8 Fundamental unsupervised learning algorithms

This chapter covers

- Dirichlet process K-means
- Gaussian mixture models
- Dimensionality reduction

In previous chapters, we looked at supervised algorithms for classification and regression; in this chapter, we shift our focus to unsupervised learning algorithms. *Unsupervised learning* takes place when no training labels are available. In this case, we are interested in discovering patterns in data and learning data representations. Applications of unsupervised learning span from clustering customer segments in e-commerce to extracting features from image data. In this chapter, we'll start by looking at the Bayesian nonparametric extension of the K-means algorithm followed by the EM algorithm for Gaussian mixture models (GMMs). We will then look at two different dimensionality reduction techniques—namely, PCA and t-SNE—used to learn an image manifold. The algorithms in this chapter were selected for their mathematical depth and usefulness in real-world applications.

8.1 Dirichlet process K-means

Dirichlet process (DP) *K-means* is a Bayesian nonparametric extension of the *K*-Means algorithm. *K-means* is a clustering algorithm that can be applied, for instance, to customer segmentation, where customers are grouped by purchase history, interests, and geographical location. DP-means is similar to *K*-means except that new clusters are created whenever a data point is sufficiently far away from all existing cluster centroids; therefore, the number of clusters grows with the data. DP-means converges to a local optimum of a K-means-like objective that includes a penalty for the number of clusters.

We select the cluster *K* based on the smallest value of the following equation.

$$\arg\min_k \{||x_i - \mu_1||^2, \ldots, ||x_i - \mu_k||^2, \lambda\} \tag{8.1}$$

The resulting update is analogous to the *K*-means reassignment step, during which we reassign a point to the cluster corresponding to the closest mean or start a new cluster if the squared Euclidean distance is greater than λ. The DP means algorithm is summarized in figure 8.1.

```
Init. $K = 1, l_1 = \{x_1, \ldots, x_n\}$ and $\mu_1$ the global mean
Init. labels $z_i = 1$ for all $i = 1, \ldots, n$
Init. $\lambda = \text{kpp\_init}(X, \text{K}_{\text{init}})$
Repeat until convergence:
    for each $x_i$:
        compute $d_{ic} = \|x_i - \mu_c\|^2$ for $c = 1, \ldots, K$
        if $\min_c d_{ic} > \lambda$, set $K = K + 1$ and $\mu_k = x_i$
        otherwise, set $z_i = \arg\min_c d_{ic}$
    for each cluster $l_j = \{x_i | z_i = j\}$, compute $\mu_j = \frac{1}{|l_j|}\sum_{x \in l_j} x$
    Compute the objective: $\sum_{c=1}^{K} \sum_{x \in l_c} \|x - \mu_c\|^2 + \lambda K$
```

Figure 8.1 DP-means algorithm pseudo-code

To evaluate cluster performance, we can use the following metrics: normalized mutual information (NMI), variation of information (VI), and adjusted Rand index (ARI). The NMI is defined as follows.

$$\text{NMI}(X, Y) = \frac{I(X;Y)}{\frac{H(X)+H(Y)}{2}} \tag{8.2}$$

Here, $H(X)$ is the entropy of X and $I(X; Y)$ is the mutual information between the ground truth label assignments X (when they are available) and the computed assignments Y. For a review of information measures, see *Elements of Information Theory* by Thomas M. Cover and Joy A. Thomas (Wiley, 2006) for details.

Let $p_{XY}(I, j) = |x_i \cap y_j|/N$ be the probability that a label belongs to cluster x_i in X and y_j in Y. Define $p_X(i) = |x_i|/N$ and $p_Y(j) = |y_j|/N$ similarly. Then, we get the following equation.

$$I(X;Y) = \sum_i \sum_j p_{XY}(i,j) \log \frac{p_{XY}(i,j)}{p_X(i)p_Y(j)} \tag{8.3}$$

Thus, NMI lies between 0 and 1, with higher values indicating more similar label assignments. The variation of information (VI) is defined as follows.

$$VI(X;Y) = H(X) + H(Y) - 2I(X;Y) = H(X|Y) + H(Y|X) \tag{8.4}$$

Thus, VI decreases as the overlap between label assignments X and Y increases. The ARI computes a similarity measure between two clusters by considering all pairs of samples and counting pairs assigned in the same or different clusters in the predicted and true clusters.

$$\begin{aligned} \text{ARI} &= \frac{RI - E[RI]}{\max RI - E[RI]} \\ RI &= \frac{TP + TN}{TP + FP + FN + TN} \end{aligned} \tag{8.5}$$

Thus, ARI approaches 1 for cluster assignments that are similar to each other. Let's implement the Dirichlet process K-means algorithm from scratch!

