Nous proposons une solution d'implémentation pour la classe `KNNClassifier` mais ça n'est pas la seule manière possible de le faire [1]

1. N'hésitez pas à googler en anglais ce que vous voulez faire avec Python, toute la communauté de développeurs du machine learning s'est posée ces questions avant vous !

```
class KNNClassifier(BaseEstimator, ClassifierMixin):
    """ Home made KNN Classifier class"""
    def __init__(self, n_neighbors=1):  // cette fonction initialise simplement
                                            la classe avec son parametre n_neigbors
        self.n_neighbors = n_neighbors

    def fit(self, X, y): // la fonction fit (fournie) instancie
                        la base de train pour ce classifieur
        self.X_ = X
        self.y_ = y
        return self

    def predict(self, X):
        n_samples, n_features = X.shape
        # TODO : Compute all pairwise distances between X and self.X_
        // astuce : google this 'Python Compute pairwise distances vector matrix'

        dist = metrics.pairwise.pairwise_distances(
            X, Y=self.X_, metric='euclidean', n_jobs=1)

        # Get indices to sort them // trier les indices en fonction de la proximite a X
        idx_sort = np.argsort(dist, axis=1)

        # Get indices of neighbors // obtenir les 'n_neighbors' les plus proches
        idx_neighbors = idx_sort[:, :self.n_neighbors]

        # Get labels of neighbors
        Y_neighbors = self.y_[idx_neighbors]

        # TODO : Find the predicted labels y for each entry in X
        # You can use the scipy.stats.mode function
        mode, _ = stats.mode(Y_neighbors, axis=1)
        y_pred = np.asarray(mode.ravel(), dtype=np.intp)
        return y_pred // attention au format des outputs de mode !
```

Pour gagner en temps de calcul, vous utiliserez à partir de maintenant l'implémentation de scikit-learn. Elle fonctionne de manière similaire à la vôtre mais elle est optimisée pour que les calculs prennent moins de temps. Utilisez-là de manière similaire à ce qui vous a été proposé dans la question 2.

3) **Utilisez cet algorithme de classification sur les trois exemples de jeux de données, avec la distance euclidienne classique** $d(\mathbf{x}, \mathbf{v}) = \|\mathbf{x} - \mathbf{v}\|_2$**.**
Pour visualiser les résultats du classifieur, utilisez la fonction `frontiere_new` fournie dans le fichier source. Il n'est pas interdit d'aller regarder à quoi ressemble cette fonction mais n'y passez pas trop de temps. Elle sert simplement à calculer le résultat de la fonction `knn.predict` sur une grille de point de l'espace afin de visualiser quelles sont les zones où votre classifieur prédirait chaque label [2].
Pour afficher les résultats `X_train,y_train`, vous pouvez écrire

```
def f(xx):
    """Classifier: needed to avoid warning due to shape issues"""
    return knn.predict(xx.reshape(1, -1))

knn.fit(X_train, y_train)
plt.figure()
plot_2d(X_train, y_train)
frontiere_new(f, X_train, y_train, w=None, step=50, alpha_choice=1)
```

2. D'où la necessité de lui passer en argument la fonction `predict` de votre classifieur que l'on réécrit dans `f` plus bas

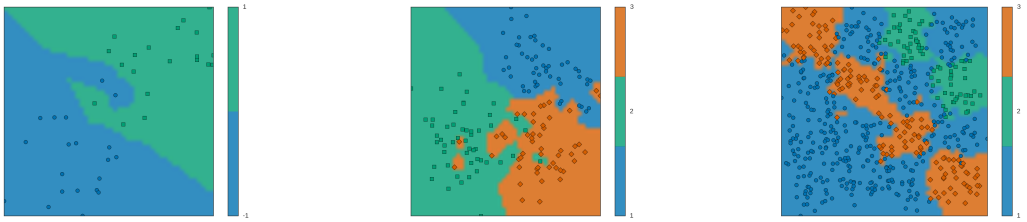


FIGURE 3 – Exemple de fonctionnement de la méthode des k-plus proches voisins pour d $k = 5$ sur trois des différents datasets.

