tp5

April 21, 2016

1 Apprentissage non supervisé

Ce domaine (unsupervised learning, clustering) regroupe un ensemble varié de techniques qui visent à trouver des sous-ensembles cohérents des données. Beaucoup d'approches existent selon les représentations considérées (similarité de données, représentation vectorielle, graphes), deux formulations principales sont généralement considérées étant un ensemble de données D :

- Clustering hard : on cherche une partition de D en k parties, deux à deux disjointes, qui minimise une certaine fonction de coût.
- Clustering soft : à chaque exemple on associe une probabilité d'appartenance à chaque cluster.

2 Apprentissage de prototype : Algorithme k-means

Un type d'approche pour le clustering est géométrique : il s'agit de trouver une partition de l'espace d'entrée en considérant la densité d'exemples pour caractériser ces partitions. Un exemple de tel algorithme est k-means : cet algorithme considère pour chaque cluster C_i un prototype $\mu_i \in \mathbb{R}^d$ dans l'espace d'entrée. Chaque exemple x est affecté au cluster le plus proche au sens de la distance euclidienne à son prototype. Soit $s_C : \mathbb{R} \to \mathbb{N}$ la fonction d'affectation associé au clustering

$$C = \{C_1, C_2, \ldots, C_k\} \ : \ s_C(x) = argmin_i \|\mu_i - x\|^2$$

.

La fonction de coût sur un ensemble de données $D = \{x_1, \ldots, x_n\}$ généralement considérée dans ce cadre est la moyenne des distances intra-clusters : $\frac{1}{n}\sum_{i=1}^{n}\sum_{j|s_C(x_j)=i}\|\mu_i - x_j\|^2 = \frac{1}{n}\sum_{i=1}^{n}\|\mu_{s_C(x_i)} - x_i\|^2$.

C'est également ce qu'on appelle le coût de reconstruction : effectivement, dans le cadre de cette approche, chaque donnée d'entrée peut être "représentée" par le prototype associé : on réalise ainsi une compression de l'information (n.b. : beaucoup de liens existent entre l'apprentissage et la théorie de l'information, la compression et le traitement de signal). L'algorithme fonctionne en deux étapes, (la généralisation de cet algorithme est appelée algorithme E-M, Expectation-Maximization) :

- à partir d'un clustering C^t, les prototypes $\mu_i^t = \frac{1}{|C_i|}\sum_{x_j \in C_i^t} x_j$, barycentres des exemples affectés à ce cluster;

- à partir de ces nouveaux barycentres, calculer la nouvelle affectation de chaque exemple (le prototype le plus proche). Si un cluster se retrouve sans aucun exemple, il est ré-initialisé au hasard, par exemple en tirant un exemple de la base d'apprentissage au hasard.

Ces deux étapes sont alternées jusqu'à stabilisation. L'initialisation est aléatoire : soit les centres sont tirés au hasard, soit les affectations.

Cet algorithme peut être entre autre utilisé pour faire de la compression d'image (connu également sous le nom de quantification vectorielle). Une couleur est codée par un triplet (r, g, b) dénotant le mélange de composantes rouge, vert et bleu. En limitant le nombre de couleurs possibles dans l'image (ces couleurs forment ce qu'on appelle un dictionnaire), on réalise une grosse compression. L'objectif est de trouver quelles couleurs doivent être présentent dans notre dictionnaire afin de minimiser l'erreur entre l'image compressé et l'image original (remarque : c'est exactement l'erreur de reconstruction ci-dessus).

En considérant l'ensemble des pixels de l'image comme la base d'exemples non supervisée, le nouveau codage de chaque pixel peut être obtenu par le résultat de l'algorithme k-means sur cette base d'exemples.

Le code suivant contient un squelette de l'algorithme k-mean. La fonction **predict(data)** permet de prédire le cluster de chaque donnée d'entrée; la fonction **fit(data)** permet d'apprendre les clusters en alternant les deux étapes de l'algorithme; la fonction **transform(data)** permet de récupérer chaque donnée remplacée par son centre et la fonction **erreur(data)** de calculer l'erreur de reconstructions. Compléter le code : vous pourrez en particulier utiliser les fonctions **numpy.linalg.norm** et **numpy.random.randint**.

Les instructions qui suivent permettent de lire, afficher, modifier, sauver une image au format **png** et de la stocker dans un tableau de taille $l \times h \times c$, l la largeur de l'image, h la hauteur, et $c = 3$ généralement pour les 3 couleurs (parfois 4, la 4eme dimension étant pour la transparence; dans la suite nous ignorons cette composante).

En codant vous même l'algorithme ou en utilisant la version de sklearn, expérimentez la compression : choisir une image, construire avec k-means l'image compressée et afficher la. Etudier en fonction du nombre de clusters (couleurs) choisis comment évolue l'erreur de reconstruction.

Quel est le gain en compresssion effectué ? Sachant que souvent une image peut être découpée en région de tonalité homogène, voyez-vous une amélioration possible pour augmenter la compression tout en diminuant l'erreur de compression ?