Listing 8.1 Dirichlet process K-means

```
import numpy as np
import matplotlib.pyplot as plt

import time
from sklearn import metrics
from sklearn.datasets import load_iris

np.random.seed(42)

class dpmeans:
                                            Initializes parameters
    def __init__(self,X):                   for DP means
        self.K = 1
        self.K_init = 4
        self.d = X.shape[1]
        self.z = np.mod(np.random.permutation(X.shape[0]),self.K)+1
        self.mu = np.random.standard_normal((self.K, self.d))
        self.sigma = 1
        self.nk = np.zeros(self.K)
        self.pik = np.ones(self.K)/self.K

        self.mu = np.array([np.mean(X,0)])          Initializes mean

        self.Lambda = self.kpp_init(X,self.K_init)      Initializes lambda

        self.max_iter = 100
        self.obj = np.zeros(self.max_iter)
        self.em_time = np.zeros(self.max_iter)

    def kpp_init(self,X,k):        K++ initialization

        [n,d] = np.shape(X)
        mu = np.zeros((k,d))
        dist = np.inf*np.ones(n)

        mu[0,:] = X[int(np.random.rand()*n-1),:]
        for i in range(1,k):
            D = X-np.tile(mu[i-1,:],(n,1))
            dist = np.minimum(dist, np.sum(D*D,1))
            idx = np.where(np.random.rand() <
     np.cumsum(dist/float(sum(dist))))
            mu[i,:] = X[idx[0][0],:]
            Lambda = np.max(dist)

        print "Lambda: ", Lambda)

        return Lambda       Lambda is the max
                            distance to k++ means
    def fit(self,X):

        obj_tol = 1e-3
        max_iter = self.max_iter
        [n,d] = np.shape(X)

        obj = np.zeros(max_iter)
        em_time = np.zeros(max_iter)
        print 'running dpmean I.')

        for iter in range(max_iter):
            tic = time.time()
            dist = np.zeros((n,self.K))

            for kk in range(self.K):                     Assignment
                Xm – X - np.tile(self.mu[kk,:],(n,1))      step
                dist[:,kk] = np.sum(Xm*Xm,1)

            dmin = np.min(dist,1)
            self.z = np.argmin(dist,1)                 Updates
            idx = np.where(dmin > self.Lambda)          labels

            if (np.size(idx) > 0):
                self.K = self.K + 1
                self.z[idx[0]] = self.K-1 #cluster labels in [0,...,K-1]
                self.mu = np.vstack([self.mu,np.mean(X[idx[0],:],0)])
                Xm – X - np.tile(self.mu[self.K-1,:],(n,1))
                dist = np.hstack([dist, np.array([np.sum(Xm*Xm,1)]).T])

            self.nk = np.zeros(self.K)
            for kk in range(self.K):
                self.nk[kk] = self.z.tolist().count(kk)
                idx = np.where(self.z == kk)
                self.mu[kk,:] = np.mean(X[idx[0],:],0)
```

Updates the step

```
            self.pik = self.nk/float(np.sum(self.nk))

            for kk in range(self.K):
                idx = np.where(self.z == kk)
                obj[iter] = obj[iter] + np.sum(dist[idx[0],
                ➥ kk],0)
            obj[iter] = obj[iter] + self.Lambda * self.K
```

Computes the objective

Checks the convergence

```
            if (iter > 0 and np.abs(obj[iter]-obj[iter-1]) <
            ➥ obj_tol*obj[iter]):
                print('converged in %d iterations\n'% iter)
                break
            em_time[iter] = time.time()-tic
        #end for
        self.obj = obj
        self.em_time = em_time
        return self.z, obj, em_time
```

Computes the normalized mutual information

```
    def compute_nmi(self, z1, z2):

        n = np.size(z1)
        k1 = np.size(np.unique(z1))
        k2 = np.size(np.unique(z2))

        nk1 = np.zeros((k1,1))
        nk2 = np.zeros((k2,1))

        for kk in range(k1):
            nk1[kk] = np.sum(z1==kk)
        for kk in range(k2):
            nk2[kk] = np.sum(z2==kk)

        pk1 = nk1/float(np.sum(nk1))
        pk2 = nk2/float(np.sum(nk2))

        nk12 = np.zeros((k1,k2))
        for ii in range(k1):
            for jj in range(k2):
                nk12[ii,jj] = np.sum((z1==ii)*(z2==jj))
        pk12 = nk12/float(n)

        Hx = -np.sum(pk1 * np.log(pk1 + np.finfo(float).eps))
        Hy = -np.sum(pk2 * np.log(pk2 + np.finfo(float).eps))

        Hxy = -np.sum(pk12 * np.log(pk12 + np.finfo(float).eps))
        MI = Hx +-Hy - Hxy;
        nmi = MI/float(0.5*(Hx+Hy))

        return nmi

    def generate_plots(self,X):

        plt.close('all')
        plt.figure(0)
        for kk in range(self.K):
            #idx = np.where(self.z == kk)
            plt.scatter(X[self.z == kk,0], X[self.z == kk,1], \
                        s = 100, marker= 'o', c = np.random.rand(3,),
                        ➥ label = str(kk))
        #end for
        plt.xlabel('X1')
        plt.ylabel('X2')
        plt.legend()
        plt.ti‘le('DP-means clus’ers')
        plt.grid(True)
        plt.show()

        plt.figure(1)
        plt.plot(self.obj)
        plt.title('DP-means objective function')
        plt.xlabel('iterations')
        plt.ylalel('penalized l2 squared distance')
        plt.grid(True)
        plt.show()

if __name__ == "__main__":

    iris = load_iris()
    X = iris.data
    y = iris.target

    dp = dpmeans(X)
    labels, obj, em_time = dp.fit(X)
    dp.generate_plots(X)

    nmi = dp.compute_nmi(y,labels)
    ari = metrics.adjusted_rand_score(y,labels)

    print("NMI: %.4f" % nmi)
    print("ARI: %.4f" % ari)
```

The performance of DP-means algorithm is shown in figure 8.2. The performance of the DP-means algorithm was evaluated on the iris dataset. We can see good clustering results based on high NMI and ARI metrics. In the following section, we'll discuss a popular clustering algorithm that models each cluster as a Gaussian distribution, taking into account not just the mean but also the covariance information.

