Computational Intelligence Lab 2020

Yu Fei

August 16, 2020

Contents

1

Linear Autoencoder

1.1 Dimension Reduction

In machine learning, when we work with data, we often start with very high-dimensional representation (raw data), like mega-pixel images or raw audio data. A question raises naturally is that is it really necessary to have so many dimensions to describe our data. This motivates this field of dimension reduction. Actually dimension reduction lies in the heart of many fields in machine learning and data science:

- visualization - e.g. 2D or 3D
- data compression - fewer coefficients
- signal recovery - discard irrelevant information(noise)
- discover modes of variation - intrinsic properties of data
- feature discovery - learn better representations
- generative models - latent variables (knowing low-dim representation often helps to synthesize data)

For a typical dimension reduction problem, what we are given are n high-dimension data points$\{\boldsymbol{x}_i \in \mathbb{R}^m\},\ i = 1, \dots, n$. We want to find low-dimensional representation $\{\boldsymbol{z}_i \in \mathbb{R}^k\}$ for $\{\boldsymbol{x}_i\},\ i = 1, \dots, n$ in a predefined reduced dimension number $k \ll m$.

An example for dimension reduction is the Eigenfaces method for face images. A 2D pixel face image $\boldsymbol{x}_i \in \mathbb{R}^{100\times 100}$ after vectorization is a vector in $\mathbb{R}^{10000}$. Eigenfaces suggests to approximate each face with a weighted superposition of few (say 4) basis images. The 4 coefficients are our desired low-dimensional representation for a given image. The fact that we can approximate a face image well with a few basis images also indicates that we have understood something like statistical properties about faces.

1.1.1 Linear Dimension Reduction

In Linear dimension reduction, the relation between the low-dimensional representation $\boldsymbol{z}_i$ and its corresponding $\boldsymbol{x}_i$ is governed by some (fixed) matrix C:

$$\boldsymbol{z}_i = C\boldsymbol{x}_i, \ C \in \mathbb{R}^{k\times m}$$

where we can regard C as a linear map $\mathbb{R}^m \to \mathbb{R}^k$. Each entry of the feature vector $\boldsymbol{z}$ is a linear combination of input variables

$$\boldsymbol{z} = C\boldsymbol{x} \Longleftrightarrow z_r = \sum_{s=1}^{m} c_{rs}x_s \ \forall r \in [k], \ C = (c_{rs})$$

In neural network terminology, each z_r is a *linear unit* that computes a linear function of its inputs with weight vector $\boldsymbol{c}_r = (c_{r1}, \dots, c_{rm})^T \in \mathbb{R}^m$, which is the r-th row of C.

Now we take a neural network view. Think about a (deep) neural network as in Figure 1.1. In supervised learning, given targets $\boldsymbol{y}$ and the inputs $\boldsymbol{x}$, the training of such a neural network can be seen as learning (better) representations along the way. The activation of middle layers with fewer dimensions are ideal low-dim representations for the original input. Although this view from neural network gives some interesting insights, in this chapter we will focus on the unsupervised learning case, where we don't have access to such targets for our $\boldsymbol{x}$.

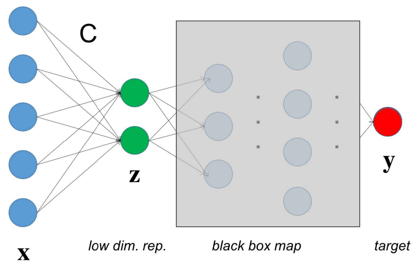


Figure 1.1: Dimension reduction from a neural network view

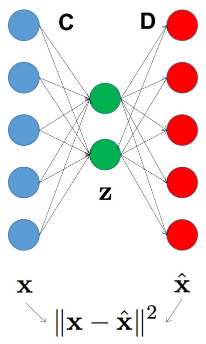


Figure 1.2: Linear Autoencoder

1.2 Linear Autoencoder

A Linear autoencoder can be represented neural network with 1 hidden layer without activation function (Figure 1.2). The idea of linear autoencoder is to learn the low-dim representation by learning an identity map. This is done in a *fully unsupervised* way. Specifically, we first map the input linearly to a lower dimension, then use a linear reconstruction to recover the original input. The parameters θ of a linear autoencoder consist of two parts: an encoder matrix $C \in \mathbb{R}^{k\times m}$ and a decoder matrix $D \in \mathbb{R}^{m\times k}$ which is a linear reconstruction map. Given the input $\boldsymbol{x}$ and the model parameters θ, the output of the linear autoencoder can therefore be written as $\hat{\boldsymbol{x}}(\theta) = DC\boldsymbol{x}$. The criteria people often use to evaluate the reconstruction is the squared reconstruction loss:

$$l(\boldsymbol{x};\theta) = \frac{1}{2}\|\boldsymbol{x} - \hat{\boldsymbol{x}}(\theta)\|^2$$

Since we often don't care about the order of entries in $\boldsymbol{x}$, linear autoencoders are trained with sample reconstruction error:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\boldsymbol{x}_i;\theta)$$

By optimizing over $J(\theta)$, a linear autoencoder learn an approximate identity map over a specific data distribution, and the *bottleneck* layer $\boldsymbol{z}$ is the desired low-dim representation.

Remark 1.1 *By definition, as long as $k < m$, an autoencoder in Figure 1.2 cannot learn an identity map for everything in $\mathbb{R}^m$, but fortunately we only care about inputs from a specific distribution.*

Mathematically, the linear autoencoder defines a linear map

$$F : \mathbb{R}^m \to \mathbb{R}^m,\ F := DC$$

Ideally, we want F to be approximately an identity map, i.e. $F \approx I$. However, the bottleneck places an upper bound on the rank of F, making F a *low-rank approximation* of the identity map over a specific data distribution.

The *rank* of a linear map $A : \mathbb{R}^k \to \mathbb{R}^l$ is defined as the dimension of the image of A:

$$rank(A) := dim(im(A)) \leq min\{k, l\}$$

Note that the ranks of matrices and the rank of their product satisfy

$$rank(AB) \leq \min\{rank(A), rank(B)\}$$

and that a matrix M has a decomposition $M = AB$ with $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$, if and only if $rank(M) \leq k$:

$$\exists A \in \mathbb{R}^{m\times k},\ B \in \mathbb{R}^{k\times n},\ M = AB \Longleftrightarrow rank(M) \leq k$$

Then the low-rank approximation F that linear autoencoder performs satisfies

$$rank(F) \leq \min\{rank(C), rank(D)\} \leq k$$

At this point, we can already try to solve this problem using methods like training a neural network to find C and D. However, if possible, we'd like to gain some understanding about the theoretical limit that the reconstruction quality can achieve. To answer this question, we introduce some extra notation and concepts first.

By writing data and the corresponding approximations in matrix forms $X = [\boldsymbol{x}_1 \dots \boldsymbol{x}_n]$, $\hat{X} = [\hat{\boldsymbol{x}}_1 \dots \hat{\boldsymbol{x}}_n]$, we can rewrite the error function:

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} \|\boldsymbol{x} - \hat{\boldsymbol{x}}(\theta)\|^2 = \frac{1}{2n} \|X - \hat{X}(\theta)\|_F^2$$

where $\|A\|_F := \|vec(A)\|_2 = \sqrt{\sum_{ij} a_{ij}^2}$ is the Frobenius norm. Then optimizing $J(\theta)$ over all possible matrix C and D is equivalent to solving a standard low-rank approximation problem.

Definition 1.2 *Given $X \in \mathbb{R}^{m \times n}$ and a rank constraint k. Low-rank approximation problem with fit measured by the Frobenius norm is defined as:*

$$\underset{\hat{X}:\ rank(\hat{X}) \leq k}{\text{minimize}} \quad \|X - \hat{X}\|_F$$

The Eckart-Young theorem gives a bound on the reconstruction error of low-rank approximation problem and also a analytic solution that can achieve this bound.

Theorem 1.3 *(Eckart-Young) Let $A \in \mathbb{R}^{m \times n}$ be a real matrix, for $k \leq \min\{m, n\}$:*

$$\|A - U\Sigma_k V^T\|_F^2 = \min_{\hat{A}:\ rank(\hat{A}) \leq k} \|A - \hat{A}\|_F^2 = \sum_{l=k+1}^{\min\{m,n\}} \sigma_l^2$$

where $A = U\Sigma V^T$ is the singular value decomposition of A and Σ_k is the truncated diagonal matrix of singular values.

Proof Without loss of generality, we assume that $m \geq n$. Σ is an $m \times n$ diagonal matrix with entries $(\sigma_1, \dots, \sigma_n)$ such that $\sigma_1 \geq \dots \geq \sigma_n \geq 0$.

To prove Eckart-Young Theorem for Frobenius Norm, we first prove the theorem for spectral norm. We claim that the best rank k approximation of A in the spectral norm is given by

$$A_k = \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T$$

where $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are columns of U and V, respectively. Since we have

$$\|A - A_k\| = \left\| \sum_{i=k+1}^{n} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \right\| = \sigma_{k+1}$$

Therefore all we need to show is given any rank k reconstruction $B_k = XY^T$, where $X, Y \in \mathbb{R}^{m\times k}$, it satisfies

$$\|A - A_k\| = \sigma_{k+1} \leq \|A - B_k\|$$

Since Y has k columns, there exists a linear combination $\boldsymbol{w}$ of $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_{k+1}\}$ such that

$$\boldsymbol{w} = \sum_{i=1}^{k+1} \alpha_i \boldsymbol{v}_i,\ Y^T\boldsymbol{w} = 0,\ \|\boldsymbol{w}\| = 1$$

therefore we obtain

$$\|A - B_k\|^2 \geq \|(A - B_k)\boldsymbol{w}\|^2 = \|A\boldsymbol{w}\|^2 = \sum_{i=1}^{k+1} \alpha_i^2\sigma_i^2 \geq \sigma_{k+1}^2 \sum_{i=1}^{k+1} \alpha_i^2 = \sigma_{k+1}^2$$

which concludes the proof for spectral norm.

For Frobenius norm, since

$$\|A - A_k\|_F^2 = \left\| \sum_{i=k+1}^{n} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \right\|_F^2 = \sum_{i=k+1}^{n} \sigma_i^2$$

all we have to prove is given any rank k reconstruction $B_k = XY^T$, where $X, Y \in \mathbb{R}^{m\times k}$, it satisfies

$$\|A - A_k\| = \sum_{i=k+1}^{n} \sigma_i^2 \leq \|A - B_k\|$$

Assume $A = A' + A''$. A'_k and A''_k are the corresponding rank k reconstruction given by SVD of A' and A'' respectively. Then $\forall 1 \leq i, j \leq n$

$$\begin{aligned}
\sigma_i(A') + \sigma_j(A'') &= \sigma_1(A' - A'_{i-1}) + \sigma_1(A'' - A''_{j-1}) \\
&\geq \sigma_1(A - A'_{i-1} - A''_{j-1}) \text{ (Triangle ineq. of spectral norm)} \\
&\geq \sigma_1(A - A_{i+j-2}) \ (*) \\
&= \sigma_{i+j-1}(A)
\end{aligned}$$

where $(*)$ used fact that $rank(A'_{i-1} + A''_{j-1}) \leq rank(A_{i+j-2})$ and applied Eckart Young theorem for spectral norm. Now choose $A' = A - B_k$ and note that $\sigma_{k+1}(B_k) = 0$ we get

$$\sigma_i(A - B_k) \geq \sigma_{k+i}(A)$$

Therefore,

$$\|A - B_k\|_F^2 = \sum_{i=1}^{n} \sigma_i(A - B_k)^2 \geq \sum_{i=k+1}^{n} \sigma_i(A)^2 = \|A - A_k\|_F^2$$

as required. □

1.3 Singular Value Decomposition

Any $m \times n$ matrix A can be decomposed into

$$A = U \cdot \Sigma \cdot V^T$$

where $U \in \mathbb{R}^{m\times m}, V \in \mathbb{R}^{n\times n}$ are orthogonal, i.e. $UU^T = I_m$, $VV^T = I_n$ and $\Sigma \in \mathbb{R}^{m\times n}$ diagonal.

The columns of U and V are called left/right singular vectors of A. Let $s := min\{m, n\}$, then we can write Σ as

$$\Sigma = \text{diag}(\sigma_1, \dots, \sigma_s),\ \ \sigma_1 \geq \dots \geq \sigma_s \geq 0$$

where σ_i's are *singular values* of A. Note that by diagonal we mean Σ is a diagonal matrix padded with zeros to match dimensionality.

The number of non-zero singular values is equal to the rank of A

$$rank(A) = r \iff \sigma_r > 0 \wedge \sigma_{r+1} = \dots = \sigma_s = 0$$

Usually the non-zeros singular values are distinct, but σ_i *degenerate* when σ_i is with two (or more) linearly independent left (or right) singular vectors. In the degenerate case, σ_i is actually associated with a subspace not a particular set of orthonormal vectors. Specifically, the singular vectors for non-degenerate σ_i are unique up to sign, while for degenerate σ_i the span of orthonormal basis are unique.

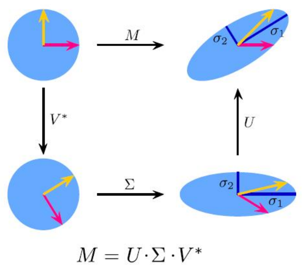


Figure 1.3: Illustration of SVD in $\mathbb{R}^2$

In the special case when A is a real square matrix, i.e. $m = n$, A can be interpreted as a linear transformation $\boldsymbol{x} \rightarrow A\boldsymbol{x}$ of $\mathbb{R}^m$. Orthogonal matrix U and V represent rotations or reflections and Σ represents a scaling transform. Thus the SVD decomposition has a clear interpretation that it decompose any invertible linear transformation into a composition of a rotation or reflection (V^T), followed by a coordinate-wise scaling (Σ), followed by another rotation or reflection (U).

Remark 1.4 *SVD can be computed in an iterative manner with time complexity $O(mn^2)$, assuming that $m \geq n$.*

1.4 Continuation of Linear Autoencoder

With the tool of SVD and Eckart Young Theorem (Theorem 1.3), we now can compute the optimal linear autoencoder. Define $U_k = [\boldsymbol{u}_1 \boldsymbol{u}_2 \dots \boldsymbol{u}_k] \in \mathbb{R}^{m \times k}$ as the first k columns of U. We have the following proposition:

Proposition 1.5 *$C^* = U_k^T$ and $D^* = U_k$ yields minimal reconstruction error for a linear autoencoder with k hidden units.*

Proof With $X = U\Sigma V^T$ $s = \min\{m, n\}$ and $\hat{X} = DCX$ we have

$$\hat{X} = U_k U_k^T U \Sigma V^T = \left(\sum_{i=1}^{k} \boldsymbol{u}_i \boldsymbol{u}_i^T\right)\left(\sum_{i=1}^{s} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T\right) = \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T = U\Sigma_k V^T,$$

where the third equality follows $\boldsymbol{u}_i^T \boldsymbol{u}_j = \delta_{ij}$. By Theorem 1.3, $\hat{X}$ is optimal in terms of Frobenius norm as the reconstruction measure. □

Remark 1.6 *The optimal linear autoencoder via SVD is not the unique optimal solution, since for any invertible matrix $A \in \mathbb{R}^{k \times k}$, $C = AU_k^T$ and $D = U_k A^{-1}$ also give the optimal reconstruction error. This gives a particularly interesting insight into the interpretability of the features found by training a linear autoencoder. We can expect that there's randomness (some invertible A introduced by randomness in our model) in the solution.*

A way to alleviate the problem of uniqueness mention in Remark 1.6 is to do so called *weight sharing*. Specifically, instead of optimizing over both C and D, we require $D = C^T$. Note that this does not reduce the modeling power. Now the ambiguity in the optimal solution reduces to orthogonal group:

$$A^{-1} = A^T, \text{ i.e. } A \in O(k)$$

In another word, the mapping $\boldsymbol{x} \mapsto \boldsymbol{z}$ now is uniquely determined up to rotations (permutations, reflections). It's the same orthogonal subspace of the first k eigenvectors (different basis or axes).

2

Principal Component Analysis

In the last chapter, we discussed linear dimension reduction via linear autoencoder. Now we try to understand linear dimension reduction in a principled manner. We start with 1-dimension case, i.e. we want to approximate a group of points via points on a line. Then we generalize it to multiple dimensions, which yields exactly the idea of PCA. Finally we introduce a simple algorithm to find the dominant eigenvector of a matrix, followed by some examples and comparison with Linear Autoencoder we discussed in the last chapter.

2.1 1D Linear Case

2.1.1 Line in $\mathbb{R}^m$

Mathematically, we can represent a line in $\mathbb{R}^m$ in the following parametric form:

$$\boldsymbol{\mu} + \mathbb{R}\boldsymbol{u} \equiv \{\boldsymbol{v} \in \mathbb{R}^m : \exists z \in \mathbb{R} \ s.t. \ \boldsymbol{v} = \boldsymbol{\mu} + z\boldsymbol{u}\}$$

where $\boldsymbol{\mu}$ is the offset or shift, $\boldsymbol{u}$ is the direction vector such that $\|\boldsymbol{u}\| = 1$. Note that representation is not unique since a scaling on $\boldsymbol{u}$ and a shifting on $\boldsymbol{\mu}$ along $\boldsymbol{u}$ will not change the line.

2.1.2 Approximation of a single point: orthogonal projection

Now consider the task to approximate a data point $\boldsymbol{x} \in \mathbb{R}^m$ by a point on the line. It natural to request that our approximation should be close to the original point in Euclidean distance. Formally, we try to solve

$$\arg\min_{z \in \mathbb{R}} \|\boldsymbol{\mu} + z\boldsymbol{u} - \boldsymbol{x}\|^2 \tag{2.1}$$

or equivalently,

$$\arg\min_{\hat{\boldsymbol{x}} \in \boldsymbol{\mu}+\mathbb{R}\boldsymbol{u}} \|\hat{\boldsymbol{x}} - \boldsymbol{x}\|^2$$

To solve (2.1), we request the first order optimality condition:

$$\frac{d}{dz}\|\boldsymbol{\mu} + z\boldsymbol{u} - \boldsymbol{x}\|^2 = 2\langle \boldsymbol{\mu} + z\boldsymbol{u} - \boldsymbol{x}, \boldsymbol{u}\rangle \overset{!}{=} 0$$

this yields solutions

$$z = \langle \boldsymbol{x} - \boldsymbol{\mu}, \boldsymbol{u}\rangle$$
$$\hat{\boldsymbol{x}} = \boldsymbol{\mu} + \langle \boldsymbol{x} - \boldsymbol{\mu}, \boldsymbol{u}\rangle \boldsymbol{u}$$

which is the *orthogonal projection* of $\boldsymbol{x}$ to the line.

2.1.3 Optimal Line for multiple points

Assume we are given data points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^m$. With the orthogonal projection result, the optimal approximation of $\{\boldsymbol{x}_i\}$ by a line $(\boldsymbol{u}, \boldsymbol{\mu})$ is given by $\{\boldsymbol{\mu} + \langle \boldsymbol{x}_i - \boldsymbol{\mu}, \boldsymbol{u}\rangle\}$. Therefore the optimal line can be found by solving

$$\arg\min_{(\boldsymbol{u},\boldsymbol{\mu})} \left[\frac{1}{2n}\sum_{i=1}^{n} \|\boldsymbol{\mu} + \langle \boldsymbol{x}_i - \boldsymbol{\mu}, \boldsymbol{u}\rangle \boldsymbol{u} - \boldsymbol{x_i}\|^2\right]$$

using that $\langle \boldsymbol{v}, \boldsymbol{u}\rangle \boldsymbol{u} = (\boldsymbol{u}\boldsymbol{u}^T)\boldsymbol{v}$, it's equivalent to

$$\arg\min_{(\boldsymbol{u},\boldsymbol{\mu})} \left[\frac{1}{2n}\sum_{i=1}^{n} \left\|\left(I - \boldsymbol{u}\boldsymbol{u}^T\right)(\boldsymbol{x}_i - \boldsymbol{\mu})\right\|^2\right] \tag{2.2}$$

Matrix $\left(I - \boldsymbol{u}\boldsymbol{u}^T\right)$ act as a linear map, more specifically, as a projection. Take an argument $\boldsymbol{v}$, by associativity, we get

$$\left(I - \boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{v} = \boldsymbol{v} - \langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u}$$

The RHS is the vector itself minus the projection to the line $\mathbb{R}\boldsymbol{u}$, which is the projection to the orthogonal complement $(\mathbb{R}\boldsymbol{u})^{\perp}$. Also, this projection is *idempotent*, i.e. not changed in value following multiplication by itself, since

$$\left(I - \boldsymbol{u}\boldsymbol{u}^T\right)(\boldsymbol{v} - \langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u}) = \boldsymbol{v} - \langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u} - (\langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u} - \langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u}) = \boldsymbol{v} - \langle \boldsymbol{u}, \boldsymbol{v}\rangle \boldsymbol{u}$$

To solve (2.2), we first request the first order optimality condition for $\boldsymbol{\mu}$

$$\nabla_{\boldsymbol{\mu}} \left[\frac{1}{2n}\sum_{i=1}^{n} \left\|\left(I - \boldsymbol{u}\boldsymbol{u}^T\right)(\boldsymbol{x}_i - \boldsymbol{\mu})\right\|^2\right] \overset{!}{=} 0$$
$$\iff \frac{1}{n}\sum_{i=1}^{n} \left(I - \boldsymbol{u}\boldsymbol{u}^T\right)(\boldsymbol{x}_i - \boldsymbol{\mu}) \overset{!}{=} 0$$
$$\iff \left(I - \boldsymbol{u}\boldsymbol{u}^T\right)(\frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i - \boldsymbol{\mu}) \overset{!}{=} 0$$

Note that $\|\boldsymbol{u}\| = 1$, then $\boldsymbol{u}$ is a eigenvector of $I - \boldsymbol{u}\boldsymbol{u}^T$ with corresponding eigenvalue 0. From this we see that the optimal solution of $\boldsymbol{\mu}$ is not unique. Any $\boldsymbol{\mu}$ satisfying

$$\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i - \boldsymbol{\mu} = k\boldsymbol{u},\ k \in \mathbb{R}$$

will be an optimal solution. We can also see this from another intuitive angle: any $\boldsymbol{\mu}$ lies on the optimal line should be a equivalent in terms of determining a line.

Despite the first order optimality condition cannot determine $\boldsymbol{\mu}$ uniquely, there is a unique solution for all $\boldsymbol{u}$:

$$\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i \equiv \textit{sample mean}$$

which means that all the lines that are optimal in this 1-D setting have to pass through $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i$. Since it will be good to have a "correct" $\boldsymbol{\mu}$ that can guarantee optimality without the dependency on $\boldsymbol{u}$, we simply choose $\boldsymbol{\mu}$ to be the sample mean.

By choosing $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i$, what we are doing is often referred to as *centering* in the data in ML literature. By centering the data, i.e.

$$\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i$$

we identify the center of mass with the origin and restrict the problem to linear (instead of affine) subspaces. This will simplify the derivations and analyses without loss in modeling power. Therefore from now on, w.l.o.g. we assume that the data points are centered.

By centering the data, the optimization problem is left with

$$\begin{aligned}
&\arg\min_{\|\boldsymbol{u}\|=1}\left[\frac{1}{n}\sum_{i=1}^{n}\|\langle\boldsymbol{x}_i,\boldsymbol{u}\rangle\boldsymbol{u} - \boldsymbol{x}_i\|^2\right] \\
\Longleftrightarrow\ &\arg\min_{\|\boldsymbol{u}\|=1}\left[\frac{1}{n}\sum_{i=1}^{n}\|\langle\boldsymbol{x}_i,\boldsymbol{u}\rangle\boldsymbol{u}\|^2 + \|\boldsymbol{x}_i\|^2 - 2\langle\boldsymbol{x}_i,\boldsymbol{u}\rangle^2\right] \\
\Longleftrightarrow\ &\arg\min_{\|\boldsymbol{u}\|=1}\left[-\frac{1}{n}\sum_{i=1}^{n}\langle\boldsymbol{x}_i,\boldsymbol{u}\rangle^2\right]\ \ (\boldsymbol{x}_i \text{ is a const}) \\
\Longleftrightarrow\ &\arg\min_{\|\boldsymbol{u}\|=1}\left[-\frac{1}{n}\sum_{i=1}^{n}\ \boldsymbol{u}^T\boldsymbol{x}_i\boldsymbol{x}_i^T\boldsymbol{u}\right] \\
\Longleftrightarrow\ &\arg\min_{\|\boldsymbol{u}\|=1}\left[-\boldsymbol{u}^T\left(\frac{1}{n}\sum_{i=1}^{n}\ \boldsymbol{x}_i\boldsymbol{x}_i^T\right)\boldsymbol{u}\right]\ \ \text{(Associativity)} \qquad (2.3)
\end{aligned}$$

where the matrix between $\boldsymbol{u}^T$ and $\boldsymbol{u}$ is call *variance-covariance matrix* of the data sample

$$\boldsymbol{\Sigma} \equiv \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i\boldsymbol{x}_i^T = \frac{1}{n}\mathbf{X}\mathbf{X}^T \in \mathbb{R}^{m\times m},\ \mathbf{X} \equiv [\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n]$$

This gives us a very important insight: in the 1-D setting, $\boldsymbol{u}$ depends on the data only through $\boldsymbol{\Sigma}$. This is remarkable, since the dependency between $\boldsymbol{u}$ and all the data points $\mathbf{X}$ can be very complicated and there are a lot of information in $\mathbf{X}$. But (2.3) tells us the only thing matters is $\boldsymbol{\Sigma}$. Consider we want to fit the data in a streaming manner, the above analysis shows that we can update $\boldsymbol{u}$ very efficiently by recomputing $\boldsymbol{\Sigma}$.

Problem (2.3) is a constrained optimization problem. To solve it we introduce the Lagrange multiplier λ and define the following Lagrangian

$$L(\boldsymbol{u},\lambda) = -\boldsymbol{u}^T\boldsymbol{\Sigma}\boldsymbol{u} + \lambda(\boldsymbol{u}^T\boldsymbol{u} - 1) \tag{2.4}$$

Before trying to solve it, we first take another view from the duality view and gain some insight into the Lagrange method. Assume we have a primal player who wants to minimize the objective in (2.3). To remove the constraint on $\boldsymbol{u}$, we introduce a dual player who wants to maximize the objective and would exploit a deviation from the original constraint on $\boldsymbol{u}$. Formally, we consider

$$\min_{\boldsymbol{u}}\max_{\lambda} -\boldsymbol{u}^T\boldsymbol{\Sigma}\boldsymbol{u} + \lambda(\boldsymbol{u}^T\boldsymbol{u} - 1) \tag{2.5}$$

If the primal player (picking $\boldsymbol{u}$ and attempting to minimize the objective) picks $\boldsymbol{u}$ that does not satisfy the original constraint, the dual player (picking λ and trying to maximizing the objective) will be able to drive the objective value to $+\infty$ since there's no constraint on λ. Then is clear that (2.5) is equivalent to (2.3).

With this intuition, we now try to solve (2.4) by again applying the first order optimality condition:

$$\nabla_{\boldsymbol{u}}L(\boldsymbol{u},\lambda) = -2(\boldsymbol{\Sigma}\boldsymbol{u} - \lambda\boldsymbol{u}) \overset{!}{=} 0 \iff \boldsymbol{\Sigma}\boldsymbol{u} = \lambda\boldsymbol{u}$$

This means that minimizing over $\boldsymbol{u}$ requires $\boldsymbol{u}$ to be an eigenvector of $\boldsymbol{\Sigma}$ with the corresponding eigenvalue λ. Since we want to maximize over λ, $\boldsymbol{u}$ should be a **principal** eigenvector of $\boldsymbol{\Sigma}$. So the optimal direction of the line is the principal eigenvector of the sample variance-covariance matrix. Also we have the following extremal characterization

$$\boldsymbol{u} = \arg\max\nolimits_{\boldsymbol{v}:\|\boldsymbol{v}\|=1}\left[\boldsymbol{v}^T\boldsymbol{\Sigma}\boldsymbol{v}\right]$$

Remark 2.1 *Usually, even we can mathematically define and find the eigenvectors and eigenvalues of a matrix, it is still hard to interpret what these vectors and values are. The discussion above give us some insight of the principal eigenvector (of a variance-covariance matrix of centered data) in terms of 1-D PCA. It is the direction that the projected data have the smallest reconstruction error and also the direction of largest data variance (we will see this soon).*

Before we finish this section, we re-interpret the 1-D PCA problem, as suggested in the end of Remark (2.1), in terms of variance maximization in 1-D representation.

Consider the variance of the projected data (remember that we assume centered data, which means the projections (a linear transformation) will also be centered)

$$\mathrm{Var}[z] = \frac{1}{n}\sum_{i=1}^{n} z_i^2 = \frac{1}{n}\sum_{i=1}^{n}\langle \boldsymbol{x}_i, \boldsymbol{u}\rangle^2 = \boldsymbol{u}^T\boldsymbol{\Sigma}\boldsymbol{u}$$

This shows the equivalence of

- Direction of the smallest reconstruction error.
- Direction of the largest data variance.