4) **Faites varier le nombre k de voisins pris en compte. Que devient la méthode dans le cas extrême où $k = 1$? $k = n$? Afficher ces cas sur les données étudiées. Dans quels cas la frontière est-elle complexe ? simple ?**
La frontière est très complexe lorsque $k = 1$ car le classifieur est très sensible au bruit présent dans les données, on dit qu'il 'surapprend'. Au contraire lorsque $k = n$, un seul label (le plus courant dans la base d'apprentissage) est prédit pour chaque point de l'espace donc la frontière n'existe plus. Il y a un juste milieu à trouver pour le paramètre k. Les résultats pour les données `rand_tri_gauss` sont présentés Figure 4.

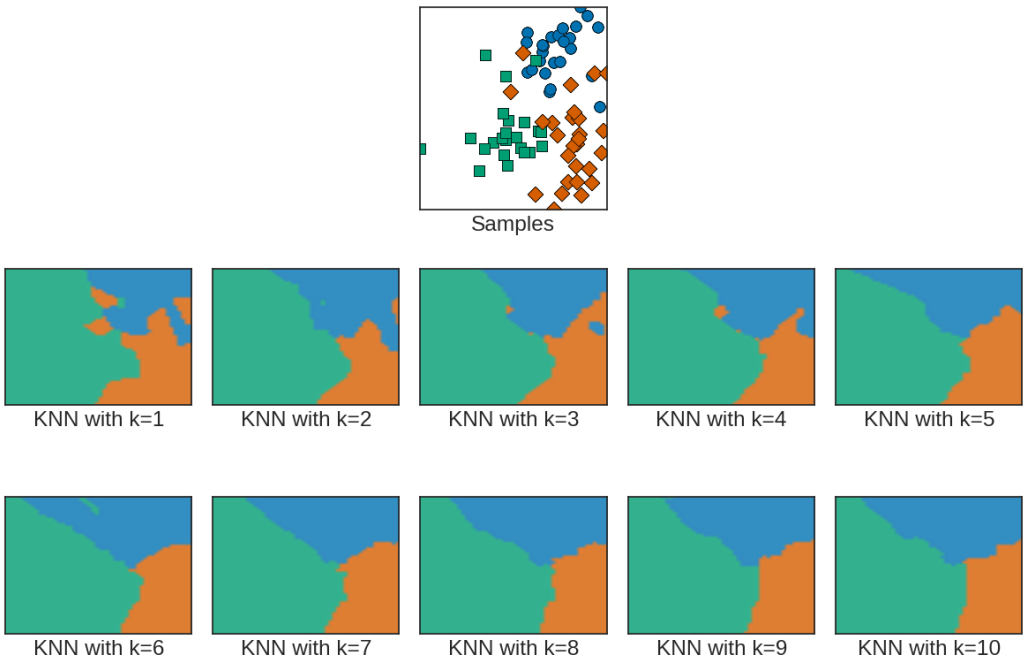


FIGURE 4 – Résultats pour k entre 1 et 10

5) **Une variante possible très utilisée consiste à pondérer les poids du $j$ème voisin selon $e^{-d_j^2/h}$ (h contrôlant le niveau de pondération). Cela revient à remplacer l'Équation (2) par :**

$$\hat{f}_k(\mathbf{x}) \in \arg\max_{y \in \mathcal{Y}} \left(\sum_{j=1}^{k} \exp(-d_j^2/h) \mathbb{1}_{\{Y_{r_j} = y\}} \right). \tag{3}$$

Implémentez cette variante dans votre classe `KNNClassifier` et dans scikit-learn en passant le paramètre `weights` au constructeur de `KNeighborsClassifier`. On pourra s'inspirer de `_weight_func` de la partie test de scikit-learn : https://github.com/scikit-learn/scikit-

learn/blob/master/sklearn/neighbors/tests/test_neighbors.py Testez l'impact du choix de h sur les frontières de classification.

```
def weights(dist):
    """Returns an array of weights, exponentially decreasing in the square
    of the distance.

    Parameters
    ----------
    dist : a one-dimensional array of distances.

    Returns
    -------
    weight : array of the same size as dist
    """
    # TODO : just write the following line :
    return np.exp(-dist ** 2 / 0.1)


n_neighbors = 5
wknn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights)
wknn.fit(X_train, Y_train)
plt.figure(4)
plot_2d(X_train, Y_train)
```

6) **Quel est le taux d'erreur sur vos données d'apprentissage (*i.e.*, la proportion d'erreur faite par le classifieur) lorsque $k = 1$? et sur des données de test ?**
On affiche l'erreur d'apprentissage grâce à la fonction `knn.score` de scikit-learn :

```
print(knn.score(X_test, Y_test))
```

Un nombre en 0 et 1 s'affiche, il correspond plutôt au taux de succès du classifieur. Pour avoir l'erreur, on calcule simplement 1−`knn.score(X_test,y_test)`. Pour les données d'entraînement, le taux de succès vaut 1 lorsque $k = 1$ puisque le plus proche voisin d'un point de la base d'entraînement est lui-même et le classifieur ne commettra jamais d'erreur. Cette dernière remarque n'est pas valable lorsque $k > 1$: dans ce cas, si un point est isolé dans une zone présentant des labels différents, il est possible que le classifieur 'se trompe' sur ce point.