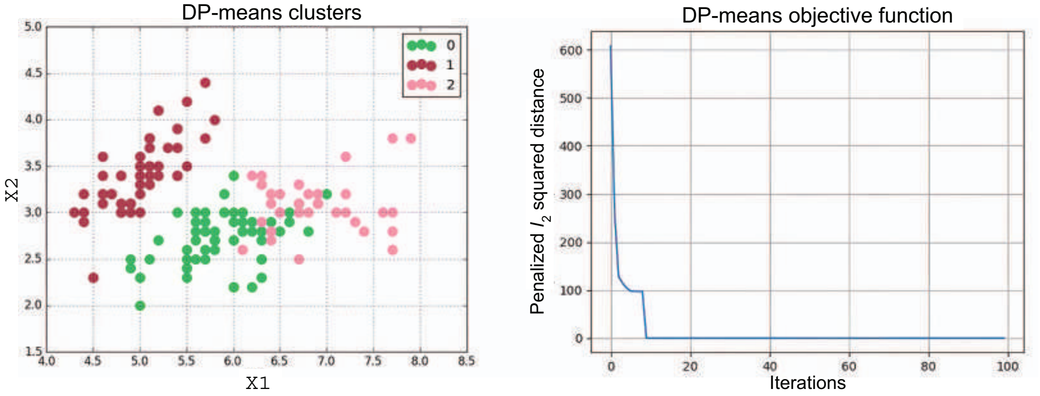

Figure 8.2 DP-means cluster centers (left) and objective (right) for the iris dataset

8.2 Gaussian mixture models

Mixture models are commonly used to model complex density distributions. For example, you may be interested in discovering patterns in census data consisting of information about the person's age, income, occupation, and other dimensions. If we plot the resulting data in high-dimensional space, we'll likely discover nonuniform density characterized by groups or clusters of data points. We can model each cluster using a base probability distribution. Mixture models consist of a convex combination of K base models. In the case of Gaussian mixture models, the base distribution is Gaussian and can be written as follows.

$$p(x_i|\theta) = \sum_{k=1}^{K} \pi_k p_k(x_i|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k) \tag{8.6}$$

Here, π_k are the mixing proportions that satisfy $0 \leq \pi_k \leq 1$ and $\sum \pi_k = 1$. In contrast to K-means that only models cluster means, GMM models the cluster covariance as well. Thus, GMMs can capture the data more accurately.

8.2.1 Expectation maximization (EM) algorithm

The *expectation maximization* (EM) *algorithm* provides a way of computing ML/MAP estimates when we have unobserved latent variables or missing data. EM exploits the fact that if the data were fully observed, then the ML/MAP estimates would be easy to compute. In particular, EM is an iterative algorithm that alternates between inferring the latent variables, given the parameters (E step) and then optimizing the parameters given filled-in data (M step).

In the EM algorithm, we define the complete data log likelihood $l_c(\theta)$, where x_i are the observed random variables and z_i are unobserved. Since we don't know z_i, we can't compute $p(x_i, z_i | \theta)$, but we can compute an expectation of $l_c(\theta)$ wrt to parameters $\theta^{(k-1)}$ from the previous iteration.

$$l_c(\theta) = \sum_{i=1}^{N} \log p(x_i, z_i|\theta) \tag{8.7}$$

The goal of the E step is to compute $Q(\theta, \theta^{(k-1)})$, on which the ML/MAP estimates depend. The goal of the M step is to recompute θ by finding the ML/MAP estimates.

$$\begin{aligned} \text{E} - \text{step} &: \quad Q\left(\theta, \theta^{(k-1)}\right) = E_{\theta^{(k-1)}}\left[l_c(\theta)|D, \theta^{(k-1)}\right] \\ \text{M} - \text{step} &: \quad \theta^{(k)} = \arg\max_{\theta} Q\left(\theta, \theta^{(k-1)}\right) + \log p(\theta) \end{aligned} \tag{8.8}$$

To derive the EM algorithm for GMM, we first need to compute the expected complete data log likelihood.

$$\begin{aligned} Q(\theta, \theta^{(k-1)}) &= E\left[\sum_i \log p(x_i, z_i|\theta)\right] \\ &= \sum_i E\left[\log\left[\prod_{k=1}^{K}(\pi_k p(x_i|\theta_k))^{1[z_i=k]}\right]\right] \\ &= \sum_i \sum_k E\left[1[z_i = k]\right] \log\left[\pi_k p(x_i|\theta_k)\right] \\ &= \sum_i \sum_k p\left(z_i = k|x_i, \theta^{(k-1)}\right) \log\left[\pi_k p(x_i|\theta_k)\right] \\ &= \sum_i \sum_k r_{ik} \log \pi_k + \sum_i \sum_k r_{ik} \log p(x_i|\theta_k) \end{aligned} \tag{8.9}$$

Here, $r_{ik} = p(z_i = k \mid x_i, \theta^{(k-1)})$ is the soft assignment of point x_i to cluster k.

We now start the E step. Given $\theta^{(k-1)}$, we want to compute the soft assignments.

$$\begin{aligned} r_{ik} &= p\left(z_i = k|x_i, \theta^{(k-1)}\right) \\ &= \frac{p\left(z_i = k, x_i|\theta^{(k-1)}\right)}{\sum_{k=1}^{K} p\left(z_i = k, x_i|\theta^{(k-1)}\right)} \\ &= \frac{p\left(x_i|z_i = k, \theta^{(k-1)}\right)\pi_k}{\sum_{k=1}^{K} p\left(x_i|z_i = k, \theta^{(k-1)}\right)\pi_k} \end{aligned} \tag{8.10}$$

In equation 8.10, $\pi_k = p(z_i = k)$ are the mixture proportions.