2.2 Principal Component Analysis

Define the residual as the projection of the data to $(\mathbb{R}\boldsymbol{u})^{\perp}$:

$$\boldsymbol{r}_i := \boldsymbol{x}_i - \langle \boldsymbol{x}_i, \boldsymbol{u}\rangle \boldsymbol{u} = \left(\mathbf{I} - \boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{x}_i$$

The variance-covariance matrix of the residual vectors is

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{r}_i\boldsymbol{r}_i^T &= \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{I} - \boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{x}_i\boldsymbol{x}_i^T\left(\mathbf{I} - \boldsymbol{u}\boldsymbol{u}^T\right)\\ &= \left(\mathbf{I} - \boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{\Sigma}\left(\mathbf{I} - \boldsymbol{u}\boldsymbol{u}^T\right)\\ &= \boldsymbol{\Sigma} - \lambda\boldsymbol{u}\boldsymbol{u}^T \ (\boldsymbol{\Sigma}\boldsymbol{u} = \lambda\boldsymbol{u},\ \boldsymbol{u}^T\boldsymbol{u} = 1)\end{aligned}$$

This matrix has some interesting properties. First, $\boldsymbol{u}$ is its eigenvector with eigenvalue 0:

$$\left(\boldsymbol{\Sigma} - \lambda\boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{u} = \boldsymbol{\Sigma}\boldsymbol{u} - \lambda\boldsymbol{u} = 0$$

Recall the eigen-decomposition of $\boldsymbol{\Sigma}$

$$\boldsymbol{\Sigma} = \sum_{i=1}^{m}\lambda_i\boldsymbol{u}_i\boldsymbol{u}_i^T \tag{2.6}$$

where $\lambda_1 \geq \cdots \geq \lambda_m \geq 0$ are the eigenvalues of Σ (the non-negativity is from that Σ is positive semi-definite (p.s.d.)) and $\boldsymbol{u}_i$'s are corresponding eigenvectors. Also we have $\lambda = \lambda_1$ and $\boldsymbol{u} = \boldsymbol{u}_1$ by the discussion so far. Therefore, we obtain

$$\boldsymbol{\Sigma} - \lambda \boldsymbol{u}\boldsymbol{u}^T = \sum_{i=1}^{m} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T - \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T = \sum_{i=2}^{m} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T$$

which suggests that the eigenvectors of $\boldsymbol{\Sigma} - \lambda \boldsymbol{u}\boldsymbol{u}^T$ are still the eigenvectors of $\boldsymbol{\Sigma}$, so are the eigenvalues expect that the eigenvalue of $\boldsymbol{u}_1$ changes to 0. This means that the principal eigenvector of $\boldsymbol{\Sigma} - \lambda \boldsymbol{u}\boldsymbol{u}^T$ is the second principal eigenvector of $\boldsymbol{\Sigma}$. Moreover, the second principal eigenvector of $\boldsymbol{\Sigma}$ is the direction that has the smallest reconstruction error as well as the largest variance over the residual data.

Repeating the above procedure, i.e. finding the principal eigenvector of $\boldsymbol{\Sigma} - \lambda \boldsymbol{u}\boldsymbol{u}^T$, which is the 2nd principal eigenvector of $\boldsymbol{\Sigma}$, we can iteratively identity the d principal eigenvectors of $\boldsymbol{\Sigma}$. Note that the eigenvectors are guaranteed to be pairwise orthogonal.

Let us take a matrix view to complement the iterative one. The eigen-decomposition (2.6) can also be written in a matrix form (also known as diagonalization):

$$\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T,\ \boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_m),\ \lambda_1 \geq \cdots \geq \lambda_m \geq 0$$

where $\mathbf{U}$ is an orthogonal matrix (unit length, orthogonal columns) with its columns are eigenvectors of $\boldsymbol{\Sigma}$ (form an eigenvector basis):

$$\mathbf{U} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m),\ \mathbf{U}^T \boldsymbol{u}_i = \boldsymbol{e}_i,\ \boldsymbol{\Sigma}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i$$

The following are 2 useful results from linear algebra

Theorem 2.2 *(Spectral Theorem): Matrix* $\mathbf{A}$ *is diagonalizable by an orthogonal matrix if and only if it is symmetric.*

Theorem 2.3 *Distinct eigenvalues of symmetric matrices have orthogonal eigenvectors.*

Proof

$$\lambda_j \boldsymbol{u}_i^T \boldsymbol{u}_j = \boldsymbol{u}_i^T \mathbf{A} \boldsymbol{u}_j = \boldsymbol{u}_j^T \mathbf{A} \boldsymbol{u}_i = \lambda_i \boldsymbol{u}_j^T \boldsymbol{u}_i$$

Since $\lambda_i \neq \lambda_j$, we have $\boldsymbol{u}_i^T \boldsymbol{u}_j = 0$. □

Note that $\boldsymbol{\Sigma} = \frac{1}{n}\mathbf{X}\mathbf{X}^{\mathbf{T}}$ is symmetric, with the above 2 theorems, the optimal reduction to d dimension via PCA is to compute the first d coordinates of the data in the basis formed by eigenvectors of $\boldsymbol{\Sigma}$:

$$\mathbf{Z} = \tilde{\mathbf{U}}^T \mathbf{X} \in \mathbb{R}^{d \times n},\ \tilde{\mathbf{U}} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_d) \in \mathbb{R}^{m \times d},\ d \leq m$$

and the optimal reconstruction (guaranteed by Eckart-Young (Theorem 1.3)) is to project the data to the spanning subspace of the d principal eigenvectors of $\mathbf{\Sigma}$:

$$\tilde{\mathbf{X}} = \tilde{\mathbf{U}}\mathbf{Z} = \tilde{\mathbf{U}}\tilde{\mathbf{U}}^T\mathbf{X}$$

where $\tilde{\mathbf{U}}\tilde{\mathbf{U}}^T$ is the projection matrix.

2.3 Algorithms and Interpretation

2.3.1 Power Method

Eigenvalues and eigenvectors are extremely important for analytic purposes. Sometimes we want to know exactly what the eigenvectors and the eigenvalues is, which call for development of efficient algorithm to compute them. Power iteration is a simple algorithm for finding dominant eigenvector of $\mathbf{A}$.

As is suggested by its name, power iteration is done in a iterative way. We start from a random vector $\boldsymbol{v}_0$, multiply by $\mathbf{A}$, normalize it and do it all over again

$$\boldsymbol{v}_{t+1} = \frac{\mathbf{A}\boldsymbol{v}_t}{\|\mathbf{A}\boldsymbol{v}_t\|}$$

where we made two technical assumptions:

$$\langle \boldsymbol{u}_1, \boldsymbol{v}_0 \rangle \neq 0 \text{ and } |\lambda_1| > |\lambda_j| (\forall j \geq 2) \tag{2.7}$$

Note that if we pick $\boldsymbol{v}_0$ uniformly random, the first assumption is satisfied with probability 1. The following proposition shows the convergence of Power Iteration for p.s.d. and symmetric $\mathbf{A}$ when the two assumptions are satisfied.

Proposition 2.4 *For Power Iteration algorithm we discussed above, if $\boldsymbol{v}_0$ and $\mathbf{A}$ satisfy (2.7) then it follows*

$$\lim_{t\to\infty} \boldsymbol{v}_t = \boldsymbol{u}_1$$

and we can also recover λ_1 from Rayleigh quotient:

$$\lambda_1 = \lim_{t\to\infty} \frac{\|\mathbf{A}\boldsymbol{v}_t\|}{\|\boldsymbol{v}_t\|}$$

Proof Since $\mathbf{A}$ is p.s.d. and symmetric, by Spectral Theorem (Theorem 2.2) we know it is diagonalizable, and therefore has eigenbasis $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m\}$. We can write $\boldsymbol{v}_0$ as a linear combination of $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m\}$:

$$\boldsymbol{v}_0 = \sum_{i=1}^{m} \alpha_i \boldsymbol{u}_i, \ \alpha \neq 0$$

With the eigen-decomposition of $\mathbf{A}$: $\mathbf{A} = \sum_{i=1}^m \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T$ and using that fact that $\boldsymbol{u}_i$'s are orthogonal, we have for any vector $\boldsymbol{v} = \sum_{i=1}^m w_i \boldsymbol{u}_i$

$$\mathbf{A}\boldsymbol{v} = (\sum_{i=1}^m \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T)(\sum_{i=1}^m w_i \boldsymbol{u}_i) = \sum_{i=1}^m \lambda_i w_i \boldsymbol{u}_i$$

which shows that multiplying a vector by $\mathbf{A}$ is equivalent to multiplying λ_i's to each corresponding coordinate under basis $\{\boldsymbol{u}_1, \dots, \boldsymbol{u}_m\}$. Note that even if we do normalization in each time step (for numerical reason), it will not affect the direction of $\mathbf{A}\boldsymbol{v}_t$ and therefore is equivalent to normalize just once at the end. Thus, denote the normalization factor in the last step as c_t we obtain

$$\lim_{t\to\infty} \boldsymbol{v}_t = \lim_{t\to\infty} \frac{1}{c_t} \sum_{i=1}^m \lambda_i^t \alpha_i \boldsymbol{u}_i = \lim_{t\to\infty} \frac{\lambda_1^t \alpha_1}{c_t} \left(\boldsymbol{u}_1 + \sum_{i=2}^m \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1} \right)^t \boldsymbol{u}_i \right) = \boldsymbol{u}_1$$

as $\lambda_j < \lambda_1$. Note that c_t is a normalization factor, thus $c_t \to 1/(\lambda_1^t \alpha_1)$. $\square$

2.3.2 Example

Digital images are saved in matrix which can be vectorized and we can then apply PCA to analyze digital images. The following is the mean vector and first four principal directions (with eigenvalues) of number three in a digital number data set. (the blue means positive and yellow means negative)

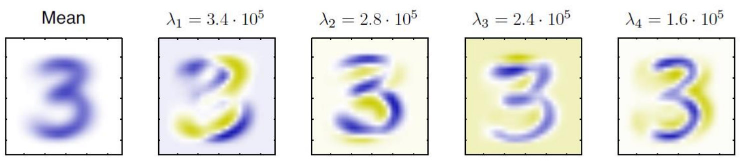

Mean
$\lambda_1 = 3.4 \cdot 10^5$
$\lambda_2 = 2.8 \cdot 10^5$
$\lambda_3 = 2.4 \cdot 10^5$
$\lambda_4 = 1.6 \cdot 10^5$

Figure 2.1: Digital number example of PCA

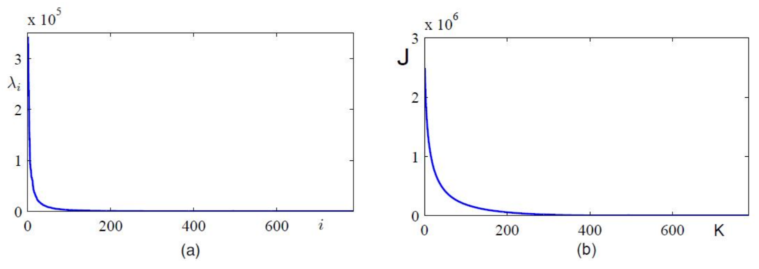


Figure 2.2: Eigenvalue spectrum example (left), and approximation error (right)

We can see that (Figure 2.1) the first 4 principal eigenvectors have very clear interpretation. The first one mainly deals with tilting, and the 4th one probably handles scaling. Also the eigenvalues show the importance of each principal vector.

It's also very helpful to look at the eigenvalue spectrum (Figure 2.2). For structured data like natural images, there is usually a fast decay in its eigenvalue spectrum. Since the reconstruction error depends on the sum of eigenvalues we ignore, there is usually a fast decay in reconstruction loss w.r.t. the number of principal components.

In the discussion so far, we always assume that the number of principal components (also for the reduced dimension number in Linear Autoencoder) is predefined and fixed. A natural question to ask is how should we choose this number, which is sometimes called *intrinsic dimensionality* . A heuristic strategy is to look at the spectrum of eigenvalues and detect the "knee" or "elbow". However, in practice, there are usually more things like the limit of computational power or storage that determine how we choose the number of reduced dimensions.

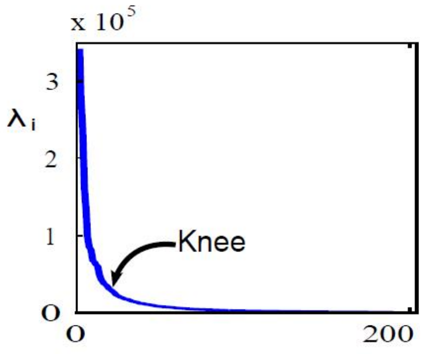


Figure 2.3: Detect "knee" to determine intrinsic dimensionality

2.3.3 Comparison with Linear Autoencoder Network

At this point, it is worth spending some time comparing the two dimension reduction method we learned:

- PCA clarifies that one should (ideally) center the data, while it's not explicit in Linear Autoencoder.
- PCA representation is unique (if no eigenvalue multiplicities) and as such (in principle) interpretable.
- Linear autoencoder without weight sharing is highly non-interpretable (lack of identifiability).
- Linear autoencoder with weight sharing: $\mathbf{C} = \mathbf{D^T}$ identifies the same subspace of k principal eigenvectors (see discussion after Remark 1.6),

but axes are non-identifiable. To modify an autoencoder to identify the principal axes, techniques like adding regularization to enforce ordering in the axis can be applied.

- General lesson for neural networks based method: caution with naively interpreting learned (neural) representations.

For the algorithm used to find the solution:

- Compute one principal component at a time via power iterations: good for small k, conceptually easy and robust.
- Train a linear auto encoder via backpropagation: it's easily extensible and we use stochastic optimization (stochastic gradient descent), which is good for very large data set and has good convergence properties.
- Compute PCA from SVD: good for mid-sized problems, can leverage wealth of numerical techniques for SVD (e.g. QR decomposition)

2.3.4 PCA via SVD

The power iteration algorithm and eigen-decompostion, which has time complexity $O(m^3)$ for $\mathbf{\Sigma} \in \mathbb{R}^{m\times m}$, suffer from large m. A way to solve the problem for the case that m is considerably large compared with n is to use SVD, which only takes $O(mn^2)$ for $\mathbf{X} \in \mathbb{R}^{m\times n}$.

To compute the eigen-decomposition of $\mathbf{AA}^T$ via SVD:

$$\mathbf{AA}^T = (\mathbf{UDV})(\mathbf{VD}^T\mathbf{U}^T) = \mathbf{UDI_nD}^T\mathbf{U}^T = \mathbf{U\Lambda U}^T$$

where $\mathbf{\Lambda} = \mathbf{DD}^T = \text{diag}(\lambda_1, \dots, \lambda_m) \in \mathbb{R}^{m\times m}$ and

$$\lambda_i = \begin{cases} \sigma_i^2 & \text{for } 1 \le i \le \min\{n, m\} \\ 0 & \text{for } n < i \le m \end{cases}$$

Similarly we have $\mathbf{A}^T\mathbf{A} = \mathbf{V\Lambda' V}^T$, where

$$\mathbf{\Lambda}' = \mathbf{D}^T\mathbf{D} = \text{diag}(\lambda'_1, \dots, \lambda'_n) \in \mathbb{R}^{n\times n},\ \lambda'_i = \begin{cases} \lambda_i^2 & \text{for } 1 \le i \le \min\{n, m\} \\ 0 & \text{for } m < i \le n \end{cases}$$

We see that $\mathbf{U}$, i.e. the left singular vectors of A, is the eigenvectors of $\mathbf{AA}^T$, and $\mathbf{V}$, i.e. the right singular vectors of A, is the eigenvectors of $\mathbf{A}^T\mathbf{A}$. $\mathbf{\Lambda}$ and $\mathbf{\Lambda}$ are identical up to zero padding. This shows that we can used the SVD of $\mathbf{X}$ to compute the eigen-decomposition of $\mathbf{\Sigma} = \frac{1}{n}\mathbf{XX}^T$. But when $n \gg m$, SVD is not so competitive. Also note that, eigen-decomposition of $\mathbf{\Sigma}$ suffers from very large and small σ_i's in terms of numerical stability since it involves squares compared with using the SVD.

3

Matrix Approximation and Reconstruction

In chapter 1, we discussed about low-rank matrix approximation, while in this chapter we will introduce a slightly different topic, Matrix Completion, which is the task of filling in the missing entries of a partially observed matrix. The matrix completion problem is in general NP-hard, but we will show in this chapter that under the low-rank assumption there are algorithms that can do exact reconstruction with high probability. We will start by introducing the famous Netflix problem and collaborative filtering, of which matrix completion is at the heart.

3.1 Collaborative Filtering

Collaborative filtering is a classical example of matrix completion and is a technique that is frequently used in recommender systems. Given the entire data set, a recommender system needs to analyze patterns of interest in terms (products, movies, ...) and provide personalized recommendations for users. With the motivation that every user has a relative stable preference and similar users have similar preferences, collaborative filtering exploits collective data from many users and builds models that can generalize across users and , possibly also, across items.

Consider the Netflix Data, which is a matrix with its rows correspond to users and columns refer to different movies. Each entry of this matrix is a 1-5 star rating of a movie given by a user, while, of course, there are a lot of missing values. Also, the data is imbalanced in a way that there are users gave a lot of ratings while some did not, so as the movies. The task is to predict the missing values. In another word, to complete the matrix.

Let the size of the rating matrix to be $m \times n$. One way to complete the matrix is to build a statistical model with $k \ll m \times n$ parameters that introduces coupling between entries and then infer the missing entries from the observed ones. This can be formalized in a low-rank decomposition manner: we want to find the best approximation with low rank $r \ll \min\{m, n\}$. In this case, number of the parameters k, i.e. the number of entries in decomposition, satisfies $k \leq r \cdot (m + n) \ll m \times n$.

3.2 Matrix Approximation via SVD

We first briefly review some of the concepts from linear algebra, which are helpful and necessary for understanding the matrix approximation method via SVD. We start by revisiting 2 frequently used matrix norms.

Definition 3.1 *(Frobenius norm) For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, its Frobenius is given by*

$$\|\mathbf{A}\|_F := \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} = \|vec(\mathbf{A})\|_2 = \sqrt{trace(\mathbf{A}^T\mathbf{A})}$$

Frobenius norm only depends on the singular values of $\mathbf{A}$:

$$\|\mathbf{A}\|_F^2 = \sum_{i=1}^{k} \sigma_i^2, \ k = \min\{m, n\}$$

which follows from the cyclic property of trace: $trace(\mathbf{ABC}) = trace(\mathbf{BCA})$:

$$trace(\mathbf{A}^T\mathbf{A}) = trace(\mathbf{V}\mathbf{D}^T\mathbf{U}^T\mathbf{U}\mathbf{D}\mathbf{V}^T) = trace(\mathbf{V}^T\mathbf{V}\mathbf{D}^T\mathbf{D}) = \sum_{i=1}^{k} \sigma_i^2$$

Definition 3.2 *(Induced p-norm) For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, its induced p-norm is given by*

$$\|\mathbf{A}\|_p := \sup\{\|\mathbf{A}\boldsymbol{x}\|_p : \ \|\boldsymbol{x}\|_p = 1\}, \ \|\boldsymbol{x}\|_p := \left(\sum_i |x_i|^p\right)^{1/p}$$

where for $p = 2$ it's also called *spectral norm*, which is equal to the largest singular value

Definition 3.3 *(Spectral norm) For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, its spectral norm is given by*

$$\|\mathbf{A}\|_2 := \sup\{\|\mathbf{A}\boldsymbol{x}\|_2 : \ \|\boldsymbol{x}\|_2 = 1\} = \sigma_1(\mathbf{A})$$

The statement that the spectral norm is equal to the largest singular value follows from

$$\begin{aligned}\|\mathbf{A}\boldsymbol{x}\|_2^2 &= \boldsymbol{x}^T\mathbf{A}^T\mathbf{A}\boldsymbol{x} \\ &= \boldsymbol{x}^T\mathbf{V}\mathbf{D}^T\mathbf{U}^T\mathbf{U}\mathbf{D}\mathbf{V}^T\boldsymbol{x} \\ &= \boldsymbol{v}^T\mathbf{D}^2\boldsymbol{v}\ (\boldsymbol{v} = \mathbf{V}\boldsymbol{x}^T,\ \|\boldsymbol{v}\|_2 = 1) \\ &= \sum_{i=1}^{k}\sigma_i^2 v_i^2 \le \sigma_1^2\sum_{i=1}^{k} v_i^2 = \sigma_1^2\end{aligned}$$

where the inequality is met with equality when $\boldsymbol{v} = (1, 0, \ldots, 0)$.

Finally, recall that Eckart-Young Theorem (Theorem 1.3) give the optimal low rank approximation in both Frobenius norm and spectral norm (see the proof of Theorem 1.3). Specifically, given $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, define for $k \le rank(\mathbf{A})$:

$$\mathbf{A}_k := \sum_{i=1}^{k}\sigma_i\boldsymbol{u}_i\boldsymbol{v}_i^T,\ rank(\mathbf{A}_k) = k$$

then $\mathbf{A}_k$ is the best approximation in terms of Frobenius norm and spectral norm:

$$\begin{aligned}\min_{rank(\mathbf{B})=k}\|\mathbf{A}-\mathbf{B}\|_F^2 &= \|\mathbf{A}-\mathbf{A}_k\|_F^2 = \sum_{r=k+1}^{rank(\mathbf{A})}\sigma_r^2 \\ \min_{rank(\mathbf{B})=k}\|\mathbf{A}-\mathbf{B}\|_2 &= \|\mathbf{A}-\mathbf{A}_k\|_2 = \sigma_{k+1}\end{aligned}$$

3.3 SVD for Collaborative Filtering

Assume that we have a fully observed rating matrix $\mathbf{A}$, then the SVD of $\mathbf{A}$ has a clear interpretation:

- The truncated number of dimension k: number of latent factors.
- $\mathbf{U}$: user-to-factor association matrix.
- $\mathbf{V}$: items-to-factor association matrix.
- $\mathbf{D}$: level of strength (importance) of each factor.

In the slide of matrix reconstruction, there is a very nice example to illustrate this interpretation of SVD.

3.4 Algorithms for matrix completion

3.4.1 The hardness of matrix completion

As we have mentioned at the beginning of the last section, the SVD solution of matrix reconstruction, suggested by Eckart-Young Theorem, requires fully observed rating matrices which in most of the times is not that case. The existence of unobserved entries turns the optimization problem from

$$\min_{rank(\mathbf{B})=k}\left[\sum_{i,j}(a_{ij}-bij)^2\right]=\min_{rank(\mathbf{B})=k}\|\mathbf{A}-\mathbf{B}\|_F^2 \tag{3.1}$$

to

$$\min_{rank(\mathbf{B})=k}\left[\sum_{(i,j)\in\mathcal{I}}(a_{ij}-bij)^2\right],\ \mathcal{I}=\{(i,j):\ \text{observed}\} \tag{3.2}$$

where in both case we usually preprocess the data to have

$$a_{ij}\leftarrow a_{ij}-\frac{1}{|\mathcal{I}|}\sum_{\mathcal{I}}a_{ij}$$

A natural attempt to remedy the problem is by replacing the unobserved entries by something like mean or zero, which is known as *data imputation.* However, this attempt will fail miserably when there are a large number of missing values.

Note that, even if Eckart-Young Theorem gives the optimal solution of (3.1) miraculously, surprisingly the optimizing problem itself is not convex, since the rank constraint is not convex. It's not difficult to see this by noticing that the convex combination of rank 1 matrices may not give a rank 1 matrix:

$$\frac{1}{2}\begin{bmatrix}1&0\\0&0\end{bmatrix}+\frac{1}{2}\begin{bmatrix}0&0\\0&1\end{bmatrix}=\frac{1}{2}\begin{bmatrix}1&0\\0&1\end{bmatrix}$$

The hardness of matrix completion can be seen from replacing the Frobenius norm in (3.1) to weighted Frobenius norm defined as follow

Definition 3.4 *(Weighted Frobenius norm) the weighted Frobenius norm with regard to matrix* $\mathbf{G}\geq\mathbf{0}$ *is defined as*

$$\|\mathbf{X}\|_{\mathbf{G}}:=\sqrt{\sum_{i,j}g_{ij}x_{ij}^2}$$

Note that the observation indicator matrix $\mathcal{I}$ is a special case of $\mathbf{G}$ for $g_{ij}\in\{0,1\}$ (Boolean).

It's known that in general, solving the following low-rank approximation problem is intrinsically hard

$$\min_{rank(\mathbf{B})\leq k}\|\mathbf{A}-\mathbf{B}\|_{\mathbf{G}}^2$$

This is shown to be NP-hard (Gillis & Glineur, 2011) even for $k=1$. The hardness also holds for approximations with prescribed accuracy and for binary $\mathbf{G}$.

So far we have seen that, matrix completion is a non-convex problem, which contributes a lot to its hardness. In general, the non-convexity of a problem comes from two aspects.

The first one is the non-convex domain, as in the matrix completion with rank constraint (3.1), where we minimize a convex objective over domain $\mathcal{Q}_k := \{\mathbf{B} : rank(\mathbf{B}) = k\}$. The way to tackle it is really at the heart of **convex relaxation** that we are going to introduce, where we relax the non-convex constraint to have a convex domain.

The other one is the non-convex objective. In our matrix completion task, to remedy the non-convex rank constraint, we force this it by re-parameterizing $\mathbf{B} = \mathbf{U}\mathbf{V}$, $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{V} \in \mathbb{R}^{k\times n}$ since by definition $rank(\mathbf{B}) \leq k$. However, there is no free lunch, by doing reparametrization, the objective becomes non-convex. Consider $f(u, v) = (a - uv)^2$, which is a simplified case for matrix problem. For $a \neq 0$, there exists $u_1v_1 = u_2v_2 = a \wedge u_1 \neq u_2$ that follows

$$f(u_1, v_1) = f(u_2, v_2) = 0 \wedge f(\frac{u_1 + u_2}{2}, \frac{v_1 + v_2}{2}) > 0.$$

Despite the non-convexity of the new objective, it has some nice properties that lead to the **alternating least squares** method.

Remark 3.5 *An interesting thing to mention is that the decomposition* $\mathbf{B} = \mathbf{U}\mathbf{V}$ *not only removes the non-convex rank constraint but also maintains a analogous interpretation as SVD.* $\mathbf{U}$ *captures user-to-factor association while* $\mathbf{V}$ *describes item-to-factor collaboration.*

3.4.2 Alternating Least Squares

Using the reparametrization, we make the domain convex, but at the same time, the objective function $f(\mathbf{U}, \mathbf{V})$ is not jointly convex in $\mathbf{U}$ and $\mathbf{V}$.

$$f(\mathbf{U}, \mathbf{V}) = \frac{1}{|\mathcal{I}|} \sum_{(i,j)\in\mathcal{I}} (a_{ij} - \langle \boldsymbol{u}_i, \boldsymbol{v}_j \rangle)^2$$

where $\boldsymbol{u}_i$'s are rows of $\mathbf{U}$ and $\boldsymbol{v}_i$'s are columns of $\mathbf{V}$. However, for fixed $\mathbf{U}$, f is convex in $\mathbf{V}$. For fixed $\mathbf{V}$ in turn, f is convex in $\mathbf{U}$. This observation suggests an alternating minimization over $\mathbf{U}$ and $\mathbf{V}$:

$$\begin{aligned}
\mathbf{U} &\leftarrow \arg\min_{\mathbf{U}} f(\mathbf{U}, \mathbf{V}) \\
\mathbf{V} &\leftarrow \arg\min_{\mathbf{V}} f(\mathbf{U}, \mathbf{V}), \quad \text{repeat until convergence}
\end{aligned}$$

Since f is never increased in this alternating minimization progress, and by definition (sum of squares) f is lower bounded by 0, by monotone convergence theorem f will converge as the number of iteration goes to infinity. However, the convergence of f does not necessarily mean that $\mathbf{U}$ and $\mathbf{V}$ will also converge (say if they are not uniquely identifiable), and optimization may stuck at local optima.

This alternating minimization for low-rank matrix factorization is called *alternating least squares* (ALS). Despite the drawbacks mentioned above, alternating least squares is very efficiently solvable. The fact that in each iteration we are dealing with a convex problem takes part of the reason. More importantly, ALS decompose f into subproblems (convex), each of which can be solved independently.

Consider $\mathbf{U}$ is fixed, then f is decomposed into subproblems for columns of $\mathbf{V}$:

$$f(\mathbf{U}, \mathbf{V}) = \sum_j \left[\sum_{i:(i,j)\in\mathcal{I}} (a_{ij} - \langle \boldsymbol{u}_i, \boldsymbol{v}_j \rangle)^2 \right] = \sum_j f_j(\mathbf{U}, \boldsymbol{v}_j)$$

where $f_j(\mathbf{U}, \boldsymbol{v}_j)$'s are independently solvable least square problems for columns $\boldsymbol{v}_j$'s of $\mathbf{V}$. By symmetry, this also holds for fixing $\mathbf{V}$ and optimizing $\mathbf{U}$. Note that columns of $\mathbf{V}$ ($\boldsymbol{v}_j \in \mathbb{R}^k$) are representations for each item, while rows of $\mathbf{U}$ ($\boldsymbol{u}_i \in \mathbb{R}^k$) are representations for each user.

