Clustering

- Introduction -

Le but de cette séance est la mise en oeuvre d'algorithmes de *clustering*. Le *clustering* est un problème non-supervisé. Étant donné un ensemble d'apprentissage $\{X_1, \dots, X_n\}$ le but est de regrouper les points par groupes, ou *clusters*. Les points d'un même groupe sont dits avoir le même *label*. Notez qu'à la différence d'un problème supervisé il n'y a pas de $\{Y_1, \dots, Y_n\}$. Dans le cadre de ce TP on s'intéressera aux algorithmes où le nombre de *clusters* (noté K dans la suite) est fixé au préalable par l'utilisateur.

- Mise en oeuvre -

Les algorithmes de *clustering* de `scikit-learn` sont disponibles dans le module *sklearn.cluster*. Les modèles de mélanges Gaussiens (*Gaussian Mixture Models* ou GMM) peuvent aussi être vus comme des estimateurs de densité et sont disponibles dans le module `sklearn.mixture`.

K-Means

L'algorithme de K-Means partitionne les points en K groupes disjoints $\{\mathcal{C}_1, \dots, \mathcal{C}_K\}$ en minimisant la variance intra-classe. Le critère minimisé est appelé *inertie* :

$$\sum_{k \in \{1,\dots,K\}} \sum_{i \in \mathcal{C}_k} \|x_i - \mu_k\|_2^2$$

où les μ_k sont les centroïdes des classes (et $|\mathcal{C}_k|$ le cardinal de chaque classe) :

$$\mu_k = \frac{1}{|\mathcal{C}_k|} \sum_{i \in \mathcal{C}_k} x_i, \quad \forall k \in \{1, \dots, K\} \ ,$$

et où $i \in \mathcal{C}_{k_o}$ si :

$$k_o = \operatorname*{arg\,min}_{k \in \{1,\dots,K\}} \|x_i - \mu_k\|_2 = \operatorname*{arg\,min}_{k \in \{1,\dots,K\}} \|x_i - \mu_k\|_2^2 \quad .$$

L'apprentissage se fait en alternant deux étapes : une étape d'assignement où, sachant les $(\mu_k)_{k=1,\dots,K}$, on calcule les labels de chaque point puis une étape de mise à jour des centroïdes sachant les labels. On arrête l'algorithme quand l'inertie ne décroît plus beaucoup (il est à noter que sans critère d'arrêt l'algorithme K-Means termine de toute manière en un nombre fini d'étape *cf.* [BMDG05]).

L'inertie est un critère non-convexe, la solution trouvée dépend de l'initialisation. En conséquence, l'algorithme est souvent lancé plusieurs fois avec des initialisations différentes pour ne garder que la solution avec l'inertie la plus faible.

Pour plus d'information sur l'algorithme K-means et ses généralisations on pourra consulter [BMDG05] (disponible : http://machinelearning.wustl.edu/mlpapers/paper_files/BanerjeeMDG05.pdf).