In the M step, we maximize Q with respect to model parameters π and θ_k. First, let's find π that maximizes the Lagrangian.

$$\begin{aligned}\frac{\partial Q}{\partial \pi_k} &= \frac{\partial}{\partial \pi_k}\left[\sum_i \sum_k r_{ik}\log \pi_k + \lambda\left(1 - \sum_k \pi_k\right)\right] \\ &= \sum_i r_{ik}\frac{1}{\pi_k} - \lambda = 0 \end{aligned} \tag{8.11}$$

Substituting th expression into the constraint, we get the following equation.

$$\sum_k \pi_k = 1 \implies \sum_k \frac{1}{\lambda}\sum_i r_{ik} = 1 \implies \lambda = \sum_i \sum_k r_{ik} = \sum_i 1 = N \tag{8.12}$$

Therefore, $\pi_k = 1/\lambda \sum r_{ik} = 1/N \sum r_{ik}$. To find the optimum parameters $\theta_k = \{\mu_k, \Sigma_k\}$, we want to optimize the terms of Q that depend on θ_k.

$$\begin{aligned} l(\mu_k, \Sigma_k) &= \sum_i r_{ik}\log p(x_i|\theta_k) \\ &\propto -\frac{1}{2}\sum_i r_{ik}\left[\log|\Sigma_k| + (x_i - \mu_k)^T \Sigma_k^{-1}(x_i - \mu_k)\right] \end{aligned} \tag{8.13}$$

To find the optimum μ_k, we differentiate this expression. First, focusing on the second term inside the sum, we can make a substitution $y_i = x_i - \mu_k$.

$$\begin{aligned} \frac{\partial}{\partial \mu_k}(x_i - \mu_k)^T \Sigma_k^{-1}(x_i - \mu_k) &= \frac{\partial}{\partial y_i} y_i^T \Sigma^{-1} y_i \frac{\partial y_i}{\partial \mu_k} \\ &= -1 \times \left(\Sigma^{-1} + \Sigma^{-T}\right) y_i \end{aligned} \tag{8.14}$$

Substituting the expression in equation 8.14, we get the following equation.

$$\begin{aligned} \frac{\partial}{\partial \mu_k} l(\mu_k, \Sigma_k) &\propto -\frac{1}{2}\sum_i r_{ik}\left[-2\Sigma^{-1}(x_i - \mu_k)\right] \\ &= \Sigma^{-1}\sum_i r_{ik}(x_i - \mu_k) = 0 \end{aligned} \tag{8.15}$$

This implies the following equation.

$$\sum_i r_{ik}(x_i - \mu_k) = 0 \implies \mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}} \tag{8.16}$$

To compute optimum Σ_k, we can use the trace identity.

$$x^T A x = \mathrm{tr}\left(x^T A x\right) = \mathrm{tr}\left(x x^T A\right) = \mathrm{tr}\left(A x x^T\right) \tag{8.17}$$

Using $\lambda = \Sigma - 1$ notation, we have the following.

$$\begin{aligned} l(\Lambda) &\propto -\frac{1}{2}\sum_i r_{ik} \log|\Lambda| - \frac{1}{2}\sum_i r_{ik} \mathrm{tr}\left[(x_i - \mu)(x_i - \mu)^T \Lambda\right] \\ &= -\frac{1}{2}\sum_i r_{ik} \log|\Lambda| - \frac{1}{2}\mathrm{tr}(S_\mu \Lambda) \end{aligned} \tag{8.18}$$

Taking the matrix derivative, we get the following equation.

$$\begin{aligned} \frac{\partial l(\Lambda)}{\partial \Lambda} &= -\frac{1}{2}\sum_i r_{ik} \Lambda^{-T} - \frac{1}{2} S_\mu^T = 0 \\ \Lambda^{-1} \sum_i r_{ik} &= S_\mu^T \implies \Sigma = \frac{S_\mu^T}{\sum_i r_{ik}} \\ \Sigma &= \frac{\sum_i r_{ik}(x_i - \mu_k)(x_i - \mu_k)^T}{\sum_i r_{ik}} = \frac{\sum_i r_{ik} x_i x_i^T}{\sum_i r_{ik}} - \mu_k \mu_k^T \end{aligned} \tag{8.19}$$

These equations make intuitive sense; the mean of cluster *k* is weighted by r_{ik} average of all points assigned to cluster *k*, while the covariance is the weighted empirical scatter matrix. We are now ready to implement the EM algorithm for GMM from scratch! Let's start by looking at the pseudo-code in figure 8.3.

The `GMM` class consists of three main functions: `gmm_em`, `estep`, and `mstep`. In `gmm_em`, we initialize the means using the K-means algorithm and initialize covariances as identity matrices. Next, we iterate between the `estep` that computes responsibilities (aka soft assignments) of data point *i* to each of the *k* clusters and the `mstep` that takes the responsibilities as input and computes model parameters: mean, covariance, and mixture proportions for each cluster in the Gaussian mixture model (GMM). The code in the `estep` and `mstep` functions follows the derivations in the text. In the following code

```
class GMM
function gmm_em($X, k$):
$\pi_{init} = \frac{1}{K}$
$\mu_{init}$ = KMeans($X, k$) ← Initializes with K-means
$\Sigma_{init} = I_{d \times d}$
for iter = $1, 2, \ldots,$ max_iter
    $r_{ik}$ = estep($\pi_k, \mu_k, \Sigma_k, X$) ← Computes responsibilities
    $\pi_k, \mu_k, \Sigma_k$ = mstep($r_{ik}, X$) ← Computes model parameters
end for
return $\pi_k, \mu_k, \Sigma_k$
function estep($\pi_k, \mu_k, \Sigma_k, X$):
$r_{ik} = \frac{N(x_i|\mu_k,\Sigma_k)\pi_k}{\sum_{k=1}^{K} N(x_i|\mu_k,\Sigma_k)\pi_k}$
return $r_{ik}$
function mstep($r_{ik}, X$):
$\pi_k = \frac{1}{N}\sum_i r_{ik}$
$\mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}$
$\Sigma_k = \frac{\sum_i r_{ik} x_i x_i^T}{\sum_i r_{ik}} - \mu_k \mu_k^T$
return $\pi_k, \mu_k, \Sigma_k$
```