Typically, we favor solutions $(\mathbf{U}, \mathbf{V})$ with small Frobenius norm. So we add (squared) Frobenius norm regularizer to regularize matrix factor $\mathbf{U}, \mathbf{V}$:

$$\Omega(\mathbf{U}, \mathbf{V}) = \|\mathbf{U}\|_F^2 + \|\mathbf{V}\|_F^2$$

then we optimize a regularized problem

$$\min_{\mathbf{U},\mathbf{V}} \left(f(\mathbf{U}, \mathbf{V}) + \mu\Omega(\mathbf{U}, \mathbf{V}) \right), \text{ for some } \mu > 0$$

Since the (squared) Frobenius norm also decomposes, adding regularization does not change the separability structure of the problem.

In a word, what ALS for collaborative filtering is doing is to alternatively do the following:

- Given low-dimensional representations for items ($\mathbf{V}$ fixed), compute for each user independently the best representation, i.e. optimize rows of $\mathbf{U}$.

- Given low-dimensional representations for users ($\mathbf{U}$ fixed), compute for each item independently the best representation, i.e. optimize columns of $\mathbf{V}$.

All optimization problems are least-square problems of small dimension ($\boldsymbol{u}_i, \boldsymbol{v}_j \in \mathbb{R}^k$).

3.4.3 Convex Relaxation

In this section, we exploit the other direction for solving the original non-convex optimization problem. We start by introducing another useful matrix norm, the nuclear norm.

Definition 3.6 *(Nuclear norm) The nuclear norm of a matrix* $\mathbf{A}$ *is defined as the sum of its singular values*

$$\|\mathbf{A}\|_* = \sum_i \sigma_i, \quad \sigma_i : \textit{ singular values of } \mathbf{A}$$

Recall the definition of Frobenius norm $\|\mathbf{A}\|_F = \sqrt{\sum_i \sigma_i^2}$, then if we define $\boldsymbol{\sigma}(\mathbf{A}) = (\sigma_1, \ldots, \sigma_n)$, we have

$$\|\mathbf{A}\|_F = \|\boldsymbol{\sigma}(\mathbf{A})\|_2, \ \|\mathbf{A}\|_* = \|\boldsymbol{\sigma}(\mathbf{A})\|_1$$

For a diagonal matrix $\mathbf{D}$, $\|\mathbf{D}\|_* = \mathrm{Tr}(\mathbf{D})$. For matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ of rank r, the following two inequalities describe the relation between the matrix norms we have met so far:

$$\|\mathbf{A}\|_2 \le \|\mathbf{A}\|_F \le \sqrt{r}\|\mathbf{A}\|_2$$
$$\|\mathbf{A}\|_F \le \|\mathbf{A}\|_* \le \sqrt{r}\|\mathbf{A}\|_F$$

With the nuclear norm, what people usually look at in matrix reconstruction are the following 2 types of problems. The first one is exact reconstruction (given observed entries indicated by Boolean matrix $\mathbf{G}$):

$$\min_{\mathbf{B}} \|\mathbf{B}\|_* \quad \text{subject to } \|\mathbf{A} - \mathbf{B}\|_{\mathbf{G}} = 0$$

Think from another direction, we get the approximate reconstruction problem:

$$\min_{\mathbf{B}} \|\mathbf{A} - \mathbf{B}\|_{\mathbf{G}}^2 \quad \text{s.t. } \|\mathbf{B}\|_* \le r$$

the Lagrangian formulation of which is

$$\min_{\mathbf{B}} \left[\frac{1}{2\tau} \|\mathbf{A} - \mathbf{B}\|_{\mathbf{G}}^2 + \|\mathbf{B}\|_* \right]$$

The reason to introduce the nuclear norm is that under certain conditions, the constraint on the nuclear norm is the convex relaxation of the constraint on the rank. Specifically, it's not difficult to see that

$$rank(\mathbf{B}) \ge \|\mathbf{B}\|_*, \quad \text{for } \|\mathbf{B}\|_2 \le 1$$

where $\|\cdot\|_2$ is the spectral norm. This means that, for $\|\mathbf{B}\|_2 \le 1$, the nuclear norm is a lower bound of the rank (in face it's the tightest convex lower bound (Fazel 2002)). To make use of this property of the nuclear norm, we can properly rescale the original data (rating matrix $\mathbf{A}$) to make its approximation $\mathbf{B}$ very likely to satisfy $\|\mathbf{B}\|_2 \leq 1$.

Now assuming that $\|\mathbf{B}\|_2 \leq 1$ is always satisfied, we get the follow convex relaxation to the original matrix completion problem (3.2):

$$\min_{\mathbf{B}\in\mathcal{P}_k} \|\mathbf{A}-\mathbf{B}\|_G^2,\ \mathcal{P}_k := \{\mathbf{B} : \|\mathbf{B}\|_* \leq k\}$$

where

$$\mathcal{P}_k \supseteq \mathcal{Q}_k := \{\mathbf{B} : rank(\mathbf{B}) \leq k\}$$

By relaxing the constraint to a larger but convex one, we keep all possible solutions of the original problem and make the problem much easier to solve. However, note that we also add some additional possible solutions ($rank(\mathbf{A}) \geq k$), in another word, we are optimizing over a larger set. For a convex relaxation problem, the final step is usually rounding or projecting the solution back to the original domain, in our case, the domain of rank constraint. However, for this matrix completion problem, there is actually no such a prior knowledge about constraints on the rank of our approximation, so the solution given by the relaxed problem is also acceptable.

SVD Shrinkage Iterations is an algorithm for solving matrix completion problems involving the nuclear norm. We start by introducing a fundamental result (*SVD Thresholding*), where we try to solve the following Lagrangian involving the nuclear norm (here $\mathbf{A}$ is an arbitrary matrix)

$$\mathbf{B}^* = \text{shrink}_\tau(\mathbf{A}) := \arg\min_{\mathbf{B}}\{\frac{1}{2}\|\mathbf{A}-\mathbf{B}\|_F^2 + \tau\|\mathbf{B}\|_*\} \tag{3.3}$$

then with $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, $\mathbf{D} = \text{diag}(\sigma_i)$, it holds that

$$\mathbf{B}^* = \mathbf{U}\mathbf{D}_\tau\mathbf{V}^T,\ \mathbf{D}_\tau = \text{diag}(\max\{0, \sigma_i - \tau\})$$

Note that all singular values are reduced by at most τ. Recall that we have assumption that $\|\mathbf{B}\|_2 \leq 1$, so in our setting we should have $\tau \leq 1$.

Recall that solving a Lagrangian is almost equivalent to solving a constrained optimization problem, and in (3.3) it's a constraint on the nuclear norm. There is a strong connection between requiring constrained rank and constrained nuclear norm. According to Eckart-Young Theorem (Theorem 1.3), the optimal low-rank approximation is achieved by "throwing away" the small singular values, while in SVD thresholding, we reduce all singular values by τ. For structured data (matrix $\mathbf{A}$ that we believe to be low-rank), it usually shares a similar spectrum of singular values as in Figure 2.2, which means by choosing a small τ SVD thresholding can zero out most of the small σ_i while keeping the principal σ_i's almost unchanged. From this we can see that, the rank constraint and nuclear norm constraint often yield the same solution, and they can both been computed by SVD.

With SVD thresholding, the iteration algorithm is then simply a combination of SVD thresholding and an extra projection. Define the projection operator with regard to index set $\mathcal{I}$ as

$$\Pi(\mathbf{X}) = \begin{cases} x_{ij} & (i,j) \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases}$$

The iteration algorithm, initialized with $\mathbf{B}_0 = 0$ is then given by

$$\mathbf{B}_{t+1} = \mathbf{B}_t + \eta_t \Pi(\mathbf{A} - \text{shrink}_\tau(\mathbf{B}_t)) \tag{3.4}$$

where $\eta_t > 0$ is the learning rate or step size for step t (learning rate sequence). Here in each step, $\text{shrink}_\tau(\mathbf{B}_t)$ acts as an approximation to $\mathbf{A}$. Think about the second term as a difference between $\mathbf{A}$ and the current approximation, then by iteratively apply (3.4), we get a better and better approximation to $\mathbf{A}$.

Note that since the second term only supports on $\mathcal{I}$, the $\mathbf{B}_t$ will always be a sequence of sparse matrices for $\mathbf{B}_0 = 0$, and therefore is efficient (in terms of, say, storage). This is important since the size of $\mathbf{B}$ can be large. In the ALS, this problem is handled by reparametrizing $\mathbf{B}$ into multiplication of two small matrices, and here we make $\mathbf{B}_t$'s sparse. It can be shown that, with appropriate choice of step sizes, $\mathbf{B}^* = \lim_{t\to\infty} \text{shrink}_\tau(\mathbf{B}_t)$ satisfies

$$\mathbf{B}^* = \arg\min_{\mathbf{B}} \{\|\mathbf{B}\|_* + \frac{1}{2\tau}\|\mathbf{B}\|_F^2\}, \text{ s.t. } \Pi(\mathbf{A} - \mathbf{B}) = \mathbf{0} \tag{3.5}$$

which means that $\mathbf{B}^*$ will agree with $\mathbf{A}$ on all observed entries.

From (3.5) we can see that for large enough τ (we know that at least in the setting of our problem, the τ cannot be arbitrarily large since we have $\|\mathbf{B}\|_2 \leq 1$) one finds a minimal nuclear-norm approximation to $\mathbf{A}$ that agrees on all observed entries. Note that the objective function is kind of different from the original one, where we try to minimize $\|\mathbf{A} - \mathbf{B}\|_{\mathcal{I}}^2$, while here the objective is a regularizer. However, (3.5) says that $\mathbf{B}^*$ will agree with $\mathbf{A}$ on all observed entries, and for the unobserved entries of $\mathbf{B}^*$ they are optimal in terms of a hybrid norm objective, which is exactly what we are looking for. Besides, by modifying Π this analysis of SVD shrinkage iterations can be extended to an approximating setting, where $\|\mathbf{A} - \mathbf{B}\|_{\mathcal{I}}$ is not zero but minimized under certain conditions.

Despite that (3.5) gives a solution that can do exact recovery, a natural question raises is whether we can get any generalization guarantees on $\mathbf{B}$ at unobserved positions of $\mathbf{A}^*$, assuming that there is actually a latent ground truth matrix $\mathbf{A}^*$ s.t. $\Pi(\mathbf{A}^*) = \mathbf{A}$. Surprisingly, but not trivially, the answer is yes for

$$\mathbf{B}^* = \arg\min_{\mathbf{B}} \{\|\mathbf{B}\|_*\}, \text{ s.t. } \Pi(\mathbf{A} - \mathbf{B}) = \mathbf{0} \tag{3.6}$$

This is justified by the follow theorem:

Theorem 3.7 *Exact reconstruction of rank k matrix $\mathbf{A}^*$ w.h.p., if it is strongly incoherent (parameter μ, spread of singular values), if*

$$|\mathcal{I}| \geq C\mu^4 k^2 n(\log n)^2 \in \tilde{O}(n), \text{ typically } \mu = O(\sqrt{n})$$

This theorem shows that with $|\mathcal{I}| \geq \tilde{O}(n)$ ($\ll n^2$ for large n), under certain conditions, $\mathbf{B}^*$ defined in (3.6) can do exact reconstruction of $\mathbf{A}^*$ (on all entries) w.h.p., which to some degree explains why the nuclear norm minimization works well in practice.

4

Non-Negative Matrix Factorization

This chapter discusses about a variant of matrix factorization, where for some reasons we want the entries of our model to be non-negative, and we will try to understand how does this assumption change our problem. We will start from a classical motivation in Natural Language Processing (NLP) field, topic models, and then introduce two important approaches, between which is brief review of the Expectation Maximization (EM) algorithm. Finally, we will discuss the non-negative matrix factorization (NMF) problem in a more general view.

4.1 Motivation

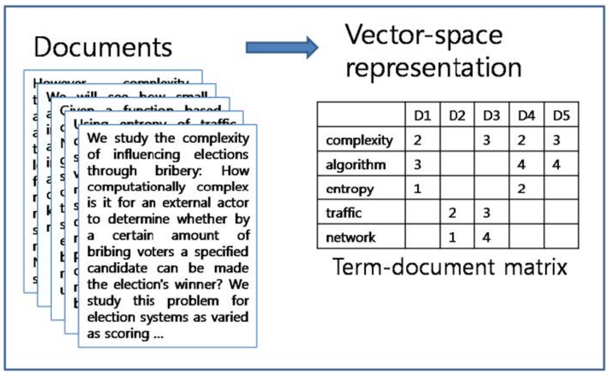
Given the corpus of text documents, like web pages, a usual challenge in NLP is to find a low-dimensional representation in semantic space of topics or concepts for the documents. This is also known as *topic models*, where we want to find these topics in the data set and represent documents with regard to these topics.

From a probabilistic view, we can address this problem by building a predictive model for each document and predicting additional words for this document. In another word, given some observed words we try to predict the probabilities for additional words. This approach comes from the intuition that model with high prediction power should extract interesting structures of the data. Concretely, we can try to maximize the log-likelihood of predicting words in the document, which gives exactly the idea of *probabilistic Latent Semantic Analysis (pLSA)*. *Latent Dirichlet Allocation (LDA)* improves pLSA by exploiting the ability to generate new data from a Bayesian point of view. pLSA is closely related to NMF and will be discussed again in the last section of this chapter.

Before pLSA, it's useful to introduce some concepts and commonly-used pre-processing. The *vocabulary* is the set of all "meaningful" words in a language and is extracted from corpus documents via *tokenization* (also known as word segmentation), which segments the text or document into a sequence of strings (words) based on certain rules. For human languages, the vocabularies are usually with large cardinality (say 1-100 million). *Term filtering* and *term normalization* are also popularly applied, where the first one excludes stop words like "the" or "at" and infrequent words, misspellings, etc; the second one reduces words to stems (*stemming*) like reducing "argue" and "argues" to the stem "arg".

As far as what the topicality is concerned of what the document is about, we can make an assumption to use *Bag-of-Word (BoW) representations* for documents, which ignore the order of word in sentences and simplify the problem to a matrix problem. The BoW representation reduces the whole data set to co-occurrence counts (Figure 4.1), where in this case the context of each word is the entire document, and each document is then a M-dimensional vector of counts. Recall that M is usually very large, then the document vectors are very sparse. Note that the sparsity here is different from the one in the rating task, where the sparsity comes from the lack of observation. Here the zeros simply mean zero counts and are actually observed values.

It might seems to be a reasonable approach to make prediction based on this co-occurrence count matrix, but since it's very sparse it will give zero probabilities to unseen words in a document. For a predictive model, we want to do better.



We study the complexity of influencing elections through bribery: How computationally complex is it for an external actor to determine whether by a certain amount of bribing voters a specified candidate can be made the election's winner? We study this problem for election systems as varied as scoring ...

	D1	D2	D3	D4	D5
complexity	2		3	2	3
algorithm	3			4	4
entropy	1			2	
traffic		2	3		
network		1	4		

Figure 4.1: Bag-of-Word embedding

4.2 Probabilistic Latent Semantic Analysis

pLSA identifies topics with characteristic distributions over words, which are the model parameters, and models each document as a **mixture of topics**. Note that the mixing weights are not the uncertainties about certain "correct" topic but show that the document belongs to multiple topics with different "importance". For example, a report about soccer world cup 2022 might contains words from soccer vocabulary (e.g. "teams", "play", "match") in a sense of distribution and words from political vocabulary (e.g. "labor", "corruption", "president") in a sense of distribution. The goal is to discover topics in an unsupervised fashion.

4.2.1 Basic Model

As a predictive model, pLSA samples words in a two-stage manner:

1. Sample topic from the mixtures for each token/word.
2. Sample token/word, given sampled topic

As illustrated in Figure 4.2, different colors correspond to different topics. Given a document d, we first sample topics z_i for every position i in the document, and then sample a word w_i conditioning on the topic z_i of that position.

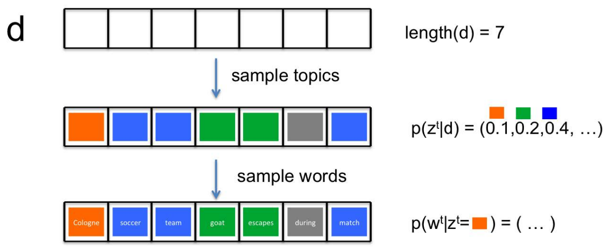


Figure 4.2: Two-stage sampling of pLSA

The model parameters are tied up with this two-stage sampling:

1. Each document has a specific mix of topics: $p(z|d)$.
2. Each topic has a specific distribution of words: $p(w|z)$.

Given the model parameters, we can compute the log-likelihood of observed data and use some criteria to evaluate how well our model performs. Specifically, we can get the occurrence (probability) of word w in context/document d (also known as context model) by marginalizing out topic z from the conditional joint distribution given d:

$$p(w|d) = \sum_{z=1}^{K} p(w,z|d) = \sum_{z=1}^{K} p(w|z,d)p(z|d) = \sum_{z=1}^{K} p(w|z)p(z|d) \tag{4.1}$$

where we identify topics with integers $z \in \{1, \ldots, K\}$ (K: pre-specified), and word w is related to a fixed "slot", i.e. fixed position in the document, and has an identical distribution for every slot. In the last equality above, we made a conditional independence assumption $w \perp d \mid z$ for that we assume topics represent regularities common to the entire collection.

Remark 4.1 *As mentioned above, we need to pre-specify the topic number K for pLSA. However, we need to be careful about the interpretability of our model on these "topics". For example, if we apply pLSA with $K = 4$ on data collected from 4 topics (say "sports", "politics", "music", and "science"), we may expect our pLSA model to find these topics, but with $K = 3$ the "meaning" of topics found by our model may be a lot more unpredictable.*

With (4.1) and an additional assumption that every word in a document is independent of each other, the log-likelihood can be computed in 2 possible views:

- Summarize data into co-occurrence counts $\mathbf{X} = x_{ij}$ (number of occurrences of w_j in document d_i)
- Alternatively: multiset $\mathcal{X}$ over index pair (i, j), e.g.

$$\mathcal{X} = \{(1,1), (1,1), (1,2), \dots\}$$

The log-likelihood, therefore, is

$$l(\mathbf{U}, \mathbf{V}) = \sum_{i,j} x_{ij} \log p(w_j|d_i) = \sum_{(i,j)\in\mathcal{X}} \log \sum_{z=1}^{K} \underbrace{p(w_j|z)}_{=:v_{zj}} \underbrace{p(z|d_i)}_{=:u_{zi}} \tag{4.2}$$

where $u_{zi} \geq 0$ and $v_{zj} \geq 0$ are model parameters. Since they are probability distributions, we further require constraints:

$$\sum_z u_{zi} = 1 \ (\forall i), \ \sum_z v_{zj} = 1 \ (\forall j)$$

A natural attempt is to optimize (4.2) via gradient descent, but the sum of probabilities inside the log can cause problem. In the following, we will address the problem from another view, namely the Expectation Maximization, which iteratively optimizes a lower bound of the original objective function. Since it's a very useful algorithm and is closely related to another topic we will meet in the sequel, the k-means algorithm, we will introduce more of EM in details in the next section, while it might be out of the scope of this course.

4.2.2 Expectation Maximization for pLSA

EM deals with the hardness of maximization of the original log-likelihood function by assuming latent variables and bring them back to play, which is known as complete data model. Denote all observed variables (i.e. words in topic model) as $\mathbf{X}$, all latent variables as $\mathbf{Z}$, and model parameters as θ. Then, $\{\mathbf{X}, \mathbf{Z}\}$ represents the complete data set, while $\{\mathbf{X}\}$ is called incomplete data set. In this case, we shall suppose that optimization of the $p(\mathbf{X}|\theta)$ is hard but easy for $p(\mathbf{X}, \mathbf{Z}|\theta)$. This will be discussed in detail in the next section.

In pLSA, we introduce missing data $Q_{zij} \in \{0, 1\}$ indicating that w_j in d_i is generated via z. Note that by construction: $\sum_z Q_{zij} = 1$. Then this naturally yields corresponding variational parameters (latent variables), which is a distribution over z:

$$q_{zij} = \mathbb{P}(Q_{zij} = 1), \ \sum_z q_{zij} = 1$$

With q_{zij}, we can lower bound $\log p(w_j|d_i)$ in (4.2) via Jensen's inequality. First re-write it into a convex combination, using that $q_{zij} \geq 0, \ \sum_z q_{zij} = 1$:

$$\log p(w_j|d_i) = \log \sum_{z=1}^{K} u_{zi} v_{zj} = \log \sum_{z=1}^{K} q_{zij} \frac{u_{zi} v_{zj}}{q_{zij}}$$

Then since $\log(\cdot)$ is a concave function, Jensen's inequality gives:

$$\log p(w_j|d_i) = \log \sum_{z=1}^{K} q_{zij} \frac{u_{zi} v_{zj}}{q_{zij}} \geq \sum_{z=1}^{K} q_{zij} [\log u_{zi} + \log v_{zj} - \log q_{zij}] \quad (4.3)$$

Optimizing over q for fixed u_{zi}, v_{zj} maximizes the lower bound (**Expectation Step**). To solve for optimal q, we can apply Lagrangian:

$$\mathcal{L}(q, \lambda) = \sum_{z=1}^{K} q_{zij} [\log u_{zi} + \log v_{zj} - \log q_{zij}] + \lambda (\sum_z q_{zij} - 1)$$

Apply the first order optimality condition, we get

$$\lambda + \sum_z (\log u_{zi} v_{zj} - \log q_{zij} - 1) \overset{set}{=} 0$$

with normalization constraint we get

$$q_{zij} = \frac{u_{zi} v_{zj}}{\sum_{z=1}^{K} u_{zi} v_{zj}} = \frac{p(w_j|z) p(z|d_i)}{\sum_{z=1}^{K} p(w_j|z) p(z|d_i)}$$

which turns out to be the posterior of Q_{zij} under current model ($\{u_{zi}, v_{zj}\}$). The interpretation of the expectation step is clear: if given u_{zi}, v_{zj} we can solve for the optimal q. But since we don't know the true model parameters in advance, we need to do it iteratively with the following **Maximization**

Step, which solves for the optimal parameters with fixed q. First apply lower bound (4.3) to the log-likelihood (4.2):

$$l(\mathbf{U}, \mathbf{V}) \geq \sum_{i,j} x_{ij} \sum_{z=1}^{K} q_{zij} [\log u_{zi} + \log v_{zj} - \log q_{zij}]$$

Then again we can apply Lagrangian to solve for optimal u_{zi} and v_{zj}. With a similar derivation, we get

$$u_{zi} = \frac{\sum_j x_{ij} q_{zij}}{\sum_j x_{ij}}, \quad v_{zj} = \frac{\sum_i x_{ij} q_{zij}}{\sum_{i,l} x_{il} q_{zil}}$$

where in the first equality we used $\sum_z q_{zij} = 1$. Note that the denominators of the two expressions differs simply because u_{zi} is a distribution over z and v_{zj} is a distribution over j. Also, we see that the numerators are simple weighted counts, and the denominators ensure proper normalization.

EM for MLE in pLSA gives a simple alternating scheme for finding a solution and is guaranteed to converge. However, EM is not guaranteed to find global optimum.

4.3 Expectation Maximization in a section

This section is not relevant to this course and is written for my personal interests. So feel free to skip it! The content mainly comes from chapter 9 of one of my favorite books, *Pattern Recognition and Machine Learning* by Christopher M. Bishop. I recommend ones who are interested in more materials to read the corresponding chapters.

4.3.1 K-means Clustering

One of the most popular clustering methods is k-means algorithm. Given $x_1, \ldots, x_n \in \mathbb{R}^d$, the k-means algorithm partitions the data points into k disjoint "groups" $S_1, \ldots, S_k$ with centers $\mu_1, \ldots, \mu_k \in \mathbb{R}$, which can be seen as prototypes of each "group". In another word, k-means defines the following optimization problem:

$$\min_{\{S_k\}, \{\mu_k\}} \sum_{k=1}^{K} \sum_{n \in S_k} \|x_n - \mu_k\|^2$$

or equivalently

$$\min_{\{r_{nk}\}, \{\mu_k\}} \sum_{k=1}^{K} \sum_{n=1}^{N} r_{nk} \|x_n - \mu_k\|^2$$

where $r_{nk} \in \{0, 1\}$ is a binary indicator variable describing which of the K cluster the data x_n is assigned to. Since directly optimizing k-means objective is NP-hard, we usually apply an iterative scheme, which is also known as Lloyd's algorithm:

- Given centers $\mu_1, \dots, \mu_K$, assign each point to the closest center:

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|x_n - \mu_j\|^2 \\ 0 & \text{otherwise.} \end{cases}$$

- Given all the assignments, find the optimal centers:

$$\mu_k = \frac{1}{|S_k|} \sum_{n \in S_k} x_n.$$

We shall see that these two stages of updating r_{nk} and updating μ_k correspond to the E steps and M steps of the EM algorithm, respectively.

4.3.2 Mixtures of Gaussians

We now turn to a formulation of Gaussian mixtures in terms of discrete latent variables, which will serve to motivate the EM algorithm. A Gaussian mixture distribution can be written in a form of linear superposition of Gaussians:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

Let us introduce a K-dimensional binary random variable $\mathbf{z}$ satisfying that $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$. Then we can define the joint distribution $p(\mathbf{x}, \mathbf{z})$ using a marginal distribution $p(\mathbf{z})$ and a conditional distribution $p(\mathbf{x}|\mathbf{z})$. $p(\mathbf{z})$ is specified in terms of the mixing coefficients π_k, such that

$$p(z_k = 1) = \pi_k \in [0, 1], \; \sum_k \pi_k = 1$$

which yields

$$p(\mathbf{z}) = \prod_k \pi_k^{z_k}.$$

On the other hand, the conditional distribution of $\mathbf{x}$ given a specific $\mathbf{z}$, note that $\mathbf{z}$ only has K different states, is given by

$$p(\mathbf{x}|z_k = 1) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

which is equivalent to

$$p(\mathbf{x}|\mathbf{z}) = \prod_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}.$$

The marginal distribution of $\mathbf{x}$ is then obtained by summing the joint distribution $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ over all possible states of $\mathbf{z}$:

$$p(\mathbf{x}) = \sum_z p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_z \prod_k (\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{z_k} = \sum_k \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

We have therefore found an equivalent formulation of the Gaussian mixture involving an explicit latent variable. This allow us to work with the joint distribution $p(\mathbf{x}, \mathbf{z})$ instead of $p(\mathbf{x})$, which will lead to significant simplifications. Another quantity that will play an important role is $p(\mathbf{z}|\mathbf{x})$. We use $\gamma(z_k)$ to denote $p(z_k = 1|\mathbf{x})$, whose value can be found using Bayes' theorem:

$$\gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) = \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_i p(z_i = 1)p(\mathbf{x}|z_i = 1)} = \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_i \pi_i \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)}.$$

$\gamma(z_k)$ has an interpretation as the *responsibility* that component k takes for "explaining" the observation $\mathbf{x}$.

Now we represent the data that we want to model with Gaussian mixtures by $\mathbf{X} \in \mathbb{R}^{N \times D}$ with the n^{th} row is given by $\mathbf{x}_n^T$, and the latent variables by $\mathbf{Z} \in \mathbb{R}^{N \times K}$ with the n^{th} row $\mathbf{z}_n^T$. Assume that the data points are independently drawn from the distribution, the log-likelihood therefore is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{4.4}$$

The difficulty of maximizing (4.4) arises from the presence of the summation over k that appears inside the logarithm. If we set the derivative to zero, we can no longer get a close form solution.

4.3.3 EM for Gaussian mixtures

Expectation-maximization is an elegant and powerful method for finding maximum likelihood solutions for models with latent variables. First we will discuss EM in the context of Gaussian mixtures, and later in a more general view.

The conditions that must be satisfied for an optimal solution is the first order optimality condition. Setting the derivatives of (4.4) with respect to $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ to zero, we obtain

$$\frac{\partial(4.4)}{\partial \boldsymbol{\mu}_k} = 0 \Rightarrow \boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \tag{4.5}$$

$$\frac{\partial(4.4)}{\partial \boldsymbol{\Sigma}_k} = 0 \Rightarrow \boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \tag{4.6}$$

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}), \quad \gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_i \pi_i \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)} \tag{4.7}$$

where we have assumed that $\boldsymbol{\Sigma}_k$ is invertible for all k, and N_k has a interpretation as the effective number of point assigned to cluster k. From the form of this solution we see that $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are weighted version of mean and variance of the data set, and the weighting factor for $\mathbf{x}_n$ is given by the posterior probability $\gamma(z_{nk})$ that component k was responsible for generating $\mathbf{x}_n$.