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

class KMeans(object):
    """ parametres :
        * k : nombre de clusters
        * maxiter : nombre d'iterations
    """
    def __init__(self,k=4,maxiter=5):
        self.k, self.maxiter, self.centres = k, maxiter, None

    #   Apprentissage des clusters
    def fit(self,data):
        N = data.shape[0] # nombre d'exemples
        d = data.shape[1] # dimension
```

```
            self.centres = np.zeros((self.k,d)) #  centres de chaque cluster, u
            clust = np.random.randint(0,self.k,N) # affectation de chaque exemp
            for t in range(self.maxiter):
                print("%d/%d : %f" %(t, self.maxiter, self.error(data)))
                for i in range(self.k):
                    pass
            return clust
        # Prediction du numero de cluster
        def predict(self,data):
            pass
        # transforme chaque exemple en son prototype
        def transform(self,data):
            pass
        # Erreur de reconstruction moyenne
        def error(self,data):
            pass


    #### KMeans et image

    im=plt.imread("image.png")[:,:,:3] #on garde que les 3 premieres composante
    im_h,im_l,_=im.shape
    data=im.reshape((im_h*im_l,3)) #transformation en matrice n*3, n nombre de

    km = KMeans(k=5,maxiter=10)
    km.fit(data)
    imnew=km.transform(data).reshape((im_h,im_l,3)) #transformation inverse
    plt.imshow(imnew) #afficher l'image
```

3 Espace latent et factorisation matricielle

Le problème dit de recommendation consiste à partir d'un historique d'avis que des utilisateurs ont donné sur un ensemble d'items de prédire les avis non observés (par exemple des notes sur des films, des achats de produits, ...). Les techniques de factorisation matricelle permettent dans ce cadre (et plus généralement) de mettre en évidence un espace "latent" (non directement observé dans les données) qui explicite les corrélations entre les avis des utilisateurs et les produits (par exemple genre des films ou profil d'utilisateurs).

Nous allons utiliser dans la suite les données movielens de notes d'utilisateurs sur des films.

3.1 Stochastic gradient descent

Soit un ensemble de m utilisateurs U et un ensemble de n items I, et une matrice R de dimension $m \times n$ telle que $r_{u,i}$ représente la note que l'utilisateur u a donné à l'item i, 0 si pas de note correspondante (on suppose les notes > 0). Dans cette partie, l'hypothèse est qu'il existe un espace latent de dimension d "commun" aux utilisateurs et aux items, qui permet d'expliquer par une combinaison linéaires les goûts des utilisateurs. Dans le cadre de la recommendation de films par exemple, cet espace pourrait décrire des genres de films. A chaque utilisateur u correspond un

vecteur de pondération x de taille d qui indique les intérêts de l'utilisateur en fonction de chaque genre; à chaque film i correspond un vecteur de pondération y de taille d qui indique la corrélation du film avec chaque genre.

L'objecitf est ainsi de trouver deux matrices X et Y de tailles $m \times d$ et $d \times n$ telles que $R \approx XY$. On utilise en général les moindres carrés comme fonction de coût pour calculer l'erreur de reconstruction. Comme seule une partie des scores est observée lors de l'apprentissage, l'erreur ne doit être comptée que pour ces scores là. La fonction à optimiser est $min_{X,Y} \sum_{u,i|r_{u,i}>0}(r_{u,i} - y'_{.,i}x_{u,.})^2 + \lambda(||y_{.,i}||^2 + ||x_{u,.}||^2)$.

Une manière d'optimiser la fonction est par descente de gradient stochastique, avec les formules de mise-à-jour suivantes :

- $e_{u,i} = r_{ui} - y'_{.,i}x_{u,.}$
- $y_{.,i} = y_{.,i} + \gamma(e_{u,i}x_{u,.} - \lambda q_{.,i})$
- $x_{u,.} = x_{u,.} + \gamma(e_{u,i}y_{.,i} - \lambda x_{u,.})$

Implémenter l'algorithme. Tester le sur les données movielens et analyser les résultats.

Le code suivant permet de charger les données movielens (données à charger à partir de http://grouplens.org/datasets/movielens/1m/, le MovieLens 1M Dataset par exemple).

```
In [ ]: def read_movielens():
            users = dict()
            movies = dict()
            with open('users.dat', "r",encoding = 'iso-8859-1') as f:
                for l in f:
                    l = l.strip().split("::")
                    users[int(l[0])] = [l[1], int(l[2]), int(l[3]), l[4]]
            with open('movies.dat', "r",encoding = 'iso-8859-1') as f:
                for l in f:
                    l = l.strip().split("::")
                    movies[int(l[0])] = [l[1],l[2].split("|")]
            ratings = np.zeros((max(users)+1,max(movies)+1))
            with open('ratings.dat',"r",encoding = 'iso-8859-1') as f:
                for l in f:
                    l = l.strip().split("::")
                    ratings[int(l[0]),int(l[1])]=int(l[2])
            return ratings, users, movies
```