Figure 8.3 EM algorithm for Gaussian mixture model pseudo-code

listing, we will use a synthetic dataset to compare our ground truth GMM parameters with the parameters inferred by the EM algorithm.

Listing 8.2 Expectation maximization for Gaussian mixture models

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn.cluster import KMeans
from scipy.stats import multivariate_normal
from scipy.special import logsumexp
from scipy import linalg

np.random.seed(3)

class GMM:

    def __init__(self, n=1e3, d=2, K=4):
        self.n = int(n)          ← Number of data points
        self.d = d               ← Data dimension
        self.K = K               ← Number of clusters

        self.X = np.zeros((self.n, self.d))

        self.mu = np.zeros((self.d, self.K))
        self.sigma = np.zeros((self.d, self.d, self.K))
        self.pik = np.ones(self.K)/K

    def generate_data(self):     ← GMM generative model
```

```
        alpha0 = np.ones(self.K)
        pi = np.random.dirichlet(alpha0)

        #ground truth mu and sigma
        mu0 = np.random.randint(0, 10, size=(self.d, self.K)) -
        ➥ 5*np.ones((self.d, self.K))
        V0 = np.zeros((self.d, self.d, self.K))
        for k in range(self.K):
            eigen_mean = 0
            Q = np.random.normal(loc=0, scale=1, size=(self.d, self.d))
            D = np.diag(abs(eigen_mean + np.random.normal(loc=0, scale=1,
            ➥ size=self.d)))
            V0[:,:,k] = abs(np.transpose(Q)*D*Q)

        #sample data
        for i in range(self.n):
            z = np.random.multinomial(1,pi)
            k = np.nonzero(z)[0][0]
            self.X[i,:] = np.random.multivariate_normal(mean=mu0[:,k],
            ➥ cov=V0[:,:,k], size=1)

        plt.figure()
        plt.scatter(self.X[:,0], self.X[:,1], color='b', alpha=0.5)
        plt.title("Ground Truth Data"); plt.xlabel("X1"); plt.ylabel("X2")
        plt.show()

        return mu0, V0

    def gmm_em(self):

        kmeans = KMeans(n_clusters=self.K,
        ➥ random_state=42).fit(self.X)              Init mu with k-means
        self.mu = np.transpose(kmeans.cluster_centers_)

        #init sigma
        for k in range(self.K):
            self.sigma[:,:,k] = np.eye(self.d)

        #EM algorithm
        max_iter = 10
        tol = 1e-5
        obj = np.zeros(max_iter)
        for iter in range(max_iter):
            print("EM iter ", iter)
            #E-step
            resp, llh = self.estep()    E step
            #M-step
            self.mstep(resp)    M step
            #check convergence
            obj[iter] = llh
            if (iter > 1 and obj[iter] - obj[iter-1] < tol*abs(obj[iter])):
                break
            #end if
        #end for
        plt.figure()
        plt.plot(obj)
        plt.title('EM-GMM objective'); plt.xlabel("iter");
        ➥ plt.ylabel("log-likelihood")
        plt.show()

    def estep(self):

        log_r = np.zeros((self.n, self.K))
        for k in range(self.K):
            log_r[:,k] = multivariate_normal.logpdf(self.X,
            ➥ mean=self.mu[:,k], cov=self.sigma[:,:,k])
        #end for
        log_r = log_r + np.log(self.pik)
        L = logsumexp(log_r, axis=1)
        llh = np.sum(L)/self.n  #log likelihood
        log_r = log_r - L.reshape(-1,1) #normalize
        resp = np.exp(log_r)
        return resp, llh

    def mstep(self, resp):

        nk = np.sum(resp, axis=0)
        self.pik = nk/self.n
        sqrt_resp = np.sqrt(resp)
        for k in range(self.K):
            #update mu
            rx = np.multiply(resp[:,k].reshape(-1,1), self.X)
            self.mu[:,k] = np.sum(rx, axis=0) / nk[k]

            #update sigma
            Xm = self.X - self.mu[:,k]
            Xm = np.multiply(sqrt_resp[:,k].reshape(-1,1), Xm)
            self.sigma[:,:,k] = np.maximum(0, np.dot(np.transpose(Xm), Xm)
            ➥ / nk[k] + 1e-5 * np.eye(self.d))
        #end for

if __name__ == '__main__':

    gmm = GMM()
    mu0, V0 = gmm.generate_data()
    gmm.gmm_em()


    for k in range(mu0.shape[1]):
        print("cluster ", k)
        print("-----------")
        print("ground truth means:")
        print(mu0[:,k])
        print("ground truth covariance:")
        print(V0[:,:,k])
    #end for

    for k in range(mu0.shape[1]):
        print("cluster ", k)
        print("-----------")
        print("GMM-EM means:")
        print(gmm.mu[:,k])
        print("GMM-EM covariance:")
        print(gmm.sigma[:,:,k])

      plt.figure()
    ax = plt.axes()
    plt.scatter(gmm.X[:,0], gmm.X[:,1], color='b', alpha=0.5)

    for k in range(mu0.shape[1]):

        v, w = linalg.eigh(gmm.sigma[:,:,k])
        v = 2.0 * np.sqrt(2.0) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])

        # plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180.0 * angle / np.pi  # convert to degrees
        ell = mpl.patches.Ellipse(gmm.mu[:,k], v[0], v[1], 180.0 + angle,
        ➥ color='r', alpha=0.5)
        ax.add_patch(ell)

        # plot cluster centroids
        plt.scatter(gmm.mu[0,k], gmm.mu[1,k], s=80, marker='x', color='k',
        ➥ alpha=1)
    plt.title("Gaussian Mixture Model"); plt.xlabel("X1"); plt.ylabel("X2")
       plt.show()
```

As we can see from the output, the cluster means and covariances match closely to the ground truth.

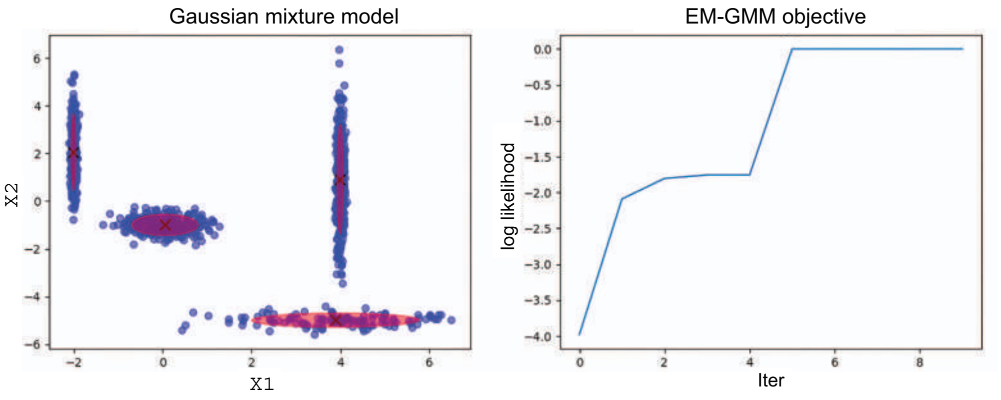

Figure 8.4 EM-GMM clustering results (left) and log likelihood objective function (right)

Figure 8.4 (left) shows the inferred Gaussian mixture overlayed with the data. We see that the Gaussian ellipses closely fit the data. This can also be confirmed by a monotonic increase of the log likelihood objective in figure 8.4 (right). In the next section, we will explore two popular dimensionality reduction techniques: principal component analysis and T-distributed stochastic neighbor embedding.

8.3 Dimensionality reduction

It is often useful to project high-dimensional data $x \in R^D$ onto lower-dimensional subspace $z \in R^L$ in a way that preserves the unique characteristics of the data. In other words, it is desirable to capture the essence of the data in the low dimensional projection. For example, if you were to train word embeddings on the Wikipedia corpus and you were trying to understand the relationships between different word vectors, it would be much easier to visualize the relationships in two dimensions by means of dimensionality reduction. In this section, we will examine two ML techniques for dimensionality reduction: principal component analysis (PCA) and t-SNE.

8.3.1 Principal component analysis

In principal component analysis (PCA), we would like to project our data vector $x \in R^D$ onto a lower dimensional vector $z \in R^L$ with $L < D$ such that the variance of the projected data is maximized. Maximizing the variance of the projected data is the core principle of PCA, and it allows us to preserve the unique characteristics of our data. We can measure the quality of our projection as a reconstruction error.

$$E = \frac{1}{N}\sum_{i=1}^{N} ||x_i - \hat{x}_i||^2 = ||X - WZ||_F^2 \tag{8.20}$$

Here, $\hat{x}_i$ of size $D \times 1$ is the reconstructed vector lifted to the higher dimensional space, z_i of size $L \times 1$ is the lower dimensional principal component vector, and $W$is an orthonormal matrix of size $D \times L$. Recall that an orthonormal matrix is a real square matrix whose transpose is equal to its inverse (i.e., $W^TW = WW^T = I$). In other words, if our PCA projection is $z_i = W^Tx_i$, then $\hat{x}_i = Wz_i$. We will show that the optimal projection matrix W (one that maximizes the variance of projected data) is equal to a matrix of L eigenvectors corresponding to the largest eigenvalues of the empirical covariance matrix $\hat{\Sigma} = 1/N\Sigma x_i x_i^T$.

We can write down the variance of the projected data as follows.

$$\frac{1}{N}\sum_{i=1}^{N} z_i^2 = \frac{1}{N}\sum_{i=1}^{N} w^T x x^T w = w^T \hat{\Sigma} w \tag{8.21}$$

We would like to maximize this quantity subject to orthonormal constraint (i.e., $||w||^2 = 1$). We can write down the Lagrangian as follows.

$$\max J(w) = w^T \hat{\Sigma} w + \lambda \left(w^T w - 1\right) \tag{8.22}$$

Taking the derivative and setting it to zero gives us the following expression.

$$\begin{aligned} \frac{\partial}{\partial w} J(w) &= 2\hat{\Sigma} w - 2\lambda w = 0 \\ \hat{\Sigma} w &= \lambda w \end{aligned} \tag{8.23}$$

Thus, the direction that maximizes the variance is the eigenvector of the covariance matrix. Left multiplying by w and using orthonormality constraint, we get $w^T \hat{\Sigma} w = \lambda$. Therefore, to maximize the variance for the first principal component, we want to choose an eigenvector that corresponds to the largest eigenvalue. We can repeat the above procedure by subtracting the first principal component from x_i, and we'll discover that $\hat{\Sigma} w^2 = \lambda w^2$. By induction, we can show that the PCA matrix WDL consists of L eigenvectors corresponding to the largest eigenvalues of the empirical covariance matrix $\hat{\Sigma}$. We are now ready to implement the PCA algorithm from scratch! The pseudo-code in figure 8.5 summarizes the algorithm.