Finally, we maximize (4.4) w.r.t. the mixing coefficients π_k. Note that π_k needs to be non-negative and normalized. We use a Lagrange multiplier to force the normalization constraint and show that the solution satisfies the non-negative constraint automatically. Define the following Lagrangian

$$\mathcal{L} = \ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \lambda \left(\sum_k \pi_k - 1 \right).$$

Set it's derivative to zero w.r.t. π_k and use the normalization constraint we obtain

$$\pi_k = \frac{N_k}{N} \tag{4.8}$$

which has a clear interpretation as the average responsibility that component k takes for explaining the data points. It is worth emphasizing that the results (4.5), (4.6), and (4.8) are not closed form solutions for the model parameters because of the dependency of the responsibilities $\gamma(z_{nk})$ on these parameters. However, these results suggest a simple iterative scheme for finding a solution, which we will see turns out to be an instance of the EM algorithm.

We first initialize all the model parameters, and then alternate between the following two updates until some conditions are satisfied:

- **E (expectation) step**: Use the current model parameters $\{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ to evaluate the posterior probabilities/responsibilities $\gamma(z_{nk})$ via (4.7).
- **M (maximization) step**: Use the updated responsibilities $\gamma(z_{nk})$ to re-estimate model parameters $\{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ via (4.5), (4.6), and (4.8).

We shall see that in each update the parameters resulting from an E step followed by an M step is guaranteed to increase the log-likelihood function. Since the log-likelihood function is always upper-bounded by 0, EM is therefore guaranteed to converge in general since EM maximizes a lower bound for the log-likelihood in each step.

4.3.4 An Alternative View of EM

In this section, we will discuss th EM algorithm in a more general setting. The goal of the EM algorithm is to find maximum likelihood solutions for models having latent variables. We denote the set of all observed data by $\mathbf{X}$ with the n^{th} row representing $\mathbf{x}_n^T$, and similarly we denote the set of all latent variables by $\mathbf{Z}$ with a corresponding row $\mathbf{z}_n^T$. Given the set of all model parameters denoted by $\boldsymbol{\theta}$, the log-likelihood of $\mathbf{X}$ is

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \ln \left(\sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) \right).$$

The hardness of optimizing this objective arises from summation over the latent variables inside the logarithm. Now suppose that, for each observation $\mathbf{X}$, we were told the corresponding value of the latent variable $\mathbf{Z}$. We shall call $\{\mathbf{X}, \mathbf{Z}\}$ the *complete* data set, and refer to the actual observed data $\mathbf{X}$ as *incomplete*. We shall suppose that maximization of the complete-data log likelihood function $\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is straightforward.

In practice, we are not given the complete data set $\{\mathbf{X}, \mathbf{Z}\}$, and our knowledge of the values of the latent variables $\mathbf{Z}$ is given only by the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$. Because we cannot use the complete-data log-likelihood, we consider instead its expected value under the posterior distribution of the latent variable, which corresponds to the E step of the EM algorithm. In the subsequent M step, we maximize this expectation. If the current estimate for the parameters is denoted $\boldsymbol{\theta}^{\text{old}}$, then a pair of successive E and M steps gives rise to a revised estimate $\boldsymbol{\theta}^{\text{new}}$. The parameters are initialized by $\boldsymbol{\theta}_0$.

In the E step, we use the current model parameters $\boldsymbol{\theta}^{\text{old}}$ to find the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$, and use this distribution to find the expectation of the complete-data log-likelihood for some general parameter value $\boldsymbol{\theta}$:

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}).$$

In the M step, we determine the revised parameter estimate $\boldsymbol{\theta}^{\text{new}}$ by maximizing this function

$$\boldsymbol{\theta}^{\text{new}} = \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}).$$

Note that in the definition of $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, the summation is outside of the logarithm, so, by assumption, the M step will be tractable.

Gaussian Mixtures Revisited

We now consider the application of this latent variable view of EM to the Gaussian mixture model. With the same notation as before, the likelihood for the complete data set takes the form

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{n=1}^{N} \prod_{k=1}^{K} (\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{z_{nk}} \tag{4.9}$$

which yields the corresponding log-likelihood

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left(\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)$$

Using (4.9) together with the Bayes rule, we have

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \propto \prod_{n=1}^{N} \prod_{k=1}^{K} (\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{z_{nk}}$$

and it's not difficult to see that under the posterior distribution the $\{\mathbf{z}_n\}$ are independent. Using that $z_{nk} = \{0,1\}$ and $\sum_k z_{nk} = 1$, the expectation of z_{nk} under the posterior distribution is given by

$$\begin{aligned}
\mathbb{E}[z_{nk}] &= \sum_{z_{nk}} z_{nk} p(z_{nk}|\mathbf{x}_n, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \\
&= \sum_{z_{nk}} z_{nk} \frac{p(z_{nk}, \mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{z_{nj}} p(z_{nj}, \mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \\
&= \frac{\sum_{z_{nk}} z_{nk}(\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{z_{nk}}}{\sum_{z_{nj}} (\pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j))^{z_{nj}}} \\
&= \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} = \gamma(z_{nk}).
\end{aligned}$$

Note that the sum over z_{nk} only has 2 terms where $z_{nk} = 1$ or $z_{nk} = 0$. The sum over z_{nj} means given n summing over all possible latent variable $\mathbf{z}_n$ which takes K different values ($z_{nj} = 1$ for one $j \in [K]$). Then the expectation of the complete-data log-likelihood under the posterior distribution w.r.t. the latent variables $\mathbf{Z}$ is

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})] = \sum_{n=1}^N \sum_{k=1}^K \gamma(z_{nk}) \left(\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right). \quad (4.10)$$

Now if we apply the EM algorithm we will get exactly the same results as (4.5), (4.6), and (4.8). Specifically, we first initialize the model parameters $\boldsymbol{\mu}^{\text{old}}$, $\boldsymbol{\Sigma}^{\text{old}}$, and $\boldsymbol{\pi}^{\text{old}}$ for some values, and use these to evaluate the responsibilities (E step). We then maximize (4.10) with fixed responsibilities over model parameters.

Relation to K-means

Comparison of the K-means algorithm with the EM algorithm for Gaussian mixtures shows a close similarity. The K-means algorithm performs a *hard* assignment of data points to clusters, in which each data point is assigned uniquely to one cluster, while the EM for Gaussian mixtures makes a *soft* assignment based on the posterior probabilities. In fact, we can derive the K-means algorithm as a particular limit of EM for Gaussian mixtures as follows.

Consider a Gaussian mixture model with covariance matrix $\epsilon\mathbf{I}$, $\epsilon \to 0$ for all components. Then we can compute the responsibility for z_{nk} as

$$\gamma(z_{nk}) = \frac{\pi_k \exp\{-\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2/2\epsilon\}}{\sum_j \pi_j \exp\{-\|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2/2\epsilon\}}. \quad (4.11)$$

If $\epsilon \to 0$, we see that $\gamma(z_{nk})$ will go to zero except for j which indicates the closest $\boldsymbol{\mu}_j$ for $\mathbf{x}_n$, and $\gamma(z_{nj})$ will go to unity. In this limit, we obtain a hard assignment of points $\gamma(z_{nk}) = r_{nk}$, just as in K-means algorithm. Each data point is thereby assigned to the cluster having the closest mean. Applying (4.11) to (4.10), one can show that the EM updates are exactly the same as the k-means updates.

4.3.5 The EM Algorithm in General

Finally, we briefly discuss the EM algorithm in general. As mentioned, EM is a general technique for finding maximum likelihood solutions for probabilistic models having latent variables. Consider a probabilistic model with observed variables $\mathbf{X}$, latent variables $\mathbf{Z}$, and parameters $\boldsymbol{\theta}$. Our goal is to maximize the likelihood function given by

$$p(\mathbf{X}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}).$$

We shall suppose that direct optimization of $p(\mathbf{X}|\boldsymbol{\theta})$ is hard but significantly easier for $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$. Next we introduce a distribution $q(\mathbf{Z})$ defined over the latent variables, and then for any choice of $q(\mathbf{Z})$ we have the following decomposition

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q||p).$$

where we have defined

$$\mathcal{L}(q, \boldsymbol{\theta}) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \right\} \tag{4.12}$$

$$\mathrm{KL}(q||p) = -\sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \right\}. \tag{4.13}$$

To verify the decomposition we substitute $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) = p(\mathbf{X}|\boldsymbol{\theta})p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$ into (4.12) to give

$$\begin{aligned}
\mathcal{L}(q, \boldsymbol{\theta}) &= \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{X}|\boldsymbol{\theta})p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \right\} \\
&= \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln p(\mathbf{X}|\boldsymbol{\theta}) + \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \right\} \\
&= \ln p(\mathbf{X}|\boldsymbol{\theta}) - \mathrm{KL}(q||p).
\end{aligned} \tag{4.14}$$

Since we always have $\mathrm{KL}(q||p) \geq 0$, with equality holds if, and only if, $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$, $\mathcal{L}(q, \boldsymbol{\theta})$ is therefore a lower bound of the log-likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$ as illustrated in Figure 4.3.

In the E step of the EM algorithm, the lower bound is maximized with respect to $q(\mathbf{Z})$ with $\boldsymbol{\theta} = \boldsymbol{\theta}^{\text{old}}$ fixed. From (4.14), we can easily see that the maximum is obtained when the KL divergence vanishes since $\ln p(\mathbf{X}|\boldsymbol{\theta}^{\text{old}})$

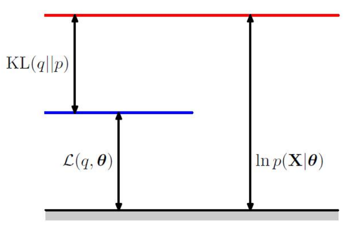

KL$(q||p)$
$\mathcal{L}(q,\boldsymbol{\theta})$
$\ln p(\mathbf{X}|\boldsymbol{\theta})$

Figure 4.3: Log-likelihood decomposition.

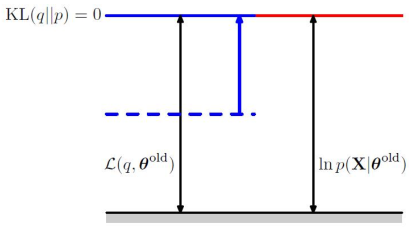

KL$(q||p) = 0$
$\mathcal{L}(q,\boldsymbol{\theta}^{\text{old}})$
$\ln p(\mathbf{X}|\boldsymbol{\theta}^{\text{old}})$

Figure 4.4: The E step of the EM algorithm.

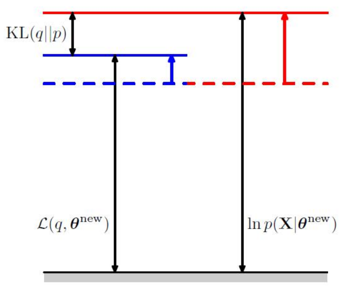

KL$(q||p)$
$\mathcal{L}(q,\boldsymbol{\theta}^{\text{new}})$
$\ln p(\mathbf{X}|\boldsymbol{\theta}^{\text{new}})$

Figure 4.5: The M step of the EM algorithm.

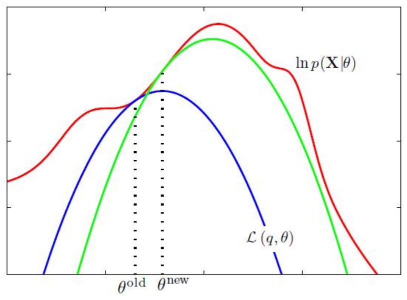


Figure 4.6: The updates of EM.

does not depend on $q(\mathbf{Z})$. The E step is illustrated in Figure 4.4. This means that in the E step, we find the posterior probability $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta})$ by maximizing the lower bound $\mathcal{L}(q,\boldsymbol{\theta})$ w.r.t. $q(\mathbf{Z})$.

In the M step, the distribution $q(\mathbf{Z})$ is fixed an the lower bound $\mathcal{L}(q,\boldsymbol{\theta})$ is maximized with respect to $\boldsymbol{\theta}$ to give some new value $\boldsymbol{\theta}^{\text{new}}$. This will cause the lower bound $\mathcal{L}(q,\boldsymbol{\theta})$ to increase, and the log-likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta}^{\text{new}})$ is therefore increased. However, these two are no longer equal because in general the KL divergence will not be zero after the update. The increase in the log-likelihood is greater than the increase of the lower bound, as shown in Figure 4.5. If we substitute $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\text{old}})$ into (4.12), we see that, after the E step, the lower bound takes the form

$$\begin{aligned}\mathcal{L}(q,\boldsymbol{\theta}) &= \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\text{old}}) \\ &= \mathcal{Q}(\boldsymbol{\theta},\boldsymbol{\theta}^{\text{old}}) + \text{const}\end{aligned}$$

where the constant is simply the negative entropy of the q distribution and is therefore independent of $\boldsymbol{\theta}$. Thus in the M step, we are maximizing the log-likelihood of the complete-data log-likelihood, just as in the case of the Gaussian mixtures.

The operation of the EM algorithm can also be viewed in the space of parameters, as illustrated in Figure 4.6. The red curve represents the (incomplete-data) log-likelihood, which we want to maximize. We begin at some parameter values $\boldsymbol{\theta}^{\text{old}}$, and in the E step we evaluate the posterior distribution of the latent variables, which yields $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, whose value equals to the log-likelihood at $\boldsymbol{\theta}^{\text{old}}$ (the blue curve). In the M step, we updates $\boldsymbol{\theta}$ by maximizing $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, which gives $\boldsymbol{\theta}^{\text{new}}$. The subsequent E step then constructs a bound that is tangential at $\boldsymbol{\theta}^{\text{new}}$ as shown by the green curve.

4.4 Latent Dirichlet Allocation

4.4.1 Motivation

In the pLSA model, both dimensions of the data matrix are fixed, which means that with pLSA we can only sample words in the vocabulary for existing documents. To sample new documents, one reasonable extension of pLSA is to allow sampling additional rows of the co-occurrence matrix $\mathbf{X}$, as illustrated in Figure 4.7. To do this, we need to be able to sample new topic weights $\mathbf{u}_i = \{u_{1i}, \dots, u_{Ki}\}$ for new documents. Combined with a fixed existing topic-to-word matrix $\mathbf{V}$, we can predict new data rows. This idea motivates the Latent Dirichlet Allocation (LDA), which acts as a Bayesian improvement of pLSA.

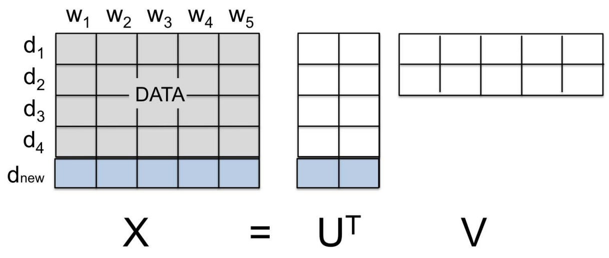


Figure 4.7: We can sample new "documents" by sampling new rows of co-occurrence matrix $\mathbf{X}$.

Remark 4.2 *If* $\mathbf{X}$ *is a count matrix,* $\mathbf{V}$ *has to take into account the number of words and can not be a row-normalized probability matrix since we assume* $\{\mathbf{u}_i\}$ *are normalized, but if* X_{ij} *represents* $p(w_j|d_i)$, $\mathbf{V}$ *can be a topic-to-word probability matrix. Or, alternatively, we can view the equality in Figure 4.7 as "sampled from".*

To sample a new topic weights $\mathbf{u}_i$, we need to make it a probability vector. The constraints, $u_{ij} \geq 0$ and $\sum_j u_{ij} = 1$, indicate that the topics of words that are sampled from $\mathbf{u}_i$ are subject to a multinomial distribution given a particular document d_i. This suggests the choice of Dirichlet distribution as the prior of $\mathbf{u}_i$ since it's the simplest (conjugate) distribution of a multinomial distribution. The word *conjugate* means a reproducing property: if the prior of parameters is from a Dirichlet distribution with the observations from a multinomial distribution, the posterior distribution of the model parameters will remain a Dirichlet distribution after applying the Bayes rule. Concretely, we have

$$p(\mathbf{u}_i|\alpha) \propto \prod_{z=1}^{K} u_{zi}^{\alpha_z - 1}$$

where $\alpha \in \mathbb{R}^K$ are hyper-parameters that can be used to generate topic weights. Note that traditionally, α_z has a interpretation as a pseudo count of the number of times that topic z is chosen which in the posterior distribution will be added with the actual count. Given α and the Dirichlet assumption, we can already sample new topic weights, but with this Bayesian setting we can benefit from doing model averaging. Specifically, we can treat $\mathbf{U}$ as nuisance parameters that need to be averaged out, and $\mathbf{V}$ are real parameters. Note that $\mathbf{U}$ can be re-constructed, if needed.

4.4.2 Model and Algorithm

The LDA defines a vector of word counts that can be regarded as a new document. Specifically, the LDA model with fixed document length $l = \sum_j x_j$ defines a multinomial observation model that is given by

$$p(\mathbf{x}|\mathbf{V}, \mathbf{u}) = \frac{l!}{\prod_j x_j!} \prod_j \pi_j^{x_j}, \; \pi_j := \sum_z v_{zj} u_z$$

where $\mathbf{x}$ is the word count vector, and $l!/\prod_j x_j!$ is the corresponding normalization factor of a multinomial distribution. Then performing Bayesian averaging over the topic weights $\mathbf{u}$ yields

$$p(\mathbf{x}|\mathbf{V}, \alpha) = \int p(\mathbf{x}|\mathbf{V}, \mathbf{u}) p(\mathbf{u}|\alpha) d\mathbf{u}.$$

It corresponds to the following generative model:

- For each document d_i: sample $\mathbf{u}_i \sim \text{Dirichlet}(\alpha)$, where the α are often chosen trivially. Note that the $\mathbf{u}_i$ will be integrated out in Bayesian averaging when we compute the marginal distribution of word count vectors generated from a LDA model.
- For each word slot w^t, $1 \le t \le l_i$, we do the followings in a i.i.d. manner (therefore we can simply treat all word slots by producting them together):
  - Sample topic $z^t \sim \text{Multi}(\mathbf{u}_i)$, which is latent and will be summed out in the marginal distribution of word count vectors generated from this LDA model.

– Sample the desired observation $w^t \sim \text{Multi}(\mathbf{v}_{z^t})$.

There are several algorithms that can be used for LDA. However, since they are beyond the scope of this course, we just list some of them here:

- variational expectation maximization
- Markov Chain Monte Carlo (MCMC): collapsed Gibbs sampling
- distributed, large-scale implementations (100Ms of documents)

4.5 Non-Negative Matrix Factorization

As illustrated in Figure 4.7, pLSA can be regarded as an instance of Non-Negative Matrix Factorization (NMF). Although it seems that we are just add some additional constraints like non-negativity to a standard matrix factorization problem, as we will see in the sequel, NMF has a quite different nature compared with standard MF.

In many cases, such as pLSA, we are dealing with a count matrix (take image reconstruction as an example where we do not deal with a count matrix), and we want to model it in a two-set setting (like topic-word and user-item) with some product of probabilities. We first normalize the rows of the count matrix as $\mathbf{X} \in \mathbb{R}_{\geq 0}^{N \times M}$, and we try to find the NMF of $\mathbf{X}$:

$$\mathbf{X} \approx \mathbf{U}^T\mathbf{V},\ x_{ij} = \sum_z u_{zi}v_{zj} = \langle \mathbf{u}_i, \mathbf{v}_j \rangle \in [0,1]$$

where we have the follow constraints on matrix factors $\mathbf{U}$ and $\mathbf{V}$:

- Non-negativity: all parameters are probabilities.
- Normalization: $\mathbf{U}$ is L_1 column-normalized, and $\mathbf{V}$ is row-normalized.

The approximation quality is measured via log-likelihood of the original count matrix $\mathbf{X}_{count} \in \mathbb{Z}_{\geq 0}^{N \times M}$. NMF, just as standard MF, acts also as a dimension reduction method since $N \cdot M \gg (N+M)K - N - M$, where the minus terms come from the normalization constraints.

There are also variations of NMF, one of which is to do a non-negative matrix approximation, i.e. qualifying the approximation to $\mathbf{X}$ via quadratic cost function without normalization constraints on $\mathbf{U}, \mathbf{V}$. Specifically, we want to solve the following optimization problem:

$$\min_{\mathbf{U},\mathbf{V}} J(\mathbf{U},\mathbf{V}) = \frac{1}{2}\|\mathbf{X} - \mathbf{U}^T\mathbf{V}\|_F^2,\ \text{s.t. } u_{zi}, v_{zj} \geq 0\ (\forall i,j,z). \tag{4.15}$$

This is a similar problem as pLSA, but they are different in the sense of

- different sampling models: quadratic cost function arises naturally from maximum (log) likelihood estimation with Gaussian random variables, while in pLSA we assume a multinomial distribution on observations;
- different objectives: we use quadratic loss instead of KL divergence (for details see the remark below);
- different constraints: we do not assume normalized parameters.

Remark 4.3 *Maximizing the log-likelihood of pLSA model has another interpretation as minimizing a weighted sum of KL divergence between our model predictions and the empirical data distributions. Recall that the log-likelihood of pLSA takes the form of*

$$l(\mathbf{U}, \mathbf{v}) = \sum_i \sum_j x_{ij} \ln p(w_j|d_i).$$

Define $l_i = \sum_j x_{ij}$ as the number of words in d_i, and we can re-write the log-likelihood as

$$\begin{aligned} l(\mathbf{U}, \mathbf{v}) &= \sum_i l_i \sum_j \frac{x_{ij}}{l_i} \ln p(w_j|d_i) \\ &= \sum_i l_i \sum_j p'(w_j|d_i) \left\{ \ln \frac{p(w_j|d_i)}{p'(w_j|d_i)} + \ln p'(w_j|d_i) \right\} \\ &= -\sum_i l_i \mathrm{KL}(p'_i||p_i) + \mathrm{const} \end{aligned}$$

where $p'(w_j|d_i) = x_{ij}/l_i$ is the empirical probability of w_j in d_i, and p'_i and p_i are empirical and predictive distribution of words in d_i respectively. We see that maximizing of the log-likelihood is equivalent to minimize a weighted sum of KL divergences between p'_i and p_i, and the weight factors are numbers of words in documents. Note that the constant term is independent of the model parameters.

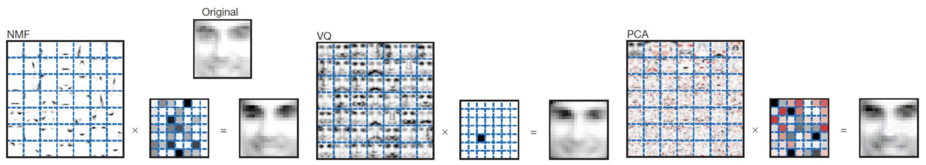


Figure 4.8: Different factorization methods for face reconstruction: NMF (left), Vector Quantization (middle), PCA (right).

We can see the different nature of factorization methods from the part-based face representation example, as illustrated in Figure 4.8. NMF is useful when modelling non-negative data (e.g. images, which are non-negative intensities), and since the non-negative constraint requires the model to do additive superpositions without cancellations, NMF learns to represent faces with a set of basis images resembling parts of face, unlike VQ and PCA (see the left picture of Figure 4.8). However, it is purely a design choice whether to introduce the non-negative constraint.

Let us go back to the NMF with quadratic costs and combine things we have covered to see how we can optimize this objective. Since the objective (4.15) is convex in $\mathbf{U}$ given $\mathbf{V}$ and vice versa, but not jointly in $(\mathbf{U}, \mathbf{V})$, we can choose alternating least squares for optimization. Specifically, we do a alternate optimization of $\mathbf{U}$ and $\mathbf{V}$, keeping the other fixed. Given a fixed $\mathbf{U}$, we can solve for the optimal $\mathbf{V}$ by re-writing (4.15) as

$$\begin{aligned}\min_{\mathbf{V}} \frac{1}{2}\|\mathbf{X} - \mathbf{U}^T\mathbf{V}\|_F^2 &= \min_{\mathbf{V}} \frac{1}{2}\sum_{i,j}(x_{ij} - \mathbf{u}_i^T\mathbf{v}_j)^2 \\ &= \min_{\mathbf{V}} \frac{1}{2}\sum_{j}(\mathbf{X}_j - \mathbf{U}^T\mathbf{v}_j)^2 \\ &= \frac{1}{2}\sum_{j}\min_{\mathbf{v}_j}(\mathbf{X}_j - \mathbf{U}^T\mathbf{v}_j)^2\end{aligned}$$

where $\{\mathbf{X}_j\}$ are columns of $\mathbf{X}$. Therefore, we can solve for $\{\mathbf{v}_j\}$ separately, and each of them can be obtained by solving a standard least-square regression type problem. By expanding the square term and setting the derivative w.r.t. $\mathbf{v}_j$ to zero we get

$$(\mathbf{U}\mathbf{U}^T)\mathbf{v}_j = \mathbf{U}\mathbf{X}_j.$$

Applying the same analysis to $\mathbf{U}$ and writing the outputs in matrix notations yield the *normal equations* given by

$$(\mathbf{U}\mathbf{U}^T)\mathbf{V} = \mathbf{U}\mathbf{X}, \quad \text{and} \quad (\mathbf{V}\mathbf{V}^T)\mathbf{U} = \mathbf{V}\mathbf{X}^T$$

which can be solved via QR-decomposition or gradient descent method. Note that we need to project in between alternations to ensure the non-negativity constraint:

$$u_{zi} = \max\{0, u_{zi}\}, \quad v_{zj} = \max\{0, v_{zj}\}$$

More detailed discussion of algorithms for NMF can be found in Berry, M.W. et al.: Algorithms and applications for approximate non-negative matrix factorization. Computational Statistics & Data Analysis, 52(1), 2007, pp.155-173.

We conclude this chapter with a brief discussion of pLSA and NMF with quadratic loss:

- They are matrix factorization obeying non-negativity and (optionally, pLSA) normalization constraints.

- They have different different cost functions: multinomial likelihood for pLSA, and quadratic loss for NMF (with quadratic loss).
- Both of them use a iterative optimization: EM for pLSA, projected ALS for NMF with quadratic loss.
- As matrix factorization, they benefit from the interpretability of factors: topics, parts, etc.
- They have a wide range of applications.

5

Word Embeddings

From this chapter on we will slowly move towards deep learning and neural stuff. As a first step, we will focus on the embeddings, specifically the embeddings of words. Our discussion will mainly focus on two popular algorithms, namely the skip-gram model and the GloVe, both of which, to some degree, find a mapping from the original discrete word space (recall the 1-hot representation we have seen) to a continues vector space with much lower dimension which keeps the semantic information. We will see the embeddings found by these methods have some interesting properties, and show surprising applications in fields like machine translation.

5.1 Motivation: Word Embeddings

Word embedding is a conceptually easy way to motivate the idea of embeddings. To understand natural languages, we usually have to deal with their atomic units of meaning, which are usually symbols like words or phrases. It's similar for music analysis, where in early times we regard each song as a black box and use techniques like collaborate filtering to do things. However, these symbols (words, phrase, or name of songs) rarely carry their meanings "on them", and sometimes we want to open the black boxes by finding representations that encodes semantics or meaning (like acoustic models for music).

For words understanding, it is widely accepted that the meaning of a word is in its use in that language (Wittgenstein, 1953). It is not difficult to see this by thinking of a monolingual dictionary, where we use easy words to explain the hard ones by showing their usages. Motivated by this observation, our goal is then to find semantic representations of words, or symbols, that can say something about the relationship between different symbols given examples of word uses in a corpus (word occurrences). The most straightforward way to do this is to embed the symbols in a vector space, which is the most basic type of representation, and the structure (e.g. angles, distances) of the vector space should relate to word meanings. In another word, the points that are close in that vector space should correspond to words with similar meanings. For example, automobile and car are not similar in terms of lexicology but should be close in a desired embedding vector space. Although we might lose some crispness for the words, we benefit a lot by moving from a discrete space to a continuous space since that it allows us to use all methods for analyzing continuous representations including the Deep Learning, where the first step is always to find a representation for the original data.

Now the problem is left with how can we create such vectors. It might be tempting to do it in a supervised manner, but in general cases we have no access to the "correct" embeddings and have even no ideas how should they look like. This suggest an unsupervised direction, and we need a way to qualify our models. Recall the linear autoencoder in Chapter 1, where we used a reconstructive objective, for word embeddings we can instead make a prediction game, as suggested by the observation that word meanings are in their usages.

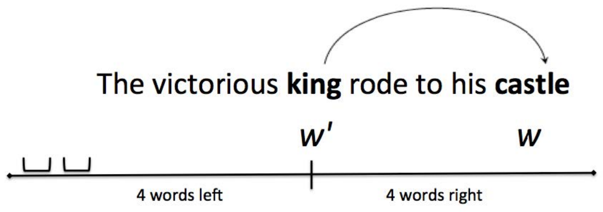


Figure 5.1: An example of the skip-gram model: we choose "king" as the active word and want to predict probabilities of words (say "castle") within a window of size 4.