1. En vous basant sur le squelette de code de la page suivante (extrait de `TP_clustering_kmeans.py`), implémentez l'algorithme K-Means. Seule la fonction *compute_ inertia_ centroids* est à compléter.
2. L'inertie décroît-elle bien au cours des itérations ?
3. En faisant varier l'initialisation du générateur de nombres aléatoires observez que la solution trouvée n'est pas toujours la même.
4. Comparez le résultat avec celui fournit pas l'implémentation de `scikit-learn`.

```
from sklearn import cluster
kmeans = cluster.KMeans(n_clusters=3, n_init=10)
kmeans.fit(X)
labels = kmeans.labels_
```

```python
# type here missing imports
# import numpy as np

def generate_data(n, centroids, sigma=1., random_state=42):
    """Generate sample data

    Parameters
    ----------
    n : int
        The number of samples
    centroids : array, shape=(k, p)
        The centroids
    sigma : float
        The standard deviation in each class

    Returns
    -------
    X : array, shape=(n, p)
        The samples
    y : array, shape=(n,)
        The labels
    """
    rng = np.random.RandomState(random_state)
    k, p = centroids.shape
    X = np.empty((n, p))
    y = np.empty(n)
    for i in range(k):
        X[i::k] = centroids[i] + sigma * rng.randn(len(X[i::k]), p)
        y[i::k] = i
    order = rng.permutation(n)
    X = X[order]
    y = y[order]
    return X, y


def compute_labels(X, centroids):
    """Compute labels

    Parameters
    ----------
    X : array, shape=(n, p)
        The samples

    Returns
    -------
    labels : array, shape=(n,)
        The labels of each sample
    """
    dist = spatial.distance.cdist(X, centroids, metric='euclidean')
    return dist.argmin(axis=1)


def compute_inertia_centroids(X, labels):
    """Compute inertia and centroids

    Parameters
```

```
    ----------
    X : array, shape=(n, p)
        The samples
    labels : array, shape=(n,)
        The labels of each sample

    Returns
    -------
    inertia: float
        The inertia
    centroids: array, shape=(k, p)
        The estimated centroids
    """
    # insert code here
    return inertia, centroids


def kmeans(X, n_clusters, n_iter=100, tol=1e-7, random_state=42):
    """K-Means : Estimate position of centroids and labels

    Parameters
    ----------
    X : array, shape=(n, p)
        The samples
    n_clusters : int
        The desired number of clusters
    tol : float
        The tolerance to check convergence

    Returns
    -------
    centroids: array, shape=(k, p)
        The estimated centroids
    labels: array, shape=(n,)
        The estimated labels
    """
    # initialize centroids with random samples
    rng = np.random.RandomState(random_state)
    centroids = X[rng.permutation(len(X))[:n_clusters]]
    labels = compute_labels(X, centroids)
    old_inertia = np.inf
    for k in xrange(n_iter):
        inertia, centroids = compute_inertia_centroids(X, labels)
        if abs(inertia - old_inertia) < tol:
            print 'Converged !'
            break
        old_inertia = inertia
        labels = compute_labels(X, centroids)
        print inertia
    else:
        print 'Dit not converge...'
    # insert code here
    return centroids, labels


if __name__ == '__main__':
    n = 1000
```

```
    centroids = np.array([[0., 0.], [2., 2.], [0., 3.]])
    X, y = generate_data(n, centroids)

    centroids, labels = kmeans(X, n_clusters=3, random_state=42)
    # plotting code
    import matplotlib.pyplot as plt
    plt.close('all')
    plt.figure()
    plt.plot(X[y == 0, 0], X[y == 0, 1], 'or')
    plt.plot(X[y == 1, 0], X[y == 1, 1], 'ob')
    plt.plot(X[y == 2, 0], X[y == 2, 1], 'og')
    plt.title('Ground truth')

    plt.figure()
    plt.plot(X[labels == 0, 0], X[labels == 0, 1], 'or')
    plt.plot(X[labels == 1, 0], X[labels == 1, 1], 'ob')
    plt.plot(X[labels == 2, 0], X[labels == 2, 1], 'og')
    plt.title('Estimated')
    plt.show()
```

Gaussian Mixture Models - GMM

Un des défauts de l'algorithme K-Means est qu'il produit des *clusters* de même taille en faisant l'hypothèse implicite que chaque *cluster* a la même variance. Les modèles de mélanges Gaussiens (GMM) permettent de relaxer cette hypothèse. En effet, on peut voire l'algorithme K-Means comme une version similaire de l'algorithme EM appliquée à un mélange de gaussiennes, dont les matrices de variance covariance sont identiques.