```
class PCA
    function transform($X$, $K$):
        $\Sigma$ = covariance_matrix($X$)   ← Computes the empirical covariance
        $V, \lambda$ = eig($\Sigma$)   ← Eigenvalue decomposition
        idx = argsort($\lambda$)   ← Sorts from largest to smallest
        $V_{pca} = V$[idx][: K]   ┐
        $\lambda_{pca} = \lambda$[idx][: K]   ┘→ Selects the top K eigen vectors and eigen values
        $X_{pca} = XV_{pca}$   ← Projects the data onto principal components
        return $X_{pca}$
```

Figure 8.5 Principal component analysis pseudo-code

The main function in the `PCA` class is called `transform`. We first compute the empirical covariance matrix from the high dimensional data matrix X. Next, we compute the eigenvalue decomposition of the covariance matrix. We sort the eigenvalues from largest to smallest and use the sorted index to select the top K largest eigenvalues and their corresponding eigenvectors. Finally, we compute the PCA representation of our data by multiplying the data matrix with the matrix of top K eigenvalues. In the following code listing, we project a random `d`-dimensional matrix onto two principal components.

Listing 8.3 Principal component analysis

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

class PCA():

    def __init__(self, n_components = 2):
        self.n_components = n_components

    def covariance_matrix(self, X, Y=None):
        if Y is None:
            Y = X
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0))
        ➥ .T.dot(Y - Y.mean(axis=0))
        return covariance_matrix

    def transform(self, X):
        Sigma = self.covariance_matrix(X)
        eig_vals, eig_vecs = np.linalg.eig(Sigma)

        idx = eig_vals.argsort()[::-1]
        eig_vals = eig_vals[idx][:self.n_components]
        eig_vecs = np.atleast_1d(eig_vecs[:,idx])[:, :self.n_components]

        X_transformed = X.dot(eig_vecs)

        return X_transformed

if __name__ == "__main__":

    n = 20
    d = 5
    X = np.random.rand(n,d)

    pca = PCA(n_components=2)
    X_pca = pca.transform(X)

    print(X_pca)

    plt.figure()
    plt.scatter(X_pca[:,0], X_pca[:,1], color='b', alpha=0.5)
    plt.title("Principal Component Analysis"); plt.xlabel("X1");
    ➥ plt.ylabel("X2")
    plt.show()
```

Sorts from largest to smallest eigenvalue

Projects the data onto principal components

Figure 8.6 shows a plot of the first two principal components applied to a random matrix. Recall that in PCA, the variance of the projected data is maximized; therefore, we can discover trends and patterns in the projected data.

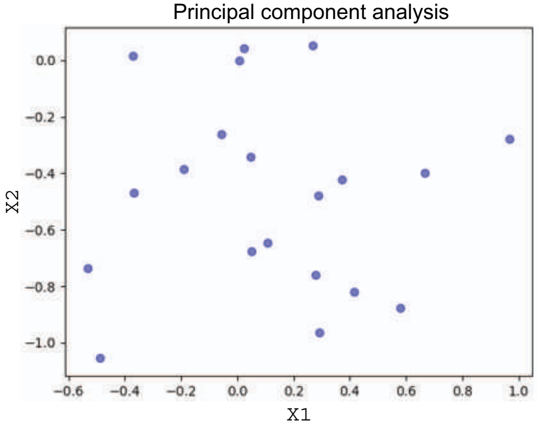

Figure 8.6 PCA projection of a random matrix

8.3.2 t-SNE manifold learning on images

Images are high dimensional objects that live on manifolds. A *manifold* is a topological space that locally resembles Euclidean space. By modeling image spaces as manifolds, we can study their geometric properties. We can visualize high dimensional objects with the help of an embedding. We consider two such embeddings: t-SNE and Isomap on the MNIST digits dataset.