7) **Tracez les différentes courbes d'erreurs en fonction du paramètre k sur l'un des jeux de données, pour des nombres d'échantillons n variant de 100, 500 à 1000. Quelle est la meilleure valeur de k ? Est-ce toujours la même pour les différents datasets ? Attention à bien évaluer l'erreur sur des données de test. Vous pourrez utiliser la classe fournie `ErrorCurve`.**

```
sigma = 0.1
plt.figure(5)
range_n_samples = [100, 500, 1000]
niter = len(range_n_samples)
for n in range(niter):
    n1 = n2 = range_n_samples[n]
    X_train, Y_train = rand_checkers(n1, n2, sigma)
    X_test, Y_test = rand_checkers(n1, n2, sigma)
    error_curve.fit_curve(X_train, Y_train, X_test, Y_test)
    error_curve.plot(color=collist[n % len(collist)], maketitle=False)

plt.legend(["training size : %d" % n for n in range_n_samples],
           loc='upper left')
```

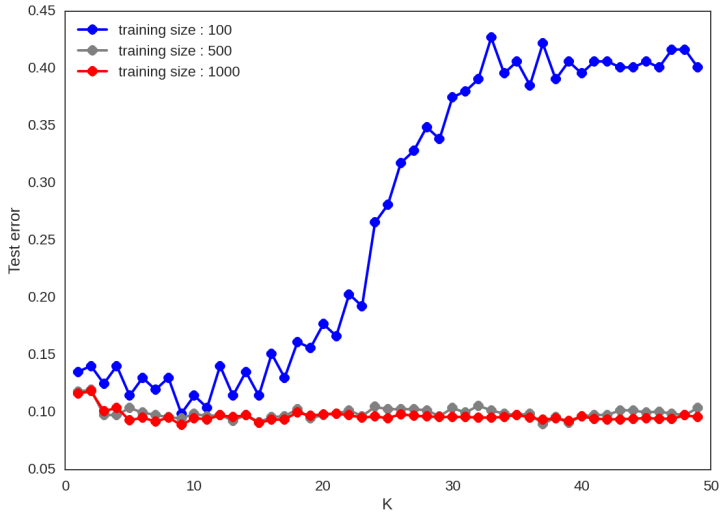


FIGURE 5 – Courbes d'erreur pour différents tailles de datasets et pour des valeurs de k variant entre

8) **A votre avis, quels sont les avantages et les inconvénients de la méthode des plus proches voisins : temps de calcul ? passage à l'échelle ? interprétabilité ?** Il s'agit d'une question ouverte. Il est certain que la méthode des k-plus-proches voisins est coûteuse en temps de calcul puisqu'il faut calculer des distances pour tous les points de la base d'entraînement pour chaque label à prédire. En outre, en grande dimension, les distances entre les points ont tendance à s'aggrandir et la notion de voisin n'est plus très raisonnable. On parle de la 'malédiction de la dimensionalité'. Cependant, cette méthode a l'avantage d'être très intuitive et très facile à comprendre pour commencer avec l'idée de classification supervisée. Elle est donc très aisément interprétable.

9) **Appliquez la méthode aux données issues de la base** ZIPCODE **avec différents choix de $k \geq 1$. On pourra se référer à** `http://scikit-learn.org/stable/_downloads/plot_digits_classification.py` **pour le chargement et la manipulation de la base de données. Pour de plus amples informations sur la nature de la classe** `'Bunch'` **(une sous-classe de dictionnaire, on se reportera à la documentation sur la classe** `'dict'` **:** `http://docs.python.org/2/library/stdtypes.html#mapping-types-dict`.

On charge la base de donnée avec la ligne suivante

```
digits = datasets.load_digits()
```

Ensuite, on peut éventuellement observer un échantillon de chacune des 10 catégories pour se rendre compte de ce à quoi ressemble la base. En gros, ce sont les dix chiffres dans une image très pixellisée.