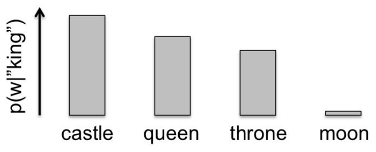


Figure 5.2: The probabilities of some words given word "king". It shows that "castle", "queen", and "throne" appear frequently near "king", and these words are therefore believed to be closely related to "king". We believe that the distribution of co-occurring words itself to some degree determines the lexical semantics

Specifically, we want to predict context words given an "active" word, which is known as the Skip-gram Model. The skip-gram model predicts the probability that w occurs in context window of w': $p_\theta(w|w')$, which captures statistic dependencies between words, as illustrated in Figure 5.1. The skip-gram model, as an instance of Distributional Context Models, or Distributional Semantics Models, assumes that the distribution of co-occurring words determines lexical semantics (Figure 5.2).

5.2 Basic Model

The objective function, or the predictive score, of our model is naturally chosen to be the log-likelihood. Assuming that given w, all words within a window are independent of each other, we obtain

$$\mathcal{L}(\theta; \mathbf{w}) = \sum_{t=1}^{T} \sum_{\Delta \in \mathcal{I}} \log p_\theta(w^{(t+\Delta)} | w^{(t)}) \tag{5.1}$$

where θ represents model parameters and defines the conditional probability distribution over the vocabulary given some words from the vocabulary; $\mathbf{w} = \{w^{(1)}, \dots, w^{(T)}\}$ is the sequence of words and is implicitly padded to handle border cases; $\mathcal{I} = \{-R, \dots, -1, 1, \dots, R\}$ defines a window of offsets, where we want to make prediction. Note that alternatively, we can also consider word pairs within the same sentence rather than in some windows.

Remark 5.1 *In the setting of windows, we neglect $p_\theta(w|w)$ for word w, i.e. $0 \notin \mathcal{I}$, because it seems more or less meaningless to consider the word given itself. Although, it does happen that a word appears multiple times within a window, the model described above works out well in practice.*

Then we can just perform a maximum likelihood estimation to solve or optimal model parameters $\hat{\theta}$:

$$\hat{\theta} = \arg\max_{\theta} \mathcal{L}(\theta; \mathbf{w})$$

which prefers model that assigns high probability to observed context. Now the remaining question is: how should we define an appropriate model $p_\theta(w|w')$?

Recall that we want to find a latent vector representation (embedding) of words. Specifically, our goal is to find the following mapping:

$$w \mapsto (\mathbf{x}_w, b_w) \in \mathbb{R}^{d+1}, \quad \text{(vector + bias)}$$

where for natural languages d is usually chosen to be around 500, and we will see the advantages to separate the bias term shortly. We hope this vector representation captures the semantics of words, as illustrated in Figure 5.3. Given 2 vectors, the inner product is one of the simplest thing we can apply and has a nice interpretation in the sense of measuring similarities. We can therefore define the a log-bilinear model that is given by

$$\log p_\theta(w|w') = \langle \mathbf{x}_w, \mathbf{x}_{w'} \rangle + b_w + \text{const} \tag{5.2}$$

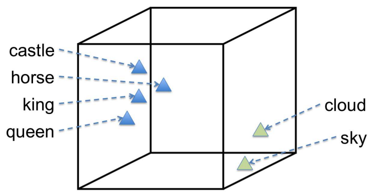


Figure 5.3: An example of words embedded to a vector space that captures semantics.

where we use a symmetric bilinear form to fit the log-probabilities, but as we will see later, we can break the symmetry by introducing a different embedding $y_{w'}$ for the "active" word and use $\langle x_w, y_{w'} \rangle$ instead. Recall that in matrix factorization, we often fit the data directly with the inner products, here everything is in a log scale. The bias term gives some words more importance regardless of the conditional word and therefore to some degree handles different word frequency. The last term is a normalization constant. Then we see the log-bilinear model has the following effects:

- the unspecific effect: $b_w \uparrow \implies p_\theta(w|w') \uparrow \forall w'$,
- the specific effect: $\angle(\mathbf{x}_w, \mathbf{x}_{w'}) \downarrow \implies p_\theta(w|w') \uparrow$.

Therefore we can say that in some sense the inner products model the interactions between words, and the biases describe the marginals of each word.

Remark 5.2 *One might ask, if we embed each word with only one vector, will this be a reasonable semantic representation for words with multiple (especially very different) meanings? For example, the word "cloud" has one meaning that is closely related to "sky", as in Figure 5.3, but also has another meaning which should be closer to "computation". It turns out that in the representation learned by the model we described, the words has close meaning to "sky", as well as words has close meaning to "computation", form a small, i.e. low-dimensional, affine subspace, and the embedding for "cloud" has large inner products with both these two affine subspaces but much smaller for others. Still, multiple representation for a single word is a possible direction for word embedding, but we have to keep in mind that we have no idea what is the right thing to do.*

Now we can derive the normalization constant by first neglecting the constant in our log-bilinear model (5.2) and exponentiating both side to give

$$p_\theta(w|w') = \frac{\exp[\langle \mathbf{x}_w, \mathbf{x}_{w'} \rangle + b_w]}{Z_\theta(w')}$$

where

$$Z_\theta(w') = \sum_{v \in \mathcal{V}} \exp[\langle \mathbf{x}_v, \mathbf{x}_{w'} \rangle + b_v]$$

is a partition function preventing naively making everything large. In the following sections, we will take a closer look into our model and discuss methods to learn the model parameters (word embeddings):

$$\theta = ((\mathbf{x}_w, b_w)_{w\in\mathcal{V}}) \in \mathbb{R}^{(d+1)\cdot|\mathcal{V}|}.$$

Before we go to the learning algorithms of model parameters, it is worth spending sometime taking another view to understand the role of b_w.

Remark 5.3 *The bias term b_w also play a role to avoid $\mathbf{x}_w$ to be too large, which leads to separate modeling of word interactions and marginals. Since in (5.2), there is no explicit regularization on the norm of $\mathbf{x}_w$, then if we want to evaluate the similarity by computing the inner products of word embeddings we need to be more careful. Consider a toy example in $\mathbb{R}^2$, where we have*

$$\mathbf{x}_1 = (1,0),\ \mathbf{x}_2 = (1,0.1),\ \mathbf{x}_2 = (1,10)$$

and

$$\langle \mathbf{x}_1, \mathbf{x}_2\rangle = \langle \mathbf{x}_1, \mathbf{x}_3\rangle.$$

Without a bias term, a model may model different importance of words by assigning embeddings with different length. However, an embedding with larger norm will in general lead to larger inner products in magnitude, which will be problematic for the model interpretability. The introduction of b_w helps relieve this problem, and in this case the $\{\mathbf{x}_w\}$ are usually self-normalized to 1 after training in practice. (This remark is written based on my memory of how Prof Thomas replied to a similar question of mine during the lecture and might not be correct. If you have an idea of how things should be, please send me an e-mail[1] and leave some comments!)

5.3 Skip-Gram Model

Substituting our log-bilinear model (5.2) into the log-likelihood (5.1) yields

$$\begin{aligned}\mathcal{L}(\theta;\mathbf{w}) = \sum_{t=1}^{T}\sum_{\Delta\in\mathcal{I}}[\qquad & b_{w^{(t+\Delta)}} && \text{ok}\\ +&\langle \mathbf{x}_{w^{(t+\Delta)}}, \mathbf{x}_{w^{(t)}}\rangle && \text{bi-linear} \longleftarrow \#1\\ -\log\sum_{v\in\mathcal{V}} & \exp[\langle \mathbf{x}_v, \mathbf{x}_{w^{(t)}}\rangle + b_v] && \text{large cardinality} \longleftarrow \#2\\ &]. \end{aligned}$$

There are two possible modifications to the basic model: one is on the bilinear term, where we introduce extra distributions to make our model more flexible; the other one is on the partition function since the current version requires evaluation on a term with very large cardinality in every iteration.

[1] yufeiyu@student.ethz.ch

5.3.1 Modification # 1: Context vectors

As mention before, we can break the symmetry of the bi-linear model by using different embeddings for the conditional word and the words we want to predict. Specifically, we distinguish the output vocabulary $\mathcal{V}$ and the input vocabulary $\mathcal{C}$ and introduce two different embeddings:

- $\mathbf{x}_w$: output embeddings, $w \in \mathcal{V}$
- $\mathbf{y}_w$: input embeddings, $w \in \mathcal{C}$.

Thus, we now use a mixed inner product instead:

$$\log p_\theta(w|w') = \langle \mathbf{x}_w, \mathbf{y}_{w'} \rangle + b_w + \text{const}.$$

By introducing extra parameters we increase the modeling flexibility but also the model dimensionality. We can alternatively apply a simpler model with $\mathbf{x}_w = \mathbf{y}_w$ for $w \in \mathcal{V} \cap \mathcal{C}$, but it is not very commonly used.

5.3.2 Modification # 2: Objective

The hardness of optimizing the log-likelihood objective (5.1) mainly comes from the partition function. A feasible fix for this problem is to alternatively optimizing other objectives whose learned parameters are also optimal for the original problem, such as

- contrastive divergence (word2vec, Mikolov et al. 2013)
- negative sampling (Mikolov et al. 2013)
- pointwise mutual information (Levy & Goldberg 2014)
- weighted squared loss (GloVe, Pennigton et al. 2013)

which is still a active area of research. In the following, we will discuss the negative sampling method, and we will see the objective used in GloVe in section 5.4.

5.3.3 Negative Sampling

The negative sampling, which is a simplified version for a more general method: *noise contrastive estimation* and is used in the original skip-gram paper, reduces the estimation to binary classification tasks. The main idea is to sample positive word pairs from observed data and negative pairs from another distribution, and we train our model to distinguish there pairs. Specifically, we first introduce a contrastive, or negative, distribution $p_n(i, j)$ that is the probability to generate negative examples of word pairs (w_i, w_j) and can be defined quite arbitrarily. Then with the following notations:

- observed pairs (taken from some windows)$\Longrightarrow$ positive training examples Δ^+
- pairs sampled from $p_n \Longrightarrow$ negative training examples Δ^-

we can perform a logistic regression, i.e. maximizing (recall that we have $\sigma(z) := \frac{1}{1+\exp(-z)}$ and $1 - \sigma(z) = \sigma(-z)$)

$$\begin{aligned}\mathcal{L}(\theta) &= \sum_{(i,j)\in\Delta^+} \log p_\theta((i,j) \in \Delta^+) + \sum_{(i,j)\in\Delta^-} \log p_\theta((i,j) \notin \Delta^+)\\ &= \sum_{(i,j)\in\Delta^+} \log \sigma(\langle \mathbf{x}_i, \mathbf{y}_j\rangle) + \sum_{(i,j)\in\Delta^-} \log[1 - \sigma(\langle \mathbf{x}_i, \mathbf{y}_j\rangle)]\\ &= \sum_{(i,j)\in\Delta^+} \log \sigma(\langle \mathbf{x}_i, \mathbf{y}_j\rangle) + \sum_{(i,j)\in\Delta^-} \log \sigma(-\langle \mathbf{x}_i, \mathbf{y}_j\rangle) \qquad (5.3)\end{aligned}$$

where we usually still use the bias terms, but for simplicity we have absorbed the bias term into the vectors by

$$\mathbf{x}_i \to (\mathbf{x}_i, b_i),\ \mathbf{y}_j \to (\mathbf{y}_j, 1).$$

For the choice of negative distribution, it is good to make it reasonably more random, i.e. to have $p_n(i,j)$ with a positive correlation to $P(w_i)P(w_j)$. In the sampling of positive pairs we still first pick an "active" word w_j and sample w_i from a window. It turns out to work well by first re-using the active words w_j, and sample "random" context words: $w_i \propto P(w_i)^\alpha$ for some positive α, e.g. $\alpha = 3/4$, which magically works well in practice. Although people do not know why this $\alpha = 3/4$ works, it appears to have an effect to exponentially dampen frequent words.

Although it might seems to be reasonable to sample as many positive pairs for negative pairs, it turns out an oversampling by a factor k shows better performance in practice, where usually we have $k = 2-20$. For larger data sets, people usually use smaller k.

Negative Sampling & PMI

Another way to understand what we are doing in negative sampling is by considering what is the best our model can do if it models everything correctly. From the Bayesian view, the best thing we can do for predicting whether a pair is a positive one is to make prediction based on the posterior distribution $p(+|(i,j))$. Applying the Bayes' theorem we obtain

$$p(+|(i,j)) = \frac{p(+)p((i,j)|+)}{p(+)p((i,j)|+) + p(-)p((i,j)|-)} \qquad (5.4)$$

which yields the Bayesian optimal discriminant for $\mathcal{L}$ define in (5.3)

$$h^*_{ij} = \sigma^{-1}(p(+|(i,j))) = \log \frac{p(+|(i,j))}{1 - p(+|(i,j))}$$

where we have used $\sigma^{-1}(z) = \log \frac{z}{1-z}$. The prior in (5.4) is defined by the oversampling rate, i.e. $p(-)/p(+) = k$, and we have

$$p((i,j)|+) = p(w_i, w_j),\; p((i,j)|-) = p_n(w_i, w_j).$$

Substituting these into the posterior (5.4), we get

$$h^*_{ij} = \log \frac{p(w_i, w_j)}{p_n(w_i, w_j)} + \log \frac{1}{k} = \log \frac{p(w_i, w_j)}{p_n(w_i, w_j)} + \log \frac{\kappa}{1-\kappa}$$

where $\kappa = 1/(k+1)$. For $k = 1$ (no oversampling) and $p_n(w_i, w_j) = p(w_i)p(w_j)$, the optimal embedding therefore satisfies

$$\langle \mathbf{x}_i, \mathbf{y}_j \rangle \approx h^*_{ij} = \log \frac{p(w_i, w_j)}{p(w_i)p(w_j)} = \mathrm{PMI}(w_i, w_j) \tag{5.5}$$

where $\mathrm{PMI(w_i, w_j)}$ is the *pointwise mutual information* (PMI) between w_i and w_j. If we write (5.5) in a matrix form, we get

$$\mathbf{X}^T\mathbf{Y} \approx \mathrm{PMI},\; \mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_{|\mathcal{V}|}),\; \mathbf{Y} = (\mathbf{y}_1, \dots, \mathbf{y}_{|\mathcal{C}|}).$$

We see that in this case, the negative sampling method actually performs a low-rank approximation to the pointwise mutual information matrix.

5.4 GloVe

GloVe uses another strategy to deal with the normalization term. The main idea is to change the problem from maximizing the log-likelihood to a regression problem. Before we discuss the details, let us first introduce some notations. First we summarize the data in a co-occurrence matrix given by

$$\mathbf{N} = (n_{ij}) \in \mathbb{N}^{|\mathcal{V}|\cdot|\mathcal{C}|}$$
$$n_{ij} = \#\text{ occurence of } w_i \in \mathcal{V} \text{ in context of } w_j \in \mathcal{C}.$$

Note that $\mathbf{N}$ is a very sparse matrix with most of its entry 0 and can be computed in one pass over the text corpus. The concept "context" can be the window in the skip-gram model, but it is more flexible. Then, the new objective is a weighted least squares fit of the log-counts of observed data given by

$$\mathcal{H}(\theta; \mathbf{N}) = \sum_{i,j} f(n_{ij}) \left(\underbrace{\log n_{ij}}_{\text{target}} - \underbrace{\log \tilde{p}_\theta(w_i|w_j)}_{\text{model}} \right)^2 \tag{5.6}$$

with an unnormalized distribution

$$\tilde{p}_\theta(w_i|w_j) = \exp[\langle \mathbf{x}_i, \mathbf{y}_j \rangle + b_i + c_j]$$

and a weighting function f, which deals with different frequencies of events and avoids computing logarithm for $n_{ij} = 0$. Note that in the basic model, the normalization factor plays the role to avoid naively making everything large to maximize the log-likelihood, here the objective is a two-sided penalty and naturally escape from this problem. Also, we see that if our model fits the data perfectly, the normalized probability $p_\theta(w_i|w_j)$ is equal to the empirical distribution $\frac{\#(w_i,w_j)}{\#w_j}$.

The weighting function is usually chosen to take the following form

$$f(n) = \min\left\{1, \left(\frac{n}{n_{\max}}\right)^\alpha\right\}, \;\alpha \in (0,1] \text{ e.g. } \alpha = \frac{3}{4}.$$

As shown in Figure 5.4, this weighting function has the following motivations:

- the cut-off at $n_{\max}$ limits the influence of large counts (frequent words);
- $f(n) \to 0$ for $n \to 0$: as small counts (rare events) are very noisy;
- the exponent α is usually heuristically chosen.

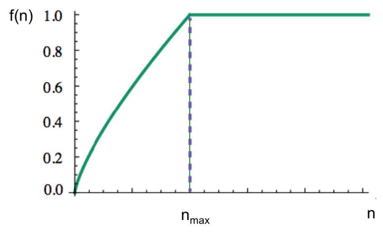


Figure 5.4: The plot of weighting function $f(n)$.

5.4.1 Interlude: Normalized vs. Unnormalized Models

At this point, it is worth comparing normalized models and unnormalized ones in a more general view. A normalized model in general over state space Ω takes the form of

$$p(w) = \frac{\exp[h(w)]}{\sum_{w' \in \Omega} \exp[h(w')]}$$

and requires computation of the partition function. The log-likelihood is given by

$$\mathcal{L} = \sum_t \log p(w_t)$$

and we have

$$h(w) \uparrow \implies p(w) \uparrow \implies \log p(w) \uparrow \implies \mathcal{L} \uparrow,$$

which is counterbalanced by normalization: we cannot naively make $h(w)$ large for every w in order to maximize $\mathcal{L}$.

An unnormalized model in general, on the other hand, usually take the form of

$$\tilde{p}_\theta(w) = \exp[h(w)]$$

where the exponential forces non-negativity, and no computation of partition function is required. For unnormalized models, two-sided loss functions are frequently used to make $\tilde{p}_\theta(w)$ neither too large nor too small. For example, GloVe uses quadratic loss with log-counts as targets.

5.4.2 GloVe Optimization

First we show that the optimization of GloVe can be regarded as a (weighted) matrix Decomposition task. Without loss of generality, we absorb bias into vectors by letting

$$x_{w,d-1} = 1,\ x_{w,d} = b_w \text{ and } y_{w,d-1} = c_w,\ y_{w,d} = 1.$$

Remark 5.4 *In the following discussion from the slides also during the lecture, after using a matrix notation, the constraint on the $(d-1)$-th dimension of $\mathbf{x}_w$ and the d-th dimension of $\mathbf{y}_w$ are ignored. Like in (5.7), the minimization is taken over $\mathbf{X}, \mathbf{Y}$ without explicit constraints. One possible explanation is the constraints are made implicitly, but this contradicts the argument of "go beyond SVD" in the slide.*

Then we can write everything in matrix forms by defining

$$\begin{aligned} &\mathbf{M} = (m_ij),\ m_{ij} := \log n_{ij} \\ &\mathbf{X} := [\mathbf{x}_{w_1} \cdots \mathbf{x}_{w_{|\mathcal{V}|}}],\ \mathbf{Y} := [\mathbf{y}_{w_1} \cdots \mathbf{y}_{w_{|\mathcal{C}|}}]. \end{aligned}$$

We therefore see that for GloVe with $f := 1$, optimizing the objective (5.6) is equivalent to solving a matrix factorization problem:

$$\min_{\mathbf{X},\mathbf{Y}} \|\mathbf{M} - \mathbf{X}^T\mathbf{Y}\|_F^2 \tag{5.7}$$

whose optimal solution is given by (if without the extra constraints, see remark 5.4) Eckart-Young Theorem (1.3) via SVD noting that the form $\mathbf{X}^T\mathbf{Y}$ implies a rank constraint implicitly. However, for a general fixed weight for each entry, we need to go beyond SVD, and this corresponds to solve a matrix factorization problem with a weighted Frobenius norm (Definition 3.4), and is known to be hard in general. However, for GloVe with weighting function

$$f(n_{ij}) := \begin{cases} 1 & \text{if } n_{ij} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

solves a matrix completion problem

$$\min_{\mathbf{X},\mathbf{Y}} \sum_{i,j:n_{ij}>0} (m_{ij} - (\mathbf{X}^T\mathbf{Y})_{ij})^2$$

which falls into a very similar setting we have discussed in section 3.4, and all the methods like alternating least squares can be applied here.

For a general setting of GloVe, the non-convexity makes it hard to find the global minimal, and methods like gradient descent are often used to find a solution. Specifically, for gradient descent (aka steepest descent), we apply the following updating rules until convergence:

$$\theta^{\text{new}} \leftarrow \theta^{\text{old}} - \eta \nabla_\theta \mathcal{H}(\theta; \mathbf{N}), \ \eta > 0 \text{ (step size)}$$

where $\mathcal{H}(\theta; \mathbf{N})$ is the GloVe objective function given by (5.6), and

$$\theta = ((\mathbf{x}_w)_{w\in\mathcal{V}}, (\mathbf{y}_w)_{w\in\mathcal{C}})$$

are model parameters, i.e. embeddings. However, since the full gradient is often too expensive to compute for a single update, stochastic optimization is more commonly used. For stochastic gradient descent (SGD), we sample (i, j) such that $n_{ij} > 0$ uniformly at random, and we perform a "cheap" update for $\mathbf{x}_i$ and $\mathbf{y}_j$ with

$$\begin{aligned}\mathbf{x}_i^{\text{new}} &\leftarrow \mathbf{x}_i + 2\eta f(n_{ij})(\log n_{ij} - \langle \mathbf{x}_i, \mathbf{y}_j\rangle)\mathbf{y}_j \\ \mathbf{y}_j^{\text{new}} &\leftarrow \mathbf{y}_j + 2\eta f(n_{ij})(\log n_{ij} - \langle \mathbf{x}_i, \mathbf{y}_j\rangle)\mathbf{x}_i.\end{aligned}$$

Remark 5.5 *Again we see that there are no explicit constraints on the updates to ensure $x_{w,d-1} = 1$ and $y_{w,d} = 1$. A possible remedy is applying projected gradient descent instead to force the constraints.*

5.5 Word embeddings

The word embeddings found by distributional context models like skip-gram model turn out to capture semantics of words as expected. For example, the nearest words to "frog" in the embedding vector space are actually frogs, as illustrated in Figure 5.5, which is remarkable since the model even do not know what a frog looks like.

Another interesting property of word embeddings is that it has an affine structure which captures analogies and relatedness. Think of a $a : b :: c :?$ type problem, where we need to find a word d that plays a similar role to c as a to b. For example, for man:woman::king:? the word "queen" will be a reasonable candidate for d. We see that, a model solves this type of problem captures word analogies. Mikolov et al. (2013) proposed

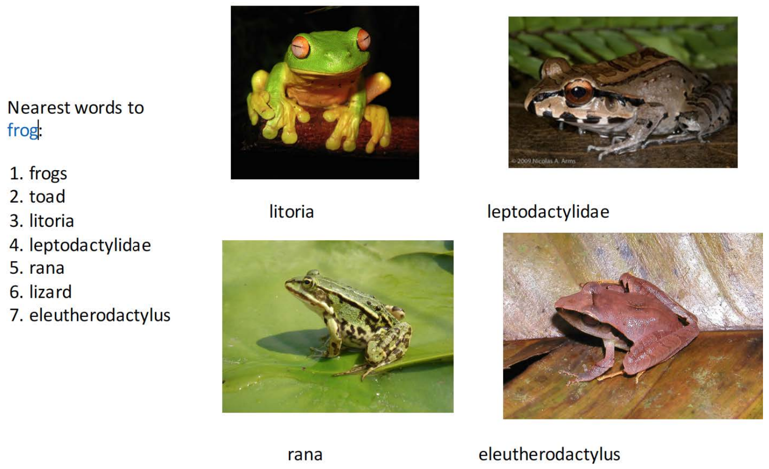


Figure 5.5: The nearest words to "frog" in embedding space.

that simple algebraic operations could be applied to embeddings to find an analogy prediction. Let $\mathbf{x}_a$ be the vector for a and so on. For the d such that the analogy holds, we expect

$$\mathbf{x}_b - \mathbf{x}_a \approx \mathbf{x}_d - \mathbf{x}_c.$$

We can find such d by searching in the vector space for the word closest to $\mathbf{x}_b - \mathbf{x}_a + \mathbf{x}_c$ measured by cosine distance. Specifically, we have

$$d = \underset{i \in \mathcal{V} \backslash \{a,b,c\}}{\arg\max} \cos(\mathbf{x}_i, \mathbf{x}_b - \mathbf{x}_a + \mathbf{x}_c) = \underset{i \in \mathcal{V} \backslash \{a,b,c\}}{\arg\max} \frac{(\mathbf{x}_b - \mathbf{x}_a + \mathbf{x}_c)^T \mathbf{x}_i}{\|\mathbf{x}_b - \mathbf{x}_a + \mathbf{x}_c\|}$$

where the second equality holds for normalized word embeddings. Figure 5.6 shows a 2-d projected visualization for analogous word pairs found by word embeddings. We see that, the word embeddings capture the relatedness of man to woman quite well. However, the antonyms ("cheap" vs. "expensive") are usually not well captured.

The last thing to discuss of word embeddings is the similar structures of different natural languages. The structures of human languages turn out to be quite similar up to orthogonal transformations (OT) (recall that the methods we have proposed can only find solutions that are unique up to OT since the inner products are invariant under OT). For two languages, if we have some word pairs each of which are of the same meaning, we can then use the embeddings of these pairs to do embedding alignment with orthogonal transformations after which we have words with the same meaning correspond to the same (or close) embedding. People have found that, after alignment, unobserved words with the close meanings also have similar embeddings, which is a remarkable result for machine translation. This suggests also an easy way of creating multilingual embedding space.

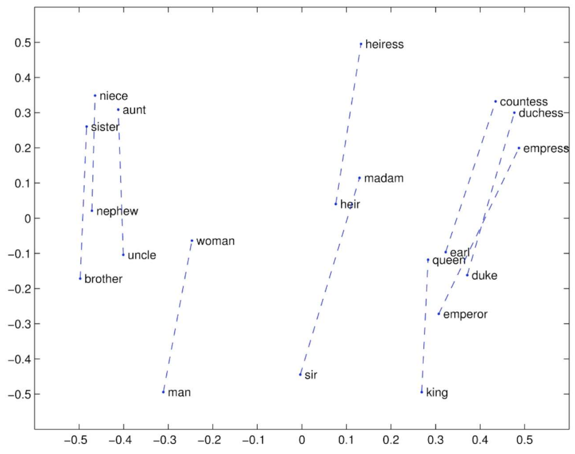


Figure 5.6: 2-d projection for word vector analogies found by optimizing the cosine distance for word embeddings.

It is worth mentioning that the idea of word embeddings can also been generalized to sentence or document embeddings. It need more sophisticated analysis and methods like convolutional or recurrent neural network, but is out of the scope of this course.

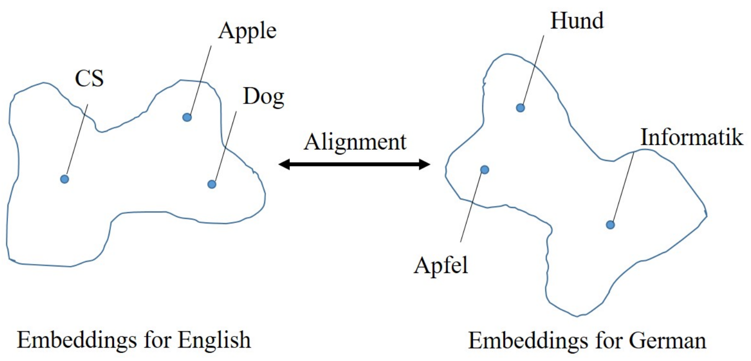


Figure 5.7: Embedding alignment for human languages

6

Data Clustering and Mixture Models

In section 4.3, we discussed K-means and (Gaussian) mixture models in order to motivate the EM algorithm, while in this chapter we revisit these methods from a view of data clustering. Most of the basic ideas have already been covered in section 4.3, and we recommend readers who are interested in more detailed analysis to go through section 4.3. Despite the overlapping of the content, in this chapter we will see some recent extensions for K-means to give better guarantees of the performance and handle large data set. We will also briefly discuss about classical model selection methods.

6.1 Motivation

In previous chapters, we have discussed various methods of data reduction, such as PCA and linear autoencoders. Data clustering, however, provide us a different way to look at the reduction problem. Think about a user analysis problem, where we have data of thousands of users and want to extract some useful information. A natural idea is to group similar users together and represent these users with some representative properties, which can be regarded as the features of a prototype associated with this group. Note that with this representation, data compressing is done at the same time.