Listing 8.4 t-SNE manifold on the MNIST digits dataset

```
import numpy as np
import matplotlib.pyplot as plt

from time import time
from sklearn import manifold

from sklearn.datasets import load_digits
from sklearn.neighbors import KDTree

def plot_digits(X):

    n_img_per_row = np.amin((20, np.int(np.sqrt(X.shape[0]))))
    img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
    for i in range(n_img_per_row):
        ix = 10 * i + 1
        for j in range(n_img_per_row):
            iy = 10 * j + 1
            img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row +
            ➥ j].reshape((8, 8))

    plt.figure()
    plt.imshow(img, cmap=plt.cm.binary)
    plt.xticks([])
    plt.yticks([])
    plt.title('A selection from the 64-dimensional digits dataset')

def mnist_manifold():

    digits = load_digits()

    X = digits.data
    y = digits.target

    num_classes = np.unique(y).shape[0]

    plot_digits(X)

    #TSNE
    #Barnes-Hut: O(d NlogN) where d is dim and N is the number of samples
    #Exact: O(d N^2)
    t0 = time()
    tsne = manifold.TSNE(n_components = 2, init = 'pca', method =
    ➥ 'barnes_hut', verbose = 1)
    X_tsne = tsne.fit_transform(X)
    t1 = time()
    print('t-SNE: %.2f sec' %(t1-t0))
    tsne.get_params()

    plt.figure()
    for k in range(num_classes):
        plt.plot(X_tsne[y==k,0], X_tsne[y==k,1],'o')
    plt.title('t-SNE embedding of digits dataset')
    plt.xlabel('X1')
    plt.ylabel('X2')
    axes = plt.gca()
    axes.set_xlim([X_tsne[:,0].min()-1,X_tsne[:,0].max()+1])
    axes.set_ylim([X_tsne[:,1].min()-1,X_tsne[:,1].max()+1])
    plt.show()

    #ISOMAP
    #1. Nearest neighbors search: O(d log k N log N)
    #2. Shortest path graph search: O(N^2(k+log(N))
    #3. Partial eigenvalue decomposition: O(dN^2)

    t0 = time()
    isomap = manifold.Isomap(n_neighbors = 5, n_components = 2)
    X_isomap = isomap.fit_transform(X)
    t1 = time()
    print('Isomap: %.2f sec' %(t1-t0))
    isomap.get_params()

    plt.figure()
    for k in range(num_classes):
        plt.plot(X_isomap[y==k,0], X_isomap[y==k,1], 'o', label=str(k),
        ➥ linewidth = 2)
    plt.title('Isomap embedding of the digits dataset')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()

    #Use KD-tree to find k-nearest neighbors to a query image
    kdt = KDTree(X_isomap)
    Q = np.array([[-160, -30],[-102, 14]])
    kdt_dist, kdt_idx = kdt.query(Q,k=20)
    plot_digits(X[kdt_idx.ravel(),:])

if __name__ == "__main__":
    mnist_manifold()
```

Figure 8.7 shows a t-SNE embedding with 10 clusters in 2D, where each cluster corresponds to a digit from 0 to 9. We can see that without labels, we are able to discover 10 clusters that use t-SNE embedding in the two-dimensional space. Moreover, we expect adjacent clusters of digits to be similar to each other. The image on the right-hand side of figure 8.7 shows sample digits from two adjacent clusters (digit 0 and digit 6). We can visualize adjacent clusters by constructing a KD tree to find KNN to a query point.

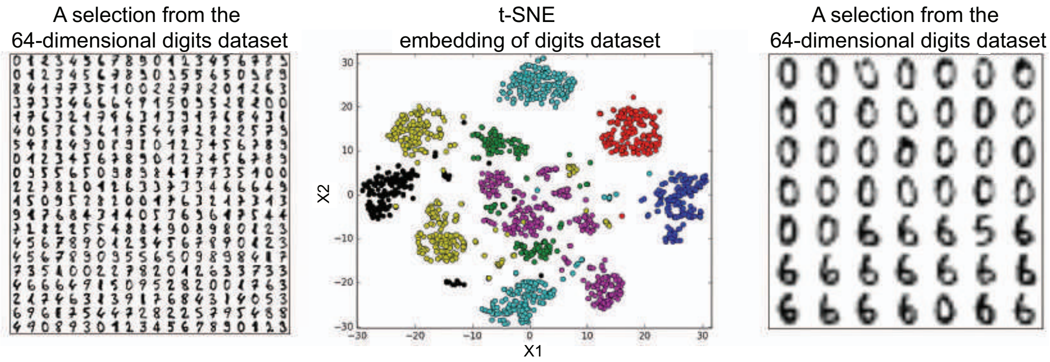

Figure 8.7 t-SNE embedding showing 10 clusters in 2D, where each cluster corresponds to a digit from 0 to 9

It's important to be aware of some of the pitfalls of t-SNE. For example, t-SNE results may vary based on the perplexity hyperparameter. The t-SNE algorithm also doesn't always produce similar outputs on successive runs and there are additional hyperparameters related to the optimization process. Moreover, the cluster sizes (as measured by standard deviation) and distances between clusters might not mean anything. Thus, the high flexibility of t-SNE also makes the results of the algorithm tricky to interpret.

8.4 Exercises

8.1 Show that the Dirichlet distribution Dir$(\theta \mid \alpha)$ is a conjugate prior to the multinomial likelihood by computing the posterior. How does the shape of the posterior vary as a function of the posterior counts?

8.2 Explain the principle behind k-means++ initialization.

8.3 Prove the cyclic permutation property of the trace: `tr(ABC) = tr(BCA) = tr(CAB)`.

8.4 Compute the runtime of the principal component analysis (PCA) algorithm.

Summary

- Unsupervised learning takes place when no training labels are available.
- The Dirichlet process K-means is a Bayesian nonparametric extension of the K-means algorithm, in which the number of clusters grows with data.
- Gaussian mixture models are commonly used to model complex density distributions. GMM parameters (means, covariances, and mixture proportions) can be inferred using the expectation maximization algorithm.
- Expectation-Maximization is an iterative algorithm that alternates between inferring the missing values given the parameters (E step) and then optimizing the parameters given filled-in data (M step).

- In dimensionality reduction, we project high dimensional data into a lower dimensional subspace in a way that preserves the unique characteristics of the data.
- In principal component analysis algorithm, the variance of the projected data is maximized.
- Common pitfalls of the t-SNE algorithm include variability of results due to the perplexity hyperparameter, dissimilar outputs on successive runs, a lack of interpretability of cluster sizes, and large distances between clusters.