Afin de procéder à la classification, on divise notre base d'entraînement en deux : un train et un test

```
X_train = digits.data[:n_samples // 2]
Y_train = digits.target[:n_samples // 2]
X_test = digits.data[n_samples // 2:]
Y_test = digits.target[n_samples // 2:]
```

Et on applique simplement la fonction de scikit learn comme précédemment :

```
knn = neighbors.KNeighborsClassifier(n_neighbors=30)
knn.fit(X_train, Y_train)

score = knn.score(X_test, Y_test)
Y_pred = knn.predict(X_test)
print(score)
```

10) **Estimez la matrice de confusion $(\mathbb{P}\{Y = i, C_k(X) = j\})_{i,\,j}$ associée au classifieur C_k ainsi obtenu. Pour la manipulation de telles matrices avec scikit-learn, on pourra consulter** http://scikit-learn.org/stable/auto_examples/plot_confusion_matrix.html. On calcule la matrice de confusion et sa version normalisée comme suit

```
CM = metrics.confusion_matrix(Y_test, Y_pred)
print(CM)

CM_norm = 1.0 * CM / CM.sum(axis=1)[:, np.newaxis]
print(CM_norm)
plt.matshow(CM)
```

Et on obtient une matrice qui indique pour chaque catégorie la proportion de prédiction de notre classifieur. Par exemple sur la ligne 0, on va avoir pour chaque colonne $j \in \{0, \ldots, 9\}$ le nombre de fois où le classieur a prédit la classe j alors que la donnée représentait un 0. Cette matrice est réprésentée Figure 6.

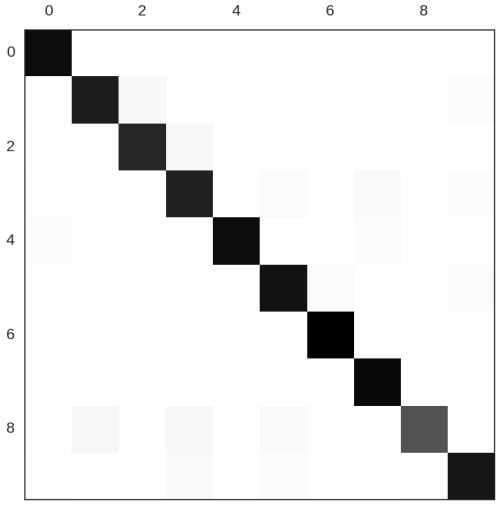


FIGURE 6 – Matrice de confusion pour la base 'digits'

11) **Proposez une méthode pour choisir k et mettez-la en œuvre. Vous pourrez utiliser la classe fournie `LOOCurve`.**

On propose de procéder par cross-validation. On divise notre dataset d'entraînement en plusieurs segments et on va entraîner chaque valeur de k sur toute la base sauf un segment et tester le classifieur obtenu sur le segment restant. Dans le cas particulier où l'on entrîane le classifieur sur toute la base sauf 1 élément (segment de taille 1), on appelle cette méthode 'leave-one-out' [3] On répète cette procédure jusqu'à trouver le k minimisant l'erreur d'apprentissage. Ceci est implément dans votre fichier source (comme indiqué dans l'énoncé), vous n'avez pas à le coder 'à la main'! Utilisez plutôt ceci :

```
loo_curve = LOOCurve(k_range=list(range(1, 50, 5)) + list(range(100, 300, 100)))
loo_curve.fit_curve(X=digits.data, y=digits.target)
```

3. D'où le nom `LOOCurve`.

- Pour aller plus loin -

Des détails généraux sur la méthode des k-plus proches voisins se trouvent dans [HTF09, Chapitre 13]. Pour améliorer la compréhension théorique de la méthode on peut se reporter au livre [DGL96, Chapitre 11] et les limites de la méthode quand $k = 1$ http://certis.enpc.fr/%7Edalalyan/Download/DM1.pdf. Enfin pour les considérations algorithmiques on pourra commencer par lire http://scikit-learn.org/stable/modules/neighbors.html#brute-force et les paragraphes suivants.

Références

[DGL96] L. Devroye, L. Györfi, and G. Lugosi. *A probabilistic theory of pattern recognition*, volume 31 of *Applications of Mathematics (New York)*. Springer-Verlag, New York, 1996. 9

[HTF09] T. J. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, New York, second edition, 2009. 9