In general, given a set of data points $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$, the goal of data clustering is to find a meaningful partition of the data, i.e. an assignment of each data point to a cluster

$$\pi : \{1, \ldots, N\} \to \{1, \ldots, K\}$$

or equivalently a partition of the whole data space

$$\pi : \mathbb{R}^D \to \{1, \ldots, K\}$$

where K is the number of cluster and can be quite arbitrary. We can also recover the j^{th} cluster by

$$\pi^{-1}(j) \in \{1, \dots, N\} \text{ or } \in \mathbb{R}^D.$$

Note that $\pi^{-1}(j)$ may not be a function since it maps a number to a set or region. As suggested before, we cluster points via some "similarities" that might help to uncover the hidden group structure of the data, and we can also learning a data density with proper methods. As an example, with the Euclidean distance as the similarity measure and clusters represented by centroids $\mathbf{u}_j \in \mathbb{R}^D$, we can partition the space $\mathbb{R}^D$ by a mapping induced via a nearest centroid rule

$$\pi(\mathbf{x}) = \underset{j=1,\dots,K}{\arg\min} \|\mathbf{u}_j - \mathbf{x}\|.$$

As illustrated in Figure 6.1, known as the Voronoi (or Dirichlet) tessellation of $\mathbb{R}^D$, this forms the main idea of *Vector Quantization.*

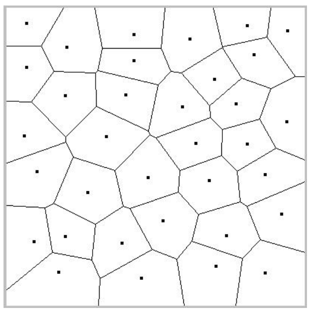


Figure 6.1: An example of Voronoi (or Dirichlet) tessellation, where we see a line connecting 2 points (centroids) will across orthogonally the boundary between the 2 points as implied by the nearest centroid rule.

6.2 K-means

6.2.1 Basic Model

K-means, as one of the most basic algorithm developed in 50s, formalizes the clustering problem as an optimization problem, where we aim to find centroids $\mathbf{u}_j \in \mathbb{R}^D$ and assignment π that minimize a loss function or distortion, e.g. squared Euclidean norm. Concretely, we encode π via an indicator matrix $\mathbf{Z} \in \{0, 1\}^{N \times K}$, such that

$$z_{ij} := \begin{cases} 1 & \text{if } \pi(\mathbf{x}_i) = j \\ 0 & \text{otherwise} \end{cases}.$$

Note that we have

$$\sum_{j=1}^{K} z_{ij} = 1, \ \forall i.$$

The K-means objective function measures the distortion of replacing each data point by the assigned centroid and thus takes the form of

$$\begin{aligned} J(\mathbf{U}, \mathbf{Z}) &= \sum_{i=1}^{N}\sum_{j=1}^{K} z_{ij}\mathrm{dist}(\mathbf{x}_i, \mathbf{u}_j) = \sum_{i=1}^{N}\sum_{j=1}^{K} z_{ij}\|\mathbf{x}_i - \mathbf{u}_j\|^2 && (6.1)\\ &= \|\mathbf{X} - \mathbf{U}\mathbf{Z}^T\|_F^2 && (6.2)\end{aligned}$$

where

$$\begin{aligned} \mathbf{X} &= [\mathbf{x}_1, \ldots, \mathbf{x}_N] \in \mathbb{R}^{D\times N} \text{ is the data matrix}\\ \mathbf{U} &= [\mathbf{u}_1, \ldots, \mathbf{u}_K] \in \mathbb{R}^{D\times K} \text{ is the centroid matrix.}\end{aligned}$$

We see that (6.2) is a matrix form of the original K-means objective function (6.1) and is followed by

$$(6.1) = \sum_{i=1}^{N} \left\| \mathbf{x}_i - \sum_{j=1}^{K} z_{ij}\mathbf{u}_j \right\|^2 = \sum_{i=1}^{N} \left\| \mathbf{x}_i - (\mathbf{U}\mathbf{Z}^T)_i \right\|^2 = (6.2).$$

The matrix form objective function (6.2) shows that K-means solves a matrix decomposition problem, where again we have a rank constraint that is introduced by K, and $\mathbf{Z}$ is not a real value matrix but a selection matrix with $z_{ij} \in \{0, 1\}$.

As mentioned in section 4.3, directly minimizing the K-means objective (6.1) is NP-hard, but we still have the following observations:

- determining optimal centroids given assignments is easy (continuous variables)
- determining optimal assignments given centroids is easy (integer variables).

These two simple observations suggest an alternating minimization, i.e. the Lloyd's algorithm in section 4.3. Specifically, we alternatively

- compute optimal assignment $\mathbf{Z}$, given centroids $\mathbf{U}$, by mapping each data point to the closest centroid since each data point contributes to exactly one term in the outer sum of objective (6.1)

$$z_{ij}^* = \begin{cases} 1 & \text{if } j = \arg\min_k \|\mathbf{x}_i - \mathbf{u}_k\|^2 \\ 0 & \text{otherwise} \end{cases}$$

- compute optimal choice of $\mathbf{U}$, given assignments $\mathbf{Z}$, by 1st order optimality condition, for non-empty clusters ($\sum_{i=1}^{N} z_{ij} \geq 1$):

$$\nabla_{\mathbf{u}_j} J(\mathbf{U}, \mathbf{Z}) = \sum_{i=1}^{N} z_{ij} \nabla_{\mathbf{u}_j} \|\mathbf{x}_i - \mathbf{u}_j\|^2 \overset{!}{=} 0 \Rightarrow \mathbf{u}_j^* = \frac{\sum_{i=1}^{N} z_{ij}\mathbf{x}_i}{\sum_{i=1}^{N} z_{ij}}$$

  which is known as the centroid condition (center of mass of assigned data points).

We can initialize the centroids on K distinct random data points (different initialization may be applied as well) and repeatedly do the above steps until the assignment does not change. With a consistent way to break the ties, where points have equal distances to multiple centroids, K-means algorithm will converge since the objective decrease monotonically, and the number of different assignment is large but finite. Note that the number of step of K-means algorithm until convergence can be very large for extreme cases. Since sometimes during the alternating optimization some clusters become empty, we can do random re-initialization to avoid this problem.

Remark 6.1 *The initialization of K-means can sometimes be problematic. Thinking about the example shown in Figure 6.2, we see that with the illustrated initialization the final partition might have no cluster taking care of the points on the left. However, it is believed that with a "proper" initialization, K-means will perform well. We will also see more recent extensions of K-means that can handle these problematic cases in the following sections.*

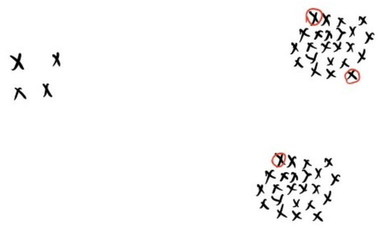


Figure 6.2: An example showing that sometimes the initialization will be problematic for K-means. Since there is no initial centroids in the left group, and there are far more points on the right, the final centroids will probably still be around the two groups to the right.

The computational cost of each iteration (2 steps) is $O(knd)$. Note that the number of step towards convergence can be very large, so the K-means algorithm can be quite expensive in the worst case. However, as mention before, K-means is guaranteed to converge. Although, the solution might not be global optimal since K-means optimizes a non-convex objective.

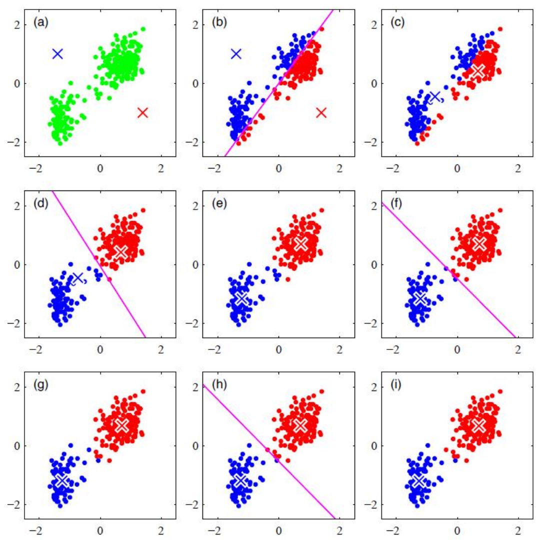


Figure 6.3: Illustration of the K-means algorithm. (Bishop 2006)

6.2.2 K-mean++

K-means++ suggests a more sophisticated seeding for initialization, which was introduced by Arthur & Vassilvitskii, 2007 and has become a standard extension for K-means for really large data set. Recall that the failure case of random initialization for K-means shows that K-means may stuck at the beginning with at bad initialization. To remedy this, K-means++ suggests an *incremental* D^2 (distance square) *sampling* consisting of the following steps

- Starting with an initial centroid set $\mathcal{U}_1 = \{\mathbf{x}_I\}$, $I \sim \text{Uniform}[1:N]$.
- For $k = 1, \ldots, K-1$

$$D_i := \min_{\mathbf{u} \in \mathcal{U}_k} \|\mathbf{x}_i - \mathbf{u}\|, \quad \mathcal{U}_{k+1} := \mathcal{U}_k \cup \{\mathbf{x}_i\}, \text{ where}$$

$$I \sim \text{Categorical}(\mathbf{p}), \quad p_i := \frac{D_i^2}{\sum_{i=1}^N D_i^2}.$$

  Here in each step k, we construct a distribution according to the centroids that we have sampled, where points that are far from all sampled centroids will have large probabilities. Also, for the points that has been selected as a centroid it has $p_i = 0$ since $D_i = 0$, which makes sure that there will be no duplicated selection.

This sampling method is more expensive (can be implemented in a paralleled way for efficiency) than do thing all randomly, but gives consistently better experimental results. It has been proved that, K-means gives a theoretical guarantee: $O(\log K)$-competitiveness in expectation. Specifically, compared with the optimal clustering, K-means++ gives a solution that might be off by a $O(\log K)$ factor in terms of, say, sum of distortions in expectation. Although the $O(\log K)$ might not be so favorable, it avoids cases that K-means can be arbitrarily bad or fails completely.

6.2.3 Core Set for K-means

K-means++ provides a proper initialization, but K-means algorithm is still computationally expensive for very large data set. Imagine that we have 100 million users, and the question is do we really need so many data to identify groups? An intuitive idea is: maybe the analysis on properly-chosen 100 thousand users is enough to extrapolate the whole data set. However, can we do better to construct the subset than just do random sampling. This is exactly the question that the core set method for K-means aims to answer. The core set approach constructs a weighted sub-data-set on which we can run algorithms like K-means, and by construction we get guarantees of what holds for the large data set. This is a quite recent results by Bachem, Lucic, Krause, 2018: Scalable k-Means Clustering via Lightweight Coresets.

Mathematically speaking, K-means aims to find a set $\mathcal{U}$ of k cluster centers in $\mathbb{R}^D$ such that the quantization error $\phi_{\mathcal{X}}(\mathcal{U})$ is minimize, where

$$\phi_{\mathcal{X}}(\mathcal{U}) = \sum_{\mathbf{x}\in\mathcal{X}} \mathrm{d}(\mathbf{x},\mathcal{U})^2 = \sum_{\mathbf{x}\in\mathcal{X}} \min_{\mathbf{u}\in\mathcal{U}} \|\mathbf{x}-\mathbf{u}\|^2. \tag{6.3}$$

For a weighted set $\mathcal{C}$ with corresponding weights $\mathbf{w}$, the quantization error is defined as

$$\phi_{\mathcal{C}}(\mathcal{U}) = \sum_{\mathbf{x}\in\mathcal{C}} \mathbf{w}(\mathbf{x})\mathrm{d}(\mathbf{x},\mathcal{U})^2.$$

It is proved that, with the following way of sampling the core set, we have

$$|\phi_{\mathcal{C}}(\mathcal{U}) - \phi_{\mathcal{X}}(\mathcal{U})| \le \epsilon\phi_{\mathcal{X}}(\mathcal{U}) \tag{6.4}$$

with high probability for any $U \subset \mathbb{R}^D$. This shows that we have a control of the performance of K-means on the whole data set.

Concretely, we sample the core set, which can be a multi-set (with entries appears multiple times), of size m in the following way. Note that different from K-means++, this is not initialization but sub-sampling data into a smaller set. We sample the core set from

$$I \sim \mathrm{Categ}(\mathbf{p}), \quad p_i := \frac{1}{2N} + \frac{D_i^2}{2\sum_{j=1}^N D_j^2}, \quad D_i^2 = \|\mathbf{x}_i - \mu\|^2$$

where $\mu := \frac{1}{N}\sum_i \mathbf{x}_i$ and each sample has a relative weight $\frac{1}{mp_i}$. The probability p_i consists of 2 parts: the first component is the uniform distribution that ensures positivity; the second component is from the intuition that

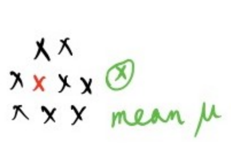


Figure 6.4: Illustration of the core set for K-means algorithm. All data points are represented as a cross with two red ones picked as members of the core set. The mean of the data is represented by the green cross in the circle. The way of sampling make sure that points that are far from the mean have greater probabilities, and the weights $\frac{1}{mp_i}$ compensate for low probabilities of important points near the mean.

points which are far from the mean of the data have a potentially large impact on the quantization error (6.3) of a clustering. The 2nd component ensures that these potentially important points are sampled frequently enough. As illustrated in Figure 6.4, the proposed distribution makes points in the small group more likely to be selected.

To explain the role of the weights, we can again take a look at Figure 6.4, where we have 2 points selected to form the core set. The points close to μ are with larger quantity but low probabilities, while points that are far from μ should be fewer but more likely to be chosen. As a result, we have one point from the larger group and one from the smaller cluster. Intuitively, if we want to recover the clustering property of the original data set, we should assign larger weights to the point that are close to μ to correct for the low probability in selection. The term m in $\frac{1}{mp_i}$ is introduced to make the quantization error of the core set $\phi_{\mathcal{C}}(\mathcal{U})$ an unbiased estimator of the quantization error of the whole data set $\phi_{\mathcal{X}}(\mathcal{U})$. This can be justified by noting that

$$\phi_{\mathcal{X}}(\mathcal{U}) = \sum_{\mathbf{x}\in\mathcal{X}} \mathrm{d}(\mathbf{x},\mathcal{U})^2 = \sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\frac{\mathrm{d}(\mathbf{x},\mathcal{U})^2}{p(\mathbf{x})} = \mathbb{E}\left[\frac{\mathrm{d}(\mathbf{x},\mathcal{U})^2}{p(\mathbf{x})}\right]$$

holds for any positive distribution $p(\mathbf{x})$ over $\mathcal{X}$. The quantization error can hence be approximated by sampling m points from $\mathcal{X}$ using $p(\mathbf{x})$ and assigning them weights inversely proportional to $p(\mathbf{x})$. Then we have an unbiased estimator (empirical expectation via sampling) of the $\phi_{\mathcal{X}}(\mathcal{U})$:

$$\frac{1}{m}\sum_{\mathbf{x}\in\mathcal{C}} \frac{\mathrm{d}(\mathbf{x},\mathcal{U})^2}{p(\mathbf{x})} = \sum_{\mathbf{x}\in\mathcal{C}} \frac{1}{mp(\mathbf{x})}\mathrm{d}(\mathbf{x},\mathcal{U})^2 = \phi_{\mathcal{C}}(\mathcal{U}) \approx \sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})\frac{\mathrm{d}(\mathbf{x},\mathcal{U})^2}{p(\mathbf{x})} = \phi_{\mathcal{X}}(\mathcal{U}).$$

It is proved that this approach gives ϵ-approximation guarantees (6.4) with probability at least $1-\delta$ for

$$m \propto \frac{dk\log k + \log 1/\delta}{\epsilon^2}$$

which shows that the size of core set $\mathcal{C}$ is almost linear in k.

Remark 6.2 *Different from K-means++, in the core set sampling we have $D_i = \|\mathbf{x}_i - \mu\|$ which depends only on $\mathbf{x}_i$ and a fixed value $\mu = \frac{1}{N}\sum_i \mathbf{x}_i$. A natural question arises is that whether the distance to μ is enough to capture the structural properties of a specific point. The answer is probably that it is worth trying. However, we have to keep in mind that we almost always do not know the "right" thing to do, and the proposed core set method works out well for complete data as well in practice. This might suggest that it is better not to make assumptions that are too complicated.*

6.3 Mixture Models

This is a field in statistics that is closely related to K-means. One way to motivate the idea of mixture models is to go from hard assignments in K-means, where each data is assigned to exactly on cluster, to probabilistic assignments, where we want to assign $\mathbf{x}_i$ to each cluster j with some probability z_{ij}. This leads to generalized or relaxed constraints on the selection matrix $\mathbf{Z}$ given by

$$z_{ij} \in [0,1] \ (\forall i,j), \quad \sum_{j=1}^{K} z_{ij} = 1 \ (\forall i).$$

To do so, we can model each cluster by a probability distribution, which can be the multivariate normal distribution since it is the simplest choice. Just as a quick reminder, the PDF (probability density function) of univariate Gaussian with mean μ and variance σ^2 is defined as

$$p(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right],$$

and an isotropic multivariate normal distribution with mean $\boldsymbol{\mu}$ is given by

$$p(\mathbf{x}; \boldsymbol{\mu}, \sigma) = \prod_{i=1}^{D} \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x_i-\mu_i)^2}{2\sigma^2}\right].$$

In a more general setting, a multivariate normal distribution with covariance matrix $\mathbf{\Sigma}$ takes the form of

$$p(\mathbf{x}, \boldsymbol{\mu}, \mathbf{\Sigma}) = \frac{1}{|\mathbf{\Sigma}|^{\frac{1}{2}}(2\pi)^{\frac{D}{2}}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right]$$

where $\mathbf{\Sigma}$ is symmetric and positive definite. Note that the setting with $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ is generally difficult to estimate for large D since there are in total $D + \frac{D(D+1)}{2}$ parameters.

A finite mixture model is defined as a mixture of k (simple) distribution with some mixing proportions

$$p(\mathbf{x};\theta) = \sum_{j=1}^{K} \pi_j p(\mathbf{x};\theta_j), \quad \theta = (\pi, \theta_1, \dots, \theta_K) \in \mathbb{R}^{K+K\times M}$$

where θ are model parameters, and mixing proportions $\pi \geq 0$, $\sum_{j=1}^{K} \pi_j = 1$. The density function of each component is given by $p(\mathbf{x};\theta_j)$ with $\theta_j \in \mathbb{R}^M$. For mixture models for clustering, π_j has an interpretation as the relative cluster size for cluster j and serves as the probability for each cluster; the location and "shape" of cluster j take a specific form of $p(\mathbf{x};\theta_j)$. As a special case, for Gaussian mixture models, we have

$$\theta_j = (\underbrace{\boldsymbol{\mu}_j}_{\text{location}}, \underbrace{\boldsymbol{\Sigma}_{\mathbf{j}}}_{\text{shape}}).$$

A Gaussian mixture model (GMM) hence takes the form of

$$p(\mathbf{x};\theta) = \sum_{j=1}^{K} \pi_j p(\mathbf{x};\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j).$$

This suggests a two-stage generative model: we can generate a data point as follows

1. sample cluster index from categorical distribution $j \sim \text{Categorical}(\pi)$
2. given j, sample a data point $\mathbf{x}$ from the j^{th} component $\mathcal{N}(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$.

Here cluster index j is a latent variable, and the final outcome $\mathbf{x}$ is what we actually observed. Mathematically speaking, the goal of probabilistic clustering is to compute posteriors of latent cluster memberships. The following part is discussed in more details in section 4.3, for maintaining the complete structure of the notes we will briefly discuss it again here.

By explicitly introducing latent variables into generative models, hopefully we can simplify the calculations. Concretely, we attach a assignment variable $\mathbf{z}$ to each data point $\mathbf{x}$ to form a generic (or complete) data point $(\mathbf{x}, \mathbf{z})$, where

$$\mathbf{z} \in \{0,1\}^K, \quad \sum_{j=1}^{K} z_j = 1.$$

The latent variable $\mathbf{z}$ obeys a categorical distribution

$$\mathcal{P}(z_j = 1) = \pi_j \quad \text{or} \quad p_\pi(\mathbf{z}) = \prod_{j=1}^{K} \pi_j^{z_j}.$$

Thus, the joint distribution over $(\mathbf{x}, \mathbf{z})$ (**complete data** distribution) is given by

$$p(\mathbf{x}, \mathbf{z}; \theta) = p(\mathbf{z}; \theta) p(\mathbf{x}|\mathbf{z}; \theta) = \prod_{j=1}^{K} \pi_j^{z_j} \prod_{j=1}^{K} p(\mathbf{x}; \theta_j)^{z_j} = \prod_{j=1}^{K} [\pi_j p(\mathbf{x}; \theta_j)]^{z_j}$$

where we have used that $p(\mathbf{x}|\mathbf{z}; \theta) = \prod_{j=1}^{K} p(\mathbf{x}; \theta_j)^{z_j}$. With these probabilities, we can play with the posterior assignments. Note that probabilistically speaking, *data generation* is to generate data points $\mathbf{x}$ given cluster assignments $\mathbf{z}$, and data inference is to infer cluster assignment $\mathbf{z}$ given a specific data point $\mathbf{x}$. Recall that we can compute the posterior by the Bayes rule

$$\text{posterior } p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

where $p(A)$ is the prior, $p(B|A)$ is the likelihood, and $p(B)$ refers to evidence. The posterior probabilities for assignments is therefore

$$\mathbb{P}(z_j = 1|\mathbf{x}) = \frac{\mathbb{P}(z_j = 1) p(\mathbf{x}|z_j = 1)}{\sum_{l=1}^{K} \mathbb{P}(z_l = 1) p(\mathbf{x}|z_l = 1)} = \frac{\pi_j p(\mathbf{x}; \theta_j)}{\sum_{l=1}^{K} \pi_l p(\mathbf{x}; \theta_l)}$$

where we have assumed access to parameters $\pi, \{\theta_j = (\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)\}$. Suppose that all the data points are drawn independently. The maximum likelihood estimation (MLE) of mixture models requires to optimize

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{N} \log p(\mathbf{x}_i; \theta) = \arg\max_{\theta} \sum_{i=1}^{N} \log \left[\sum_{j=1}^{K} \pi_j p(\mathbf{x}_i; \theta_j) \right]. \tag{6.5}$$

However, the summation over j inside the logarithm prevents a analytical closed-form solution. A remedy to this problem is the Expectation Maximization (EM) algorithm, which maximize a lower bound on the log-likelihood based on complete data distribution. Specifically

$$\begin{aligned} \log p(\mathbf{x}; \theta) = \log \left[\sum_{j=1}^{K} \pi_j p(\mathbf{x}; \theta_j) \right] &= \log \left[\sum_{j=1}^{K} q_j \frac{\pi_j p(\mathbf{x}; \theta_j)}{q_j} \right] \\ &\geq \sum_{j=1}^{K} q_j [\log p(\mathbf{x}; \theta_j) + \log \pi_j - \log q_j] \end{aligned} \tag{6.6}$$

where q_j is a distribution over $j = 1, \ldots, K$, such that $q_j > 0$, $\sum_j q_j = 1$, and the inequality follows by the Jensen's inequality noting the concavity of logarithm. Note that the summation over i in MLE objective (6.5) is outside of the logarithm, so we can evaluate the contribution of each data point $\mathbf{x}_i$ by the proposed lower bound separately (additive).

In the expectation step (E-step), we optimized bound (6.6) with regard to the distribution q. We formulate the following Lagrangian (decoupled for each data point)

$$\max_q \left\{ \sum_{j=1}^{K} q_j [\log p(\mathbf{x};\theta_j) + \log \pi_j - \log q_j] + \lambda(\sum_{j=1}^{K} q_j - 1) \right\}.$$

Apply the first order optimality condition (setting gradient to zero) we get:

$$\begin{aligned} \log p(\mathbf{x};\theta_j) + \log \pi_j - \log q_j - 1 + \lambda &\overset{!}{=} 0 \iff \\ q_j^* = \frac{\pi_j p(\mathbf{x};\theta_j)}{\sum_{l=1}^{K} \pi_l p(\mathbf{x};\theta_l)} &\overset{\text{Bayes rule}}{=} \mathbb{P}(z_j = 1|\mathbf{x}) \end{aligned}$$

which shows that the optimal q-distribution equals to the posterior given the parameters. In another word, E-step selects the best lower bound on the log-likelihood given θ.

In the maximization step (M-step), we optimize expected complete data log-likelihood with regard to the model parameters. Note that the problem decouples for each cluster and with regard to π. Specifically, combining the MLE objective (6.5) and the lower bound for each point (6.6), we get

$$\begin{aligned} \sum_{i=1}^{N} \log p(\mathbf{x}_i;\theta) &\geq \sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij}[\log p(\mathbf{x}_i;\theta_j) + \log \pi_j - \log q_{ij}] && (6.7) \\ &= \sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij} \log p(\mathbf{x}_i|z_{ij} = 1;\theta)p(z_j = 1) - q_{ij}\log q_{ij} \\ &= \sum_{i=1}^{N}\sum_{j=1}^{K} \mathbb{P}(z_j = 1|\mathbf{x}_i) \log p(\mathbf{x}_i, z_j = 1;\theta) + \text{const} && (6.8) \end{aligned}$$

where q_{ij} and z_{ij} are the q_j in (6.6) and z_j for $\mathbf{x}_i$ respectively, and we have assumed fixed $q_{ij} = \mathbb{P}(z_j = 1|\mathbf{x}_i)$; the constant is for that they are only related to q_{ij}. If we write the first term of (6.8) in a matrix form, we get

$$\sum_{\mathbf{Z}} \mathbb{P}(\mathbf{Z}|\mathbf{X}) \log p(\mathbf{X},\mathbf{Z};\theta) = \mathbb{E}_{\mathbf{Z}|\mathbf{X}} \log p(\mathbf{X},\mathbf{Z};\theta)$$

which shows that we are optimizing the expected complete data log-likelihood. Given fixed $q_{ij} = \mathbb{P}(z_j = 1|\mathbf{x}_i)$, we can again apply the first order optimality condition with Lagrangian to solve for optimal π and $\theta_j = (\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$:

$$\nabla_{\pi_j}\left[\sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij}\log\pi_j + \lambda(\sum_{l=1}^{K}\pi_l - 1)\right] \overset{!}{=} 0 \Rightarrow \pi_j^* = \frac{1}{N}\sum_{i=1}^{N} q_{ij}$$

$$\nabla_{\boldsymbol{\mu}_j}\sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij}\log\mathcal{N}(\mathbf{x}_i;\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j) \overset{!}{=} 0 \Rightarrow \boldsymbol{\mu}_j^* = \frac{\sum_{i=1}^{N} q_{ij}\mathbf{x}_i}{\sum_{i=1}^{N} q_{ij}}$$

$$\nabla_{\boldsymbol{\Sigma}_j}\sum_{i=1}^{N}\sum_{j=1}^{K} q_{ij}\log\mathcal{N}(\mathbf{x}_i;\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j) \overset{!}{=} 0 \Rightarrow \boldsymbol{\Sigma}_j^* = \frac{\sum_{i=1}^{N} q_{ij}(\mathbf{x}_i-\boldsymbol{\mu}_j)(\mathbf{x}_i-\boldsymbol{\mu}_j)^T}{\sum_{i=1}^{N} q_{ij}}.$$

Solving the optimal covariance matrix needs some extra tool of matrix derivative, where one might need:

$$\frac{\partial}{\partial\mathbf{A}}\ln|\mathbf{A}| = (\mathbf{A}^{-1})^T,\ \frac{\partial}{\partial\mathbf{A}}\mathrm{Tr}(\mathbf{AB}) = \mathbf{B}^T.$$

The EM algorithm works by alternate E-steps and M-steps, and both E and M step are maximizing the same objective. As in K-means, it is guaranteed to converge towards a point θ^* (with some convergence criteria), while, again like in K-means, θ^* may not be the global maximizer. As a summary, we have

- E-step: compute probabilistic assignments of points to clusters (keeping their location and shape fixed)
- M-step: recompute optimal cluster locations and shapes, given probabilistic assignments.

To conclude this section, we now spend some time comparing the two clustering algorithms mentioned up to now:

- Assignments:
  - K-means algorithm: hard assignment points to clusters
  - EM algorithm: soft assignment based on posteriors
- Shapes:
  - K-means: spherical cluster shapes, uniform spread
  - EM: can learn covariance matrix (ellipsoid-shaped)
- K-means as a special case: as discussed in section 4.3.4, K-means can be regarded as a special case of Gaussian mixture models with (fixed) covariance $\boldsymbol{\Sigma}_j = \sigma^2\mathbf{I}$ in the limit of $\sigma \to 0$ (hard assignment).