Pour la version générale, chaque *cluster* est modélisé comme une distribution gaussienne dont la variance est estimée au cours des itérations. Afin de mettre en évidence la capacité des GMM à estimer la covariance des classes exécutez l'exemple suivant `TP_clustering_gmm.py` :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import mixture

n_samples = 300
np.random.seed(0)
C = np.array([[0., -0.7], [3.5, .7]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          np.random.randn(n_samples, 2) + np.array([20, 20])]

clf = mixture.GMM(n_components=2, covariance_type='full')
clf.fit(X)
labels = clf.predict(X)
x = np.linspace(-20.0, 30.0)
y = np.linspace(-20.0, 40.0)
XX, YY = np.meshgrid(x, y)
Z = np.log(-clf.eval(np.c_[XX.ravel(), YY.ravel()])[0]).reshape(XX.shape)
plt.close('all')
CS = plt.contour(XX, YY, Z)
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.plot(X[labels == 0, 0], X[labels == 0, 1], 'or')
plt.plot(X[labels == 1, 0], X[labels == 1, 1], 'ob')
plt.axis('tight')
plt.show()
```

Clustering de documents

Le but de cette dernière partie est le *clustering* de documents textes en ce basant sur la similarité entre les TF-IDF (Term Frequency–Inverse Document Frequency). L'approche est dites du type *bag-of-words*. Le *clustering* se fait avec un K-Means en ligne (ou *on-line*) qui est plus rapide que le K-Means décrit ci-dessus. Il travaille aussi avec des matrices creuses (*cf.* `scipy.sparse`) pour être plus efficace. Les données sont des messages postés sur 20 forums de discussion à thème, par exemple *alt.atheism, comp.graphics*. Le but est de dégager des groupes de messages, des *clusters*, utilisant des mots similaires.

On se basera sur le code suivant `TP_document_clustering.py` :

```
from time import time
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import MiniBatchKMeans

# Load some categories from the training set
categories = ['alt.atheism','comp.graphics']

print "Loading 20 newsgroups dataset for categories:"
print categories

dataset = fetch_20newsgroups(subset='all', categories=categories,
                             data_home='../trash',
                             shuffle=True, random_state=42)

print "%d documents" % len(dataset.data)
print "%d categories" % len(dataset.target_names)

# Print the content of a message
print dataset.data[0]
labels = dataset.target
true_k = np.unique(labels).shape[0]
print "Extracting features from the training set using a sparse vectorizer"
t0 = time()
vectorizer = TfidfVectorizer(max_df=0.5, max_features=10000,
                             stop_words='english')
X = vectorizer.fit_transform(dataset.data)
print "done in %fs" % (time() - t0)
print "n_samples: %d, n_features: %d" % X.shape


# Do the actual clustering with online K-Means
km = MiniBatchKMeans(n_clusters=true_k, init='k-means++', n_init=1,
                     init_size=1000, batch_size=500, verbose=1)

print "Clustering sparse data with %s" % km
t0 = time()
km.fit(X)
print "done in %0.3fs" % (time() - t0)

# Look at terms that are the most present in each class
feature_names = vectorizer.get_feature_names()
n_top_words = 10

for k, centroid in enumerate(km.cluster_centers_):
    print "Cluster #%d:" % k
    print " ".join([feature_names[i]
```

```
                for i in centroid.argsort()[:-n_top_words - 1:-1]])
```

5. Qu'affiche le script précédent ? Comparer les résultats obtenus en appliquant séparément l'algorithme aux catégories `'alt.atheism'` et `'comp.graphics'`.

Quantification / compression d'images

Les algorithmes de *clustering* peuvent aussi servir à la quantification vectorielle : un ensemble de valeurs sont quantifiées sur quelques valeurs dans un but par exemple de compression. C'est ce qui est utilisé pour représenter des images numérisées.

6. Appliquez l'algorithme K-Means pour réduire le nombre de niveaux de gris de l'image suivante : `Grey_square_optical_illusion.jpg` (disponible dans le fichier .zip). Pour cela, on pourra passer d'une représentation couleurs (RGB) à une représentation en niveau gris, puis à une vectorisation de l'image avec un `reshape`. À partir de combien de classes ne voit-on plus la différence avec l'image initiale ?

Références

[BMDG05] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh. Clustering with Bregman divergences. *J. Mach. Learn. Res.*, 6 :1705–1749, 2005. 1