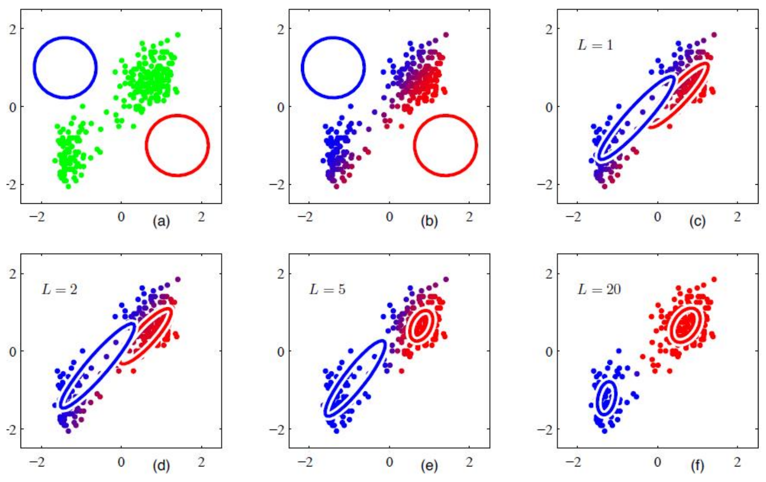


Figure 6.5: Gaussian mixture model fitting via EM for two clusters. Remark: here the covariance is also estimated (illustrated by the two ellipsoids). (Bishop 2006)

In practice, EM algorithm takes much more iterations to converge and in each cycle requires significantly more computation. In this case, K-means can be used to find a good initialization of the EM: covariance matrices can be initialized to the sample covariances of the clusters found by the K-means algorithm; mixing coefficients can be set to the fractions of data points assigned to the respective clusters.

6.4 Model Selection

The last topic of this chapter is about model selection which is often motivated by this kind of questions: what is the proper number of model parameters, or how should we determine the model complexity? In a typical machine learning task people usually have to find trade-off between two conflicting goals:

- Data fit: We want to predict the data well, e.g., maximizing the data log-likelihood. In this case, usually more complex model gives better fit on the observed data.
- Complexity: Choose a model that is not very complex which is often measured by the number of free parameters. Too complex models tend to over fit the observed data and give bad generalization power.

Take the number of clusters in a data clustering problem as an example. Recall that the negative log-likelihood of data for K mixture Gaussians can be written as

$$-\log p(\mathbf{X};\theta) = -\sum_{i=1}^{N} \log \left[\sum_{j=1}^{K} \pi_j p(\mathbf{x}_i;\theta_j) \right].$$

For one specific clustering algorithm like K-means, the negative log-likelihood v.s. number of clusters K usually takes the form as illustrated in Figure 6.6. Note that smaller negative log-likelihood means better fit (to the observed data). As illustrated, in general the objective decreases with K (some noise due to local minima), but very small does not necessarily mean a good fit.

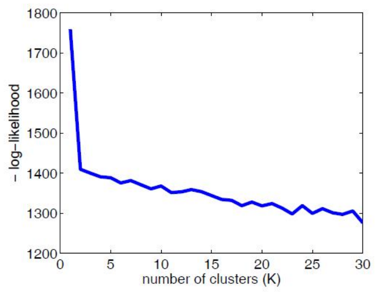


Figure 6.6: A typical negative log-likelihood plot of clustering algorithms

If we measure the model complexity by the number of free parameters $\kappa(\cdot)$, there are two heuristics for choosing K:

- Akaike Information Criterion (AIC):

$$\text{AIC}(\theta|\mathbf{X}) = -\log p(\mathbf{X};\theta) + \kappa(\theta)$$

- Bayesian information Criterion (BIC):

$$\text{BIC}(\theta|\mathbf{X}) = -\log p(\mathbf{X};\theta) + \frac{1}{2}\kappa(\theta)\log N$$

Generally speaking, the BIC criterion penalizes complexity more than the AIC criterion. Note that a single AIC (BIC) result is meaningless. One has to repeat the analysis for different K and compare the differences: the most suitable number of clusters corresponds to the smallest AIC (BIC) value. As an example, we can consider a mixture of Gaussians:

- Number of free parameters (with fixed covariance matrices) is

$$\kappa(\theta) = \underbrace{K \cdot D}_{\boldsymbol{\mu}} + \underbrace{(K-1)}_{\pi}$$

- Number of free parameters (with full covariance matrices) is given by

$$\kappa(\theta) = K \cdot \left(\underbrace{D}_{\boldsymbol{\mu}} + \underbrace{\frac{D(D+1)}{2}}_{\boldsymbol{\Sigma}} \right) + \underbrace{(K-1)}_{\pi}.$$

Figure 6.7 and Figure 6.8 illustrate AIC and BIC examples for 3 and 5 clusters respectively. We can see that in both case, the minimal values of BIC and AIC value correspond to the "right" number of clusters.

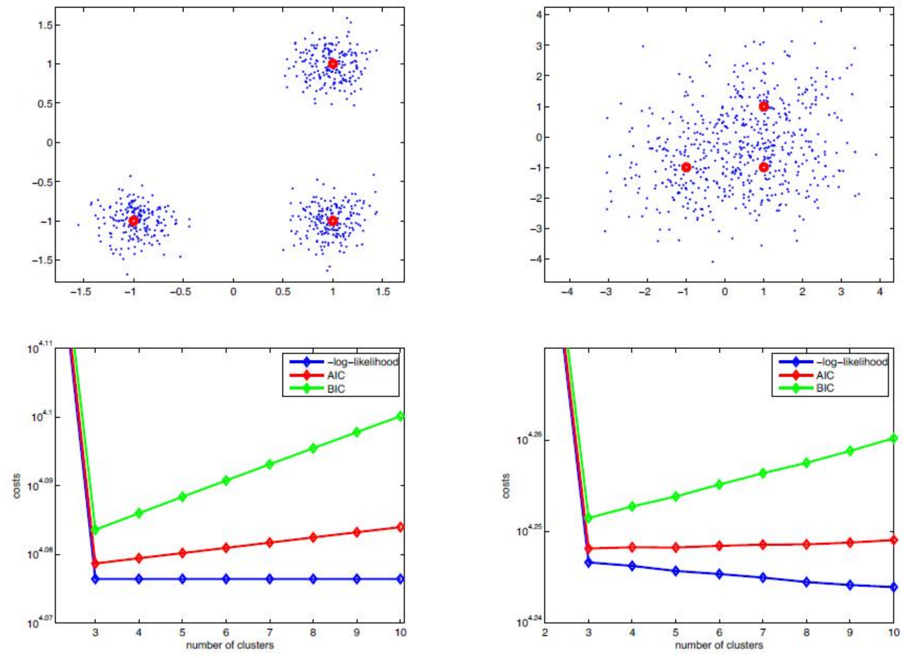


Figure 6.7: Information criteria for a synthetic dataset with 3 clusters. Synthetic data has smaller variance on the left than on the right.

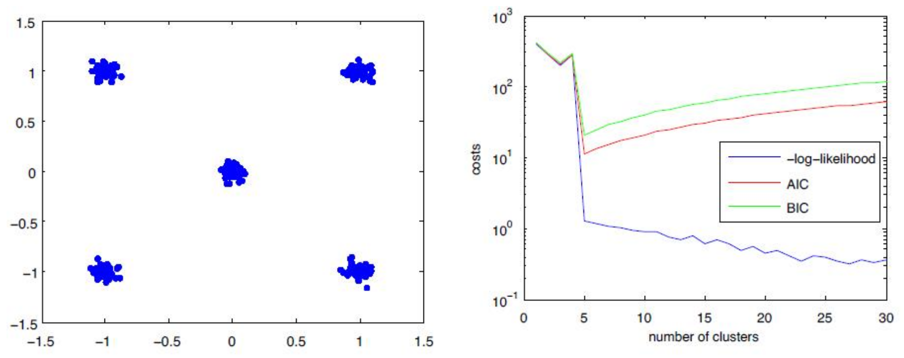


Figure 6.8: Information criteria for a synthetic dataset with 5 clusters.

7

Neural Networks

In the previous chapters we met neural networks several times. We now formally introduce the basic ingredients of them and the most well-known algorithm for parameter learning, backpropagation. From several "modern" extensions of neural networks in the end of this chapter we will get a glimpse of the modeling power of neural networks. Neural networks have been investigated very early in late 50s and 60s as a model of computation for brains and computers. In the 80s with backpropagation and so on and so forth, neural networks succeeded in few things, and were forgot by the research community until their reemergence in around 2010 when deep learning came into play. We start with a standard model.

7.1 Multilayer Perceptrons

7.1.1 An overview

A neural network can be regard as a function that computes outputs given inputs by connected computational units. These units, also refer to neurons, are parametrized by some adaptable weights that can be learned. Each unit implements only a basic operation: a generalized linear function combined with a non-linear activation function. So to speak, each unit computes a non-linear function consisting of

$$\text{linear function}: \ \mathbb{R}^n \to \mathbb{R} \ + \ \text{non-linear activation function } \sigma : \mathbb{R} \to \mathbb{R},$$

where we use linear functions for their simplicity. The non-linear activation function σ is usually fixed and not adaptable. There are several popular choices of σ, but in a particular neural network people usually only use one type of σ. The reason to use this kind of combination to encode non-linearity rather than allow any non-linear function is because we want to keep things as simple as possible in terms of each computational unit (recall that we have the flexibility in connecting them). It is also proved that neural nets with fixed simple activation functions can approximate any non-linear function arbitrarily well. Mathematically, each unit is parametrized with weight $\mathbf{w} \in \mathbb{R}^{n+1}$, and computes

$$f^{\sigma}(\mathbf{x}, \mathbf{w}) := \sigma\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) = \sigma(\mathbf{w}^T \mathbf{x})$$

as shown in Figure 7.1, where in the last equality we ignore or absorb (rewrite $\mathbf{x} \to (1, \mathbf{x})$) bias parameter w_0 for clarity.

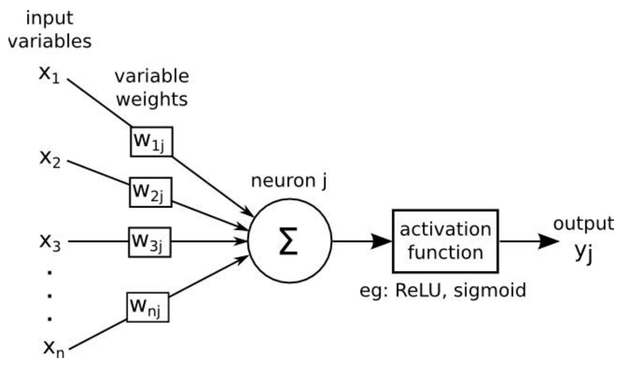


Figure 7.1: A schematic view for a unit in neural networks

7.1.2 Activation function

Despite the arbitrariness of choosing activation functions, there are two types of functions that are mostly used: the old school choice, logistic function (left of Figure 7.2), and the new school choice, ReLU (rectified linear unit) (right of Figure 7.2).

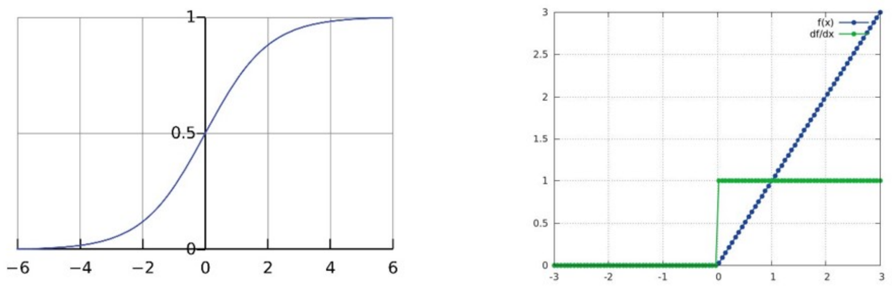


Figure 7.2: Left: logistic function. Right: ReLU.

The logistic function (left of Figure 7.2), also called sigmoid function because of its shape, is given by

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

Note that it can be regard as the same function as tanh up to a rescaling and shifting since

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\frac{e^x}{e^x + e^{-x}} - 1 = 2\frac{1}{1 + e^{-2x}} - 1 = 2\text{sigmoid}(2x) - 1.$$

The logistic function has the property that $x \to \infty, \sigma(x) \to 1$ and $x \to -\infty, \sigma(x) \to 0$. This has a motivation from biology side where this type of saturate effect (maximal level of activity and inactivity) is quite common. Also mathematically speaking, the logistic function is a smooth version of the "switch" function, where by smooth it means differentiable up to any order, and thus has many nice mathematical properties.

On the other hand, ReLU (right of Figure 7.2) appears to be a simpler and more straightforward function defined by

$$\text{ReLU}_{\mathbf{w}}(\mathbf{x}) = \max(0, \mathbf{w}^T\mathbf{x}),$$

where as its name rectified linear unit suggests it is a unit parametrized by weight $\mathbf{w}$. The non-linear activation function of ReLU is just given by $\max(0, x)$. ReLU is a linear function over half-space $\mathcal{H} = \{\mathbf{x} : \mathbf{w}^T\mathbf{x} > 0\}$ and zero on its complement $\mathcal{H}^c = \mathbb{R} - \mathcal{H}$. Different from the logistic function, it's non-smooth, but simple derivatives over $\mathbb{R} - \{0\}$. Therefore, to some degree, we can say that the ReLU function has minimal non-linearity.

7.1.3 Multilayer Perceptron

With a model for each unit, we can put these neurons together to form a computational graph where we do this in terms of layers (Figure 7.3). Although we have the freedom to make (directed) connections quite arbitrarily as long as there's no loop, which is essential for these graphs to be computable, this layer architecture gives a rather simple but still adaptable model enough for exploring.

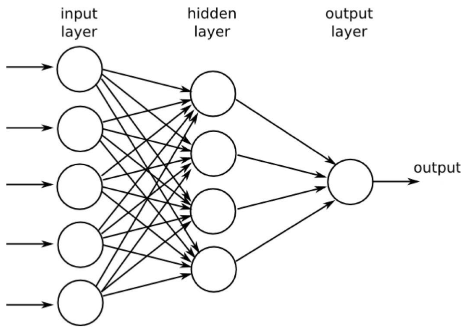


Figure 7.3: A typical structure for neural networks.

Typically, the first layer is the raw input $\mathbf{x}$, where no computation is involved; the last layer serves as the final output whose form depends on specific task (class label for image classification). Although what the input and the output should be is clear in this case, we have no idea about what the intermediate layers represent, and how to learn the weights becomes a big question. To start our discussion, we first build up some useful notations.

As mentioned above, we arrange units in layers. With units indexed by j and a shared choice of σ, we can write the (fully connected) mapping between two layers in a vector form

$$F^{\sigma}: \mathbb{R}^n \to \mathbb{R}^m,\ F_j^{\sigma}(\mathbf{x}) = \sigma(\mathbf{w}_j^T\mathbf{x}),\ j = 1, \dots, m,$$

where n and m represent the number of units of the previous/old and the current/new layer respectively; $\mathbf{x}$ can be the input or activation of the previous layer, and F_j^{σ} denotes a unit in the current layer. We can also take one step further to make everything in a matrix-vector notation (σ is applied elementwise):

$$F^{\sigma}(\mathbf{x}; \mathbf{W}) = \sigma(\mathbf{W}\mathbf{x}),\ \mathbf{W} = (\mathbf{w}_1, \dots, \mathbf{w}_m)^T.$$

If we index layers by l, the activation vector of l-th layer by $\mathbf{x}^{(l)}$, we get a indexed notation for layer-to-layer forward propagation in the form of

$$\mathbf{x}^{(l)} = \sigma^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l-1)}),\ \mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}},$$

where $\mathbf{W}^{(l)}$ is the weight matrix between layer $l-1$ and layer l; $\mathbf{x}^{(1)}$ is the input; $\mathbf{x}^{(L)}$ is the output, and $\mathbf{x}^{(l)}$ $(1 < l < L)$ are the hidden layers.

With these notations, we can represent a L-layer network with a nested function given by

$$\mathbf{y} = \sigma^{(L)}\left(\mathbf{W}^{(L)}\sigma^{(L-1)}\left(\cdots\left(\mathbf{W}^{(2)}\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}^{(1)}\right)\right)\cdots\right)\right).$$

There 2 important degrees of freedom for our layer architecture:

- Layer width: a wider layer means “more of the same” feature since the units in the same layer have the same input, the activation from the previous layer or the input.

- Network depth: different from the layer width, a deeper network allows “more compositionality” meaning that the network can have more ability to combine things and thus capture complex things. By using a deep network, we form a feature hierarchy.

The output layer, as mention before, depends on that type of problem we want to solve, where we may not want to naively apply the same activation function as previous layers. To simplify the notation, we have $\mathbf{W} = \mathbf{W}^{(L)}$, $\mathbf{x} = \mathbf{x}^{(L-1)}$. For a linear regression task, it’s convenient to simply have a linear activation:

$$\mathbf{y} = \mathbf{W}\mathbf{x}.$$

While for a binary classification task, we want only one output ranging from 0 to 1 to form a valid conditional probability distribution. Then the logistic function can be a reasonable choice:

$$y_1 = P(Y = 1|\mathbf{x}) = \frac{1}{1 + \exp[-\mathbf{w}^T\mathbf{x}]}.$$

For the multiclass case with K classes, soft-max function is usually the top choice:

$$y_k = P(Y = k|\mathbf{x}) = \frac{\exp[\mathbf{w}_k^T\mathbf{x}]}{\sum_{j=1}^{K}\exp[\mathbf{w}_j^T\mathbf{x}]}.$$

At this point, it is worth spending some time take a look at the multilayer perceptron and logistic regression in terms of classification methods. In logistic regression, we compute a linear function of inputs and map it to $(0, 1)$:

$$P(Y = 1|\mathbf{x}) = \frac{1}{1 + \exp[-\langle\mathbf{w}, \mathbf{x}\rangle]},$$

where $\mathbf{x}$ can be the original input or some learned better representations. While a multilayer perceptron first tries to learn intermediate feature representation and then perform logistic regression on learned representation $\mathbf{x}^{(L-1)}$. In a word, logistic regression is about: give me good representations I tell you how to linearly separate things, where logistic regression do nothing about the representations. Multilayer perceptron can learn different representations and applies logistic regression to give outputs.

7.1.4 Loss Function

Before we move forward to the learning algorithm, let us first think about what in the model needs to be learned. Things like model architecture (including the number of layers, the number of units in each layer, etc.) and the activation function σ are our choices. We desire algorithms to learn the weights, i.e. algorithms that can automatically fiddling with network weights.

The idea is to make the learning problem an optimization problem. To do so, we first need to define a **loss** function $l(y^*; y)$, where y^* and y are the target output and prediction respectively. Again, the form of loss function depends on the nature of the task. For a regression problem, where $y^*, y \in \mathbb{R}$, we might want to use a squared loss

$$l(y^*; y) = \frac{1}{2}(y^* - y)^2.$$

For a binary classification task, we might desire the cross-entropy loss

$$l(y^*; y) = -y^* \log y - (1 - y^*) \log(1 - y),$$

where $0 \leq y \leq 1$ is used to model a Bernoulli distribution; $y^* \in \{0,1\}$ or $\in [0,1]$ (when $y^* \in [0,1]$ is a real number, this has the interpretation as a soft target encoding the uncertainty about the labeling). There is a close relationship between the cross-entropy loss and the log-likelihood. In general, the cross-entropy $H(p,q)$ measures the difference between distributions q and p:

$$H(p,q) = -\mathbb{E}_p[\log q] = -\sum_{x\in\mathcal{X}} p(x)\log q(x) \text{ (discrete case)}.$$

For a classification problem, if the predicted probabilities are q_i, while the frequency (empirical probability) of i in the training set is p_i, and there are N independent samples in the training set, then the likelihood of the training set is

$$\prod_i (\text{probability of } i)^{\text{number of occurences of } i} = \prod_i q_i^{Np_i}.$$

So the log-likelihood, divided by N is

$$\frac{1}{N}\log\prod_i q_i^{Np_i} = \sum_i p_i \log q_i = -H(p,q).$$

For a single point in a binary classification problem, $p \in \{y^*, 1-y^*\}$ and $q \in \{y, 1-y\}$, then we have

$$H(p,q) = -\sum_i p_i \log q_i = -y^* \log y - (1-y^*)\log(1-y).$$

When we generate y with logistic function, the cross-entropy loss sometimes also refers to negative logistic log-likelihood.

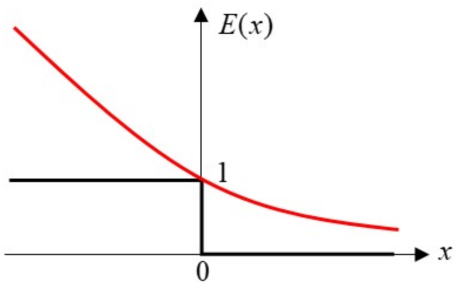


Figure 7.4: The plot of misclassification error (black) and cross-entropy loss (red) for $y^* = 1$. The cross-entropy is rescaled by a factor of $1/\ln 2$, so that it pases trough the point $(0,1)$, and x is the input for logistic function, i.e. $y = 1/(1+\exp[-x])$.

The cross-entropy loss can be regarded as a differentiable approximation or surrogate to the misclassification error as illustrated in Figure 7.4, and can never reach 0 when logistic function is applied.

With a loss function, we can define the objective of the optimization task for parameter learning. Specifically, given a training set of examples $\mathcal{X} - \{(\mathbf{x}_t, y) : t = 1, \ldots, T\}$, we'd like to minimize the empirical risk (average loss on the training data) given by

$$\mathcal{L}(\theta; \mathcal{X}) = \frac{1}{T} \sum_{t=1}^{T} l(y_t; \underbrace{y(\mathbf{x}_t; \theta)}_{\text{NN output}}), \ \theta = \left(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}\right).$$

Sometimes people apply regularization to make networks generalize better. Briefly speaking, by adding regularization we add prior knowledge (not depends on data) like we favor models with small weights, which is believed to be able to relieve overfitting. As an example, we can add L_2 regularization or "weight decay", which favors smaller weights, and get a regularized risk minimization objective:

$$\mathcal{L}_\lambda(\theta; \mathcal{X}) = \mathcal{L}(\theta; \mathcal{X}) + \frac{\lambda}{2} \|\theta\|^2.$$

The name "weight decay" comes from the fact that with L_2 regularization, the weights get shrunk by a factor in each gradient descent update as we will see in the following sections. Besides L_2 regularization, we also use many modern variant, such as drop out (training with noise), where we remove units at random during the training phase. Drop out prevents model from specialize too much.

7.2 Backpropagation

Typically the loss function is non-convex and has no closed-form optimal solutions. Gradient descent method then becomes the workhorse for most of neural network training tasks. However, because of the non-convexity of loss functions, gradient descent methods in this case have no or little theoretical guarantees. Nevertheless, in practice we "just do it", and it works out fine in most cases, while we might suffer from problems caused by poor local minima and saddle points, where the latter one can be more of an issue.

Since for large data set simply applying the steepest descent is too expensive, stochastic gradient descent (SGD) is a more popular choice. SGD with step size η and L_2 regularization with λ picks one data point t at one time (it can also be multiple points, i.e. minibatch):

$$\theta \leftarrow \underbrace{(1 - \eta\lambda)}_{\text{weight decay}} \theta - \eta \nabla_\theta l(y_t^*; y(\mathbf{x}_t; \theta)).$$

The problem we are left with is how to compute the gradients with respect to the weights for a large (many units) and deep (many layers) networks efficiently. This means we have to compute the partial derivative (sensitivity of outputs/loss with regard to each weight) for each weight. Thanks to the layer architecture, we can do this simply by applying the chain rule.

Let us start the computation from the output layer. To compute the derivative w.r.t. the outputs, we just take gradient of the loss function:

$$\nabla_{\mathbf{y}} l = \dots \text{ (depends on loss).}$$

For 1-dimension squared loss, for example, we have

$$\nabla_y = \frac{\partial l}{\partial y} = (y - y^*).$$

To determine how weights affect the output, we look from one layer back each time and evaluate how will these weights change the outputs, and ultimately we can compute the derivative w.r.t. all weights by applying the chain rules. To make this idea concrete, we take a local view with $\mathbf{x}$ to be the previous layer activation and $\mathbf{x}^+$ to be the next layer activation. How $\mathbf{x}$ affect $\mathbf{x}^+$ can be formalized by the Jacobian matrix $\mathbf{J} = (J_{ij})$ of mapping $\mathbf{x} \to \mathbf{x}^+,\ x_i^+ = \sigma(\mathbf{w}_i^T \mathbf{x})$:

$$\mathbf{J} = \frac{\partial \mathbf{x}^+}{\partial \mathbf{x}},\ J_{ij} = \frac{\partial x_i^+}{\partial x_j} = w_{ij} \cdot \sigma'(\mathbf{w}_i^T \mathbf{x}),$$

which is essentially a modified weight matrix. We see that how the change of $\mathbf{x}_j$ affect $\mathbf{x}_i^+$ will be proportional to the original weight w_{ij} linking them and the "sensitivity" of unit $\mathbf{x}_i^+$ measured by $\sigma'(\mathbf{w}_i^T \mathbf{x})$. Then idea of "sensitivity" can be understood by looking at the example of logistic activation (Figure 7.2), where if $|\mathbf{w}_i^T \mathbf{x}|$ is large, the corresponding absolute value of gradient will be small and insensitive to the change of $\mathbf{x}$. Note that if $x_i^+ = \mathbf{w}_i^T \mathbf{x}$ (no activation), then the Jacobian matrix will simply be the weight matrix $\mathbf{W}$.

Since the units in layer $(l-n),\ 1 \le n < l$ can only influence layer l through layer $(l-1)$ (no direct connection), we can compute derivatives across multiple layers by chain rule:

$$\frac{\partial x_i^{(l)}}{\partial x_j^{(l-n)}} = \sum_j \frac{\partial x_i^{(l)}}{\partial x_i^{(l-1)}} \frac{\partial x_i^{(l-1)}}{\partial x_i^{(l-n)}} = \sum_j J_{ij}^{(l)} \frac{\partial x_i^{(l-1)}}{\partial x_i^{(l-n)}}.$$

Use the result we have derived for adjacent layers, we have

$$\frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{x}^{(l-n)}} = \mathbf{J}^{(l)} \cdot \frac{\partial \mathbf{x}^{(l-1)}}{\partial \mathbf{x}^{(l-n)}} = \mathbf{J}^{(l)} \cdot \mathbf{J}^{(l-1)} \cdots \mathbf{J}^{(l-n+1)},$$

which means we only need to multiply (layer-to-layer) Jacobians. Together with the gradient w.r.t. the output, we get

$$\nabla_{\mathbf{x}^{(l)}}^T l = \underbrace{\nabla_{\mathbf{y}}^T l \cdot \mathbf{J}^{(L)} \cdots \mathbf{J}^{(l+1)}}_{\longrightarrow \text{ back propagation}}. \tag{7.1}$$

To determine how weights affect the loss, we consider again where the weights come into play. Since $w_{ij}^{(l)}$ only change unit x_i in layer l directly, all we need is a simple local computation by chain rule

$$\frac{\partial l}{\partial w_{ij}^{(l)}} = \frac{\partial l}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial w_{ij}^{(l)}},$$

where the first term we have already known in (7.1). The second term is

$$\frac{\partial x_i^{(l)}}{\partial w_{ij}^{(l)}} = \underbrace{\sigma'\left(\left[\mathbf{w}_i^{(l)}\right]^T \mathbf{x}^{(l-1)}\right)}_{\text{sensitivity of down-stream unit}} \qquad \underbrace{x_j^{(l-1)}}_{\text{activation of up-stream unit}}.$$

Since we can save the Jacobians along the back propagation, the cost will be roughly the same as a forward propagation. Modern tools like Tensorflow allows to build models with differentiable units and uses the built computational graph together with the chain rule to compute the gradients automatically. We see that, compared with computing the gradients for each weight separately, the symbolic calculation like what we have done can save time and cost by avoiding redundant computation.

7.3 Convolutional Neural Networks

An important observation for model designing in machine learning is that no learning machine can do well on all problems. As a results, we need to constrain our function class appropriately according to different tasks. For vision tasks, the convolutional neural networks are among the most powerful models. The idea comes from the fact that image data is in a way that often translation invariant. For example, the same type of objects should be roughly the same no matter it is in the right or left corner of an image. Therefore, we desire a kind of models that has a topological connectivity which

- encourages network to first extract localized features (ignore a little bit about the location);
- allows subsequent layers to extract less and less localized features.

The idea is to look at a local receptive field and shift all over the image, where we compute the same thing (use the same function) at different locations. The idea of "the same function" is exactly the idea of weight sharing, and can dramatically decrease the parameter-to-data ratio comparing with a fully connection. With this type of design we have shift-invariant filters allowing the model to exploit the translation invariance of images. Since the discussion of how these filters actually work is rather simple but quite tedious, we skip this part and recommend readers to refer to the slides where one can find more figure illustration.

8

Generative Models

In this chapter, we will talk about another interesting thing that is happening in machine learning, particularly in deep neural models, the generative models. Our discussion contains three parts: Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs), and Autoregressive Models. The first two methods are a little bit involved with a bunch of mathematical derivations, while the last one is much more straightforward but still able to produce interesting results.

8.1 Motivation

Let us start with some motivations. Figure 8.1 shows some recent results for face generating. It's based on 4 or 5 years works of the community on generative models. The looks quite amazing in a sense that even if us human are extremely suspicious about faces one can hardly tell that they are synthesized by computers. There's a lot of progress on voice synthesis as well, where people have be able to generate very human-like computer voice and can even imitate one's voice given only 30s voice sample.

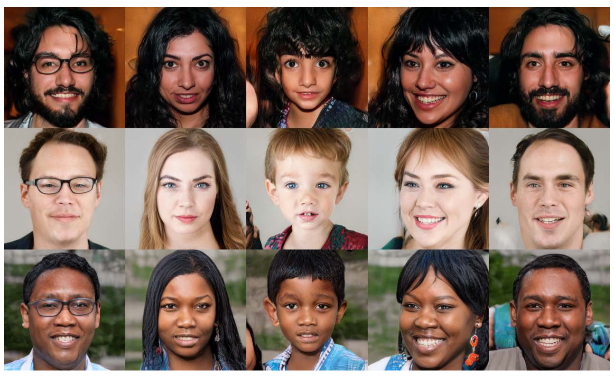


Figure 8.1: Faces generated by StyleGAN (NVIDIA), March 2020

Generally speaking, machine learning for synthesis (or generation) is more fascination than analysis (or recognition). If we have a generative model, rather than a discriminate model providing prediction for labels, we have much more details for what a computer has learned, and we are able to check where are the perceptible mistakes and improve our models. Recent developments in this area open up vastly new possibility for machine learning bringing breakthroughs in fields like animation, games, movies, art and mixed reality. Probably we are witnessing a revolution about how artificial worlds will be created, and how artificial things will be embedded into our real worlds. Though we are really only at the beginning, we will discuss several methods that we deal with in this fascinating area.

8.2 Variational Autoencoders

As the name suggests, VAEs is basically using the idea of Autoencoders (Chapter 1) that performs dimension reduction, take the data, compress it to some lower dimensional space, and then be able to reconstruct from that low dimensional thing. The key idea of VAEs, also GANs, is that if we want to sample or generate random things, we need to put some randomness into our mechanisms.

8.2.1 Deep Generative Models

Since DNNs have the power to create complex distributions, we can start by sampling random vectors from a simple distribution, say a m-dimensional Gaussian: $\mathbb{R}^m \ni \mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ (it can be other distributions as well, it doesn't really matter). We then take this random vector, plug it into a network to output a sample, say an image. The idea is based on that DNNs can implement really complex transformations (recall that in image classification the learned mapping is from thousands of pixels to a single label). Here we go the other way: given a random vector, we use a deep network to transform it into something like a face, and if we sample the random vectors we sample in the face space.

Formally, we want to transform random vectors though a (deterministic) deep network $F_\theta : \mathbb{R}^m \to \mathbb{R}^n$ parametrized by some θ with usually $n \gg m$. For example, with n, as the number of pixels in a image, say 40000, m might be only few hundreds. The distribution of $\mathbf{z}$ together with F_θ induce a (possibly complex) distribution over $\mathbb{R}^n$ with parameters θ. This distribution is easy to sample: we can sample $\mathbf{x}$ by sampling $\mathbf{z}$ and setting $\mathbf{x} = F_\theta(\mathbf{z})$. By the *law of the unconscious statistician* we can compute the expectations over the induced distribution of any function $f(\mathbf{x})$:

$$\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})] = \mathbb{E}_{\mathbf{z}}[f(F_\theta(\mathbf{z}))]. \tag{8.1}$$

This proposition is known as the law of the unconscious statistician because students have been accused of using the identity without realizing that it must be treated as the result of a rigorously proved theorem, not merely a definition (comment from Wiki). A short proof for a invertable and differentiable $F_\theta(\mathbf{z})$ with continuous random variables goes as follow (while the law holds for more general functions)

$$\begin{aligned}
\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})] &= \int p_{\mathbf{x}}(\mathbf{x})f(\mathbf{x})d\mathbf{x} \\
&= \int \left|\frac{\partial F_\theta^{-1}(\mathbf{x})}{\partial \mathbf{x}}\right| f(\mathbf{x})p_{\mathbf{z}}(F_\theta^{-1}(\mathbf{x}))d\mathbf{x} \text{ (change of variables formula)} \\
&= \int p_{\mathbf{z}}(\mathbf{z})f(F_\theta(\mathbf{z}))d\mathbf{z} \ (\mathbf{z} = F_\theta^{-1}(\mathbf{x}),\ d\mathbf{z} = \left|\frac{\partial F_\theta^{-1}(\mathbf{x})}{\partial \mathbf{x}}\right| d\mathbf{x}) \\
&= \mathbb{E}_{\mathbf{z}}[f(F_\theta(\mathbf{z}))].
\end{aligned}$$

According to (8.1), if we want to compute the expectation w.r.t. to the induced distribution over the high-dim space, we could get that expectation by sampling $\mathbf{z}$ from that simpler distribution, sending $\mathbf{z}$ through the mapping F_θ, and averaging them. Why is this relevant? Because it says that to compute the expectations we don't need to know the density of $\mathbf{x}$ if we know the density of $\mathbf{z}$. Since in the discussions involving distributions, very often we need to compute expectations (usually via sample approximation) which can be obtained in this way, no matter what $f(\mathbf{x})$ is.

Remark 8.1 *Even if with the DNN, a distribution on a low-dimensional space induces a distribution over a higher-dimensional space, it's clear that if F_θ is differentiable from ReLU and things we use, it will not fill that space. Generally, the network finds a manifold in that high-dim space, which actually makes sense since as mentioned, in many cases the data are believed to be highly structured in a sense that they lie in a manifold or are sparse in some bases (e.g. it is believed that natural images are sparse in wavelet bases as we will see in the following chapters).*

Usually the way we think about statistical model for generating data is in terms of densities. Recall that we have a simple distribution over $\mathbf{z}$ with the corresponding generated data $\mathbf{x}$ (say a face) induced by a deterministic function F_θ implemented by a DNN: $\mathbf{x} = F_\theta(\mathbf{z})$. Then if F_θ is invertable as well, we can compute the density of $\mathbf{x}$ in terms of the one of $\mathbf{z}$ by the *change of variable formula*:

$$\mathbf{x} = F_\theta(\mathbf{z}), \qquad \underbrace{p_{\mathbf{x}}(\mathbf{x})}_{\mathbf{x}-\text{density}} = \left|\frac{\partial F_\theta^{-1}(\mathbf{x})}{\partial \mathbf{x}}\right| \underbrace{p_{\mathbf{z}}(F_\theta^{-1}(\mathbf{x}))}_{\mathbf{z}-\text{density}}. \tag{8.2}$$

Specifically, we would find the pre-image $\mathbf{z}$ that is mapped to $\mathbf{x}$ by F_θ. Then we could compute the density of that pre-image $\mathbf{z}$, and then we adjust for the volume effect: how much distortio of the volume is locally happening, which is captured by the inverse Jacobian determinant. The density of $\mathbf{x}$ is useful since with it we can do things like maximum likelihood estimation given some real data by computing their likelihood function via (8.2). However, in practice this will be problematic since it requires to

- find the network inversion
- compute the inverse Jacobian determinant
- compute gradients with respect to θ to learn,

some of which are often impossible (non-invertible) or intractable/impractical (dimensionality) avoiding the attempt to construct density directly. For simple F this is okay, but for models we are interested in, i.e. deep models, this shows not much practical success.

8.2.2 Evidence Lower Bound

What can we do if the attempt to get density and evaluate the likelihood function doesn't work? One thing that does work is to apply the principle we have seen in the EM algorithm. In the context here, it refers to the variational lower bound or Evidence Lower BOund (ELBO).

Specifically, instead of using a deterministic F_θ, we relate $\mathbf{x}$ to $\mathbf{z}$ via a slightly more general way, a conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$ parametrized by some θ. This setting allows a little bit noise. Then together with the distribution of the latent variable $p(\mathbf{z})$, we can define a complete data model that is given by

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z}).$$

Integrating out $\mathbf{z}$ yields

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z})d\mathbf{z} = \int p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z},$$

which is the marginal likelihood of $\mathbf{x}$. At this point, one might attempt to maximize the marginal likelihood directly, but we may not know how to perform the integration. Recall what we have seen in EM, to maximize the log marginal likelihood, we can maximize a lower bound of it instead. Again, we first introduce a variational distribution $q_\phi(\mathbf{z}; \mathbf{x})$ and rewrite the log-likelihood as

$$\log p_\theta(\mathbf{x}) = \log \int q_\phi(\mathbf{z}; \mathbf{x}) \frac{p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z}; \mathbf{x})} d\mathbf{z} = \log \mathbb{E}_{q_\phi} \left[\frac{p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z}; \mathbf{x})} \right].$$

Use the concavity and apply the Jensen's inequality yields

$$\begin{aligned} \log p_\theta(\mathbf{x}) \geq \text{ELBO}(\phi, \theta) &= \mathbb{E}_{q_\phi} \left[\log p_\theta(\mathbf{x}|\mathbf{z}) + \log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}; \mathbf{x})} \right] \\ &= \mathbb{E}_{q_\phi} \left[\log p_\theta(\mathbf{x}|\mathbf{z}) \right] - \text{KL}(q_\phi(\mathbf{z}; \mathbf{x}) || p(\mathbf{z})), \end{aligned}$$

where $\mathrm{KL}(q||p)$ is the Kullback-Leibler divergence, which is a measure of distance between distributions. Just as in EM, we can alternatively

- maximize w.r.t. θ (generative model, given q_ϕ), M step in EM;
- maximize w.r.t. ϕ (inference model, given p_θ), E step in EM.

For a discussion for general EM and what do E and M step do please refer to section 4.3.5. Usually in E step maximizing w.r.t. ϕ means we need to compute the posterior of latent variable $p_{\theta^{\text{old}}}(\mathbf{z}|\mathbf{x})$ w.r.t. the old parameters, but for deep models it's too complicated. Instead, we try to compute some approximation via updates with specifically designed q_ϕ.

Before the discussion of how we can maximize w.r.t. θ and ϕ, let us see the VAE in a big picture that is a bit more abstract. A typical VAE can be represented in a diagram shown in Figure 8.2. The right half is the generative path, where we sample $\mathbf{z}$ and pass it through a network to generate data $\mathbf{x}$ (note that this is done by sampling from the distribution $p_\theta(\mathbf{x}|\mathbf{z})$ determined by the generative model). The left half is the inference path, where we do the opposite: conditioning on a data point $\mathbf{x}$, we want to find good Gaussian vectors $\mathbf{z}$ for that data $\mathbf{x}$, which is done in a probabilistic way (output a distribution $q_\phi(\mathbf{z};\mathbf{x})$ rather than a deterministic value). The inference model encodes $\mathbf{x}$ (probabilistically) as a vector $\mathbf{z}$ in a much more lower dimensional space, while the generative model decodes $\mathbf{z}$ to produce data point $\mathbf{x}$. During the train, since we do thing in an unsupervised manner, we want the loop to produce or approximate in some sense the identity mapping.

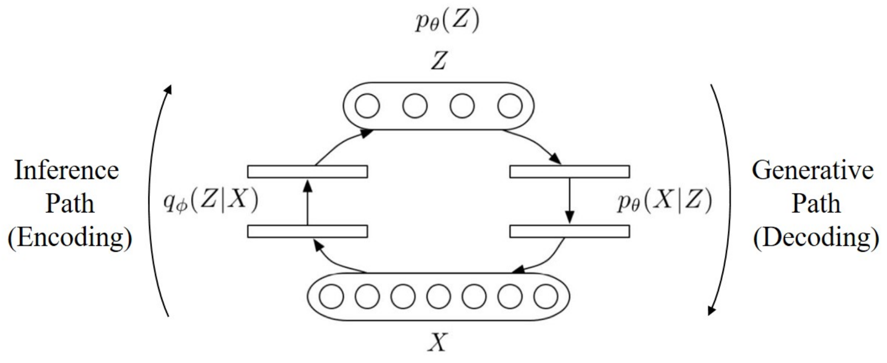


Figure 8.2: The diagram of VAE.

8.2.3 ELBO: Generative Model Updates

To maximize the ELBO w.r.t. θ (update the generative model), we need to be able to compute the gradients w.r.t. θ. The idea is to apply a technique called *stochastic approximation*. Specifically, with the observation that only the first term in ELBO depends on θ we can do the following

$$\nabla_\theta \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] = \mathbb{E}_{q_\phi}[\nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z})].$$

Here we apply the Leibniz integral rule which allows us to exchange $\mathbb{E}_{q_\phi}$ and ∇_θ. This rule does not always hold but for all the cases we are interested in. For the expectation, if we have q_ϕ we can compute a simple Monte Carlo approximation of it by sampling from q_ϕ. Therefore, we obtain

$$\nabla_\theta \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] \approx \frac{1}{L}\sum_{r=1}^{L} \nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z}^{(r)}),\ \mathbf{z}^{(r)} \overset{i.i.d}{\sim} q_\phi(\cdot;\mathbf{x}),$$

where L is the number of sample and is a hyperparameter. For a fixed $\mathbf{z}^{(r)}$, $\nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z}^{(r)})$ is the network gradient which can be obtained with backpropagation. (As shown in Figure 8.2, the generative model outputs a conditional probability distribution over $\mathbf{x}$ given $\mathbf{z}$ rather than a deterministic value. This is a little bit different from the networks we have seen.) This Monte Carlo approximation produces an unbiased gradient estimate (SGD). The gradient is obtained in a sense that is similar to supervised learning, where we have $\mathbf{z}$ as the input and use the output $\mathbf{x}$ to compute the gradient. To show how this is done, for a Gaussian observation model

$$p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(F_\theta(\mathbf{z}), \mathbf{I}), \tag{8.3}$$

the gradient is generated via a squared error $\frac{1}{2}\|F_\theta(\mathbf{z}) - \mathbf{x}\|^2$.

The updates for the generative model turns out to be the easy part in the maximization of ELBO. The inference model performs approximate model inversion in a way that given $\mathbf{x}$ it finds a reasonable $\mathbf{z}$. Determine the gradient w.r.t. ϕ turns out to be more tricky, and we will discuss it in the next section.

8.2.4 ELBO: Inference Model Updates

Let us first look at the update step for the inference model in an abstract view. For a general function $\mathcal{L}(\mathbf{x},\mathbf{z})$, given $\mathbf{x}$ and $\mathbf{z} \sim q_\phi(\mathbf{z};\mathbf{x})$, the gradient w.r.t. ϕ of the expectation of $\mathcal{L}(\mathbf{x},\mathbf{z})$ (w.r.t. q_ϕ) is given by

$$\nabla_\phi \mathbb{E}_{q_\phi}[\mathcal{L}(\mathbf{x},\mathbf{z})] = \underbrace{\int \mathcal{L}(\mathbf{x},\mathbf{z}) \nabla_\phi q_\phi(\mathbf{z};\mathbf{x}) d\mathbf{z}}_{\text{still intractable}} = \mathbb{E}[?]$$

where again we have applied the Leibniz rule. Different from the generative model, here the dependency on ϕ is in the expectation. We can apply the Leibniz rule, but the result is not obviously in an expectation form. Note that if so we can again apply the Monte Carlo approximation by sampling and averaging. Following this idea, one way to rewrite the gradient into a form of expectation is by applying the *reinforce trick* (Williams 1992):

$$\nabla_\phi \mathbb{E}_{q_\phi}[\mathcal{L}(\mathbf{x}, \mathbf{z})] = \mathbb{E}_{q_\phi}[\mathcal{L}(\mathbf{x}, \mathbf{z}) \nabla_\phi \log q_\phi(\mathbf{z}; \mathbf{x})],$$

where we have used

$$\nabla q = q \nabla \log q.$$

Then we can estimate the gradient via sampling, but despite the unbiasedness of this estimation, the variance is usually very high, which requires impractically large number of samples to guarantee a accurate estimation. Nevertheless, this reinforce trick can already be a feasible way to update the inference model. By doing so we have completed the map of optimizing VAEs.

Another way to make the gradient some expectations is by applying the *re-parameterization trick*. Instead of allowing any variational distribution q_ϕ, we use q_ϕ such that

$$\mathbf{z} = g_\phi(\zeta; \mathbf{x}) \quad \text{and} \quad \mathbf{z} \sim q_\phi(\mathbf{z}; \mathbf{x}).$$

where

$$\zeta \sim \text{ simple distribution (e.g. } \mathcal{N}(\mathbf{0}, \mathbf{I})).$$

What we are doing here is basically a change of variable. We transform a very simple random variable ζ using a function $g_\phi(\cdot; \mathbf{x})$ ($\mathbf{x}$ is its parameter) such that the resulting random variable $\mathbf{z} = g_\phi(\zeta; \mathbf{x})$ obeys $q_\phi(\mathbf{z}; \mathbf{x})$, i.e. $\mathbf{z} \sim q_\phi(\mathbf{z}; \mathbf{x})$. With this re-parameterization, the gradient of expectation can be converted into expectation of gradient (known as *stochastic back-propagation*). Specifically, first recall the law of the unconscious statistician (LOTUS) (8.1), we can rewrite the gradient of expectation as

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{\mathbf{z} \sim q_\phi}[\mathcal{L}(\mathbf{x}, \mathbf{z})] &= \nabla_\phi \mathbb{E}_{\zeta \sim \mathcal{N}(\mathbf{0}, \mathbf{I})}[\mathcal{L}(\mathbf{x}, g_\phi(\zeta; \mathbf{x}))] \text{ (LOTUS)} \\
&= \mathbb{E}_\zeta[\nabla_\phi \mathcal{L}(\mathbf{x}, g_\phi(\zeta; \mathbf{x}))] \text{ (Leibniz rule)} \\
&\approx \frac{1}{L} \sum_{r=1}^{L} [\nabla_\phi \mathcal{L}(\mathbf{x}, g_\phi(\zeta^{(r)}; \mathbf{x}))], \ \zeta^{(r)} \overset{i.i.d.}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{I}),
\end{aligned}$$

where ζ can obey other simple distributions as well. As an example, consider $\zeta \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. If we define

$$\mathbf{z} = g_\phi(\zeta; \underbrace{\mu, \mathbf{U}}_{\phi}) = \mu + \mathbf{U}\zeta.$$

Then

$$\mathbf{z} \sim \mathcal{N}(\mu, \mathbf{U}\mathbf{U}^T).$$

In practice, it is often observed that re-parameterization trick leads to lower variance estimates than the reinforce trick, but this does not mean that the re-parameterization is always uniformly better than the other one (no theoretical results).

8.2.5 Deep Latent Gaussian Models

Now we look at an instantiation of the VAE framework we have just discussed about, namely the Deep Latent Gaussian Models (DLGM). In DLGM, we inject randomness in a simple form at each layer to the deep model. Specifically, for each layer we define a noise variable

$$\mathbf{z}^l \overset{i.i.d.}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{I}),\ l = 1, \ldots, L.$$

Let $\mathbf{x}^l$ be the activation for layer l. We start from the generative model of DLGM with top-down-indexed layers (we start from the L-th layer all the way to $\mathbf{x}^1$ from which we can sample data points). The forward propagate starts from the L-th layer with

$$\mathbf{x}^L = \mathbf{W}^L \mathbf{z}^L.$$

The intermediate hidden activities (latent random variables) act in the following way:

$$\mathbf{x}^l = \underbrace{F^l(\mathbf{x}^{l+1})}_{\text{deterministic}} + \underbrace{\mathbf{W}^l \mathbf{z}^l}_{\text{stochastic}},$$

where the first term is deterministic with the layer mapping F^l, and the index $l+1$ shows that we are propagating top-down; the second term shows that we always inject randomness in each layer (via a linear transform of a part of latent variable $\mathbf{z} = (\mathbf{z}^1, \ldots, \mathbf{z}^L)$, see also remark 8.2). The hidden layer (conditional) distribution is then

$$\mathbf{x}^l | \mathbf{x}^{l+1} \sim \mathcal{N}\left(F^l(\mathbf{x}^{l+1}), \mathbf{W}^l \mathbf{W}^{l^T}\right).$$

Finally, we generate pattern $\mathbf{x} \sim \pi(\mathbf{x}^l)$, where $\pi(\cdot)$ is an observation/noise model with parameter $\mathbf{x}^1$ (as an example: the Gaussian observation model (8.3)).

Remark 8.2 *In Figure 8.2, the diagram seems to require the model (networks) to give $q_\phi(\mathbf{z}; \mathbf{x})$ at the last layer of the inference model. However, as the deep latent Gaussian models suggest, this is not the case in the sense that the model can generate the distribution of a part of $\mathbf{z}$ at the intermediate layers. So we see that the VAE shown in Figure 8.2 is an abstract framework, and $q_\phi(\mathbf{z}; \mathbf{x})$ and $p_\theta(\mathbf{x}|\mathbf{z})$ don't even have to be produced by a deep model, but other models work as well.*

For the inference model of DLGM, first recall that given $\mathbf{x}$ the inference model helps to find $\mathbf{z}$ that we have to inject to the generative model to produce such a $\mathbf{x}$. In DLGM, $\mathbf{z}$ consist of all randomness injected in each layer (known as amortized inference):

$$\mathbf{z} = (\mathbf{z}^1, \ldots, \mathbf{z}^L).$$

Assume we draw $\mathbf{z}^l$ independently. The variational distribution $q_\phi(\mathbf{z};\mathbf{x})$ is given by

$$q_\phi(\mathbf{z};\mathbf{x}) = \prod_{l=1}^{L} \mathcal{N}(\mathbf{z}^l|\mu^l(\mathbf{x}), \mathbf{C}^l(\mathbf{x})), \quad \mathbf{C}^l(\mathbf{x}) = \mathbf{U}^l(\mathbf{x})\mathbf{U}^l(\mathbf{x})^T,$$

where μ and $\mathbf{U}$ are represented by DNNs with input $\mathbf{x}$. The update equations for μ in the inference model can be obtained using the Bonnet formula (for $\mathbf{z} \sim \mathcal{N}(\mu, \mathbf{C})$):

$$\nabla_\mu \mathbb{E}_\mathbf{z}[f(\mathbf{z})] = \mathbb{E}_\mathbf{z}[\nabla_\mathbf{z} f(\mathbf{z})].$$

Here's a brief proof for the Bonnet formula:

$$\begin{aligned}
\nabla_{\mu_i}\mathbb{E}_{\mathcal{N}(\mu,\mathbf{C})}[f(\mathbf{z})] &= \int \nabla_{\mu_i}\mathcal{N}(\mathbf{z}|\mu,\mathbf{C}) f(\mathbf{z}) d\mathbf{z} \\
&= -\int \left[\nabla_{z_i}\mathcal{N}(\mathbf{z}|\mu,\mathbf{C})\right] f(\mathbf{z}) d\mathbf{z} \\
&= \left[\int \mathcal{N}(\mathbf{z}|\mu,\mathbf{C}) f(\mathbf{z}) d\mathbf{z}_{\neg i}\right]_{z_i=-\infty}^{z_i=\infty} + \int \mathcal{N}(\mathbf{z}|\mu,\mathbf{C}) \nabla_{z_i} f(\mathbf{z}) d\mathbf{z} \\
&= \mathbb{E}_{\mathcal{N}(\mu,\mathbf{C})}[\nabla_{z_i} f(\mathbf{z})].
\end{aligned}$$

For $\mathbf{U}$, we can use the re-parameterization trick. Recall that we have $\mathbf{z} \sim \mathcal{N}(\mu, \mathbf{C})$, and this can be re-parameterized by $\mathbf{z} = \mathbf{U}\zeta + \mu,\ \zeta \sim \mathcal{N}(0, \mathbf{I})$. Then LOTUS yields

$$\nabla_\mathbf{U}\mathbb{E}_\mathbf{z}[f(\mathbf{z})] = \nabla_\mathbf{U}\mathbb{E}_\zeta[f(\mathbf{U}\zeta+\mu)] = \mathbb{E}_\zeta[\zeta^T\mathbf{g}],\ \mathbf{g} := \nabla_\xi f(\xi)_{|\xi=\mathbf{U}\zeta+\mu}.$$

Remark 8.3 *In the original paper (Rezende et al. 2014), the Price's Theorem and Gaussian backpropagation refer to different things, so there might be some name misusing.*

Finally we can have a look at the big picture of DLGM. As illustrated in Figure 8.3, DLGM consists of 2 parts: the generative model on the left and the recognition (inference) model on the right. The red arrows in the image represent propagation in learning, while the black ones are the forward passes.

To generate new data point $\mathbf{x}$, we sample $\mathbf{z}^l \sim \mathcal{N}(\mu^l, \mathbf{C}^l)$ with given $(\mu^l, \mathbf{C}^l)$ (black horizontal dotted arrows); the activation $\mathbf{x}^l$ are sampled from $\mathcal{N}\left(F^l(\mathbf{x}^{l+1}), \mathbf{W}^l\mathbf{W}^{l^T}\right)$ (black vertical dotted arrows). The red arrows in generative model shows how the backpropagation goes along the layers. Given $\mathbf{x}_0$ and $\mathbf{z}$ produced by the inference model $q_\phi(\mathbf{z}|\mathbf{x}_0)$, the gradients are obtained by the error $l(x_0, \text{Gen}(\mathbf{z}))$, where $\text{Gen}(\mathbf{z})$ is an output of the generative model with $\mathbf{z}$.

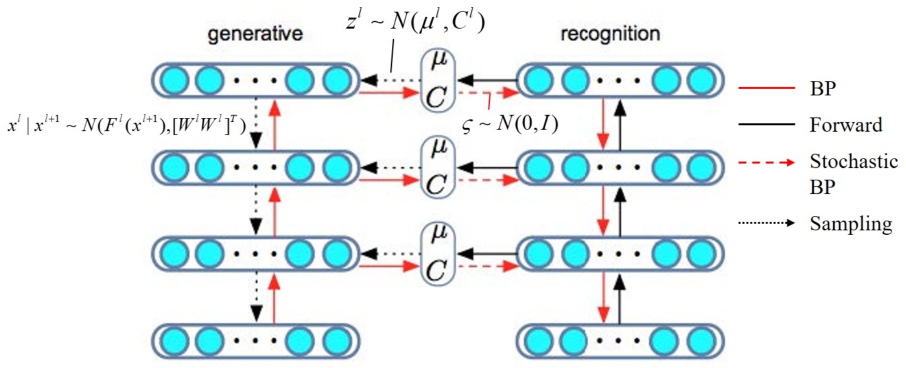


Figure 8.3: Learning and forward propagation scheme for Deep Latent Gaussian Models.

The inference model is similar. The forward pass represented by black arrows are deterministic given $\mathbf{x}$, since $\{\mu(\mathbf{x}), \mathbf{C}(\mathbf{x})\}$ are deterministic functions w.r.t. $\mathbf{x}$. The red dashed arrows are stochastic backpropagation, where we use the re-parameterization trick and need to sample $\zeta \sim (0, \mathbf{I})$.

Here is some example generated by VAEs. As we can see that they are not bad but kind of blurry (Figure 8.4). We will another type of model that is able to generate better results in the next section.

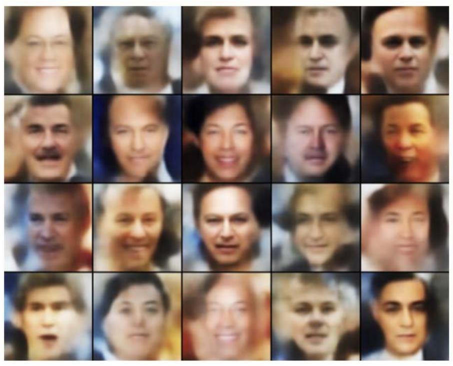


Figure 8.4: Face generation via VAEs.

8.3 Generative Adversarial Networks

8.4 Autoregressive Models

9

Sparse Coding

9.1 Another section heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
Lorem ipsum list:

- Mauris sit amet nulla mi, vitae rutrum ante.
- Maecenas quis nulla risus, vel tincidunt ligula.
- Nullam ac enim neque, non *dapibus* mauris.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis risus ante, auctor et pulvinar non, posuere ac lacus. Praesent egestas nisi id metus rhoncus ac lobortis sem hendrerit. Etiam et sapien eget lectus interdum posuere sit amet ac urna[1]:

[1] Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis risus ante, auctor et pulvinar non, posuere ac lacus.

10

Dictionary Learning

10.1 Another section heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
Lorem ipsum list:

- Mauris sit amet nulla mi, vitae rutrum ante.
- Maecenas quis nulla risus, vel tincidunt ligula.
- Nullam ac enim neque, non *dapibus* mauris.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis risus ante, auctor et pulvinar non, posuere ac lacus. Praesent egestas nisi id metus rhoncus ac lobortis sem hendrerit. Etiam et sapien eget lectus interdum posuere sit amet ac urna[1]:

[1] Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis risus ante, auctor et pulvinar non, posuere ac lacus.

Table 10.1: Sample table

S. No.	Column#1	Column#2	Column#3
1	50	837	970
2	47	877	230
3	31	25	415
4	35	144	2356
5	45	300	556