Sparse and misaligned Data

Jianxin Wu

LAMDA Group
National Key Lab for Novel Software Technology
Nanjing University, China
wujx2001@gmail.com

February 20, 2019

Contents

In the learning and recognition methods we have introduced till now, we have not made strong assumptions about the data: nearest neighbor, SVM, distance metric learning, normalization and decision trees do not explicitly assume distributional properties of the data; PCA and FLD are optimal solutions under certain data assumptions, but they work well in many other situations too; parametric probabilistic models assume certain functional form of the underlying data distribution, but GMM and nonparametric probabilistic methods have relaxed these assumptions.

Many types of data in the real-world, however, exhibit strong characteristics that cannot be ignored. On one hand, some data properties (such as sparsity) are helpful to obtain better representation and accuracy if they are properly utilized. On the other hand, some data properties are adversarial—they will

seriously hurt a pattern recognition system if they are not appropriately handled (such as the misaligned data).

In this chapter, we will discuss two examples that handle such good or bad data characteristics: sparse machine learning and dynamic time warping (DTW).

1 Sparse machine learning

As previously mentioned, a vector is *sparse* if many of its dimensions are 0. If a vector is not sparse, we say it is *dense*. However, in sparse machine learning, we usually do not mean the input features are sparse when we say some data exhibit the sparse property. Given an input vector $\boldsymbol{x}$, sparse machine learning often transforms $\boldsymbol{x}$ into a new *representation* $\boldsymbol{y}$, and the learning process ensures the new representation $\boldsymbol{y}$ is sparse.

1.1 Sparse PCA?

Given a training set $\{\boldsymbol{x}_i\}$ ($\boldsymbol{x}_i \in \mathbb{R}^D$, $1 \le i \le n$), let the new representation for $\boldsymbol{x}_i$ be $\boldsymbol{y}_i \in \mathbb{R}^d$.[1] For example, in principal component analysis, $\boldsymbol{y}_i = E_d^T(\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \in \mathbb{R}^d$, in which E_d comprises of the eigenvectors of the covariance matrix of $\boldsymbol{x}$, corresponding to the d largest eigenvalues. The PCA parameters E_d are learned such that $\|\boldsymbol{y}_i - E_d^T(\boldsymbol{x}_i - \bar{\boldsymbol{x}})\|^2$ is smallest in the average squared error sense.

The new PCA representation $\boldsymbol{y}_i$, however, is not sparse. In order to make $\boldsymbol{y}_i$ sparse, a regularization term is needed. For example, to learn a sparse representation for $\boldsymbol{x}_i$, we can solve the following optimization

$$\min_{\boldsymbol{y}_i} \|\boldsymbol{y}_i - E_d^T(\boldsymbol{x}_i - \bar{\boldsymbol{x}})\|^2 + \lambda \|\boldsymbol{y}_i\|_0 \,, \tag{1}$$

where $\lambda > 0$ is a tradeoff parameter between small reconstruction error and the sparsity of $\boldsymbol{y}_i$, and $\|\cdot\|_0$ is the ℓ_0 "norm". Because $\|\boldsymbol{y}_i\|_0$ is the number of nonzero elements in $\boldsymbol{y}_i$, minimizing this term encourages the solution to have many zero elements, i.e., to be sparse.

There are at least two issues with this formulation. First, ℓ_0 "norm" is not continuous (let alone being differentiable), which makes the optimization very difficult. Second, since E_d is proven to result in the smallest reconstruction error, enforcing sparsity in $\boldsymbol{y}_i$ probably will lead to large reconstruction error because obviously a sparse $\boldsymbol{y}_i$ will be significantly different from $E_d^T(\boldsymbol{x} - \bar{\boldsymbol{x}})$.

There are established solutions to these problems: use the ℓ_1 norm to replace the ℓ_0 "norm", and learn a good *dictionary* from the training data to replace E_d. We will introduce these two methods in the next two subsections. To learn a sparse PCA, we will replace E_d with a dictionary D and learn it using the training data; and we need to replace $\|\boldsymbol{y}_i\|_0$ with $\|\boldsymbol{y_i}\|_1$.

[1] For now we do not consider the labels for $\boldsymbol{x}_i$.

1.2 Using the ℓ_1 norm to induce sparsity

A regularizer $\|\boldsymbol{x}\|_1$ will also encourage the elements of the solution $\boldsymbol{x}^\star$ to be 0. Hence, we often *relax* an ℓ_0 constraint to the ℓ_1 constraint.

The ℓ_1 norm is defined as

$$\|\boldsymbol{x}\| = \sum_{i=1}^{d} |x_i| \,,$$

in which x_i is the i-th element in $\boldsymbol{x}$. This function is convex, which is easy to verify. It is also a continuous function and its gradients exist except when $x_i = 0$ for at least one i. Overall, the ℓ_1 norm is friendly for optimization and a good *surrogate* for the ℓ_0 "norm".

But, why $\|\boldsymbol{x}\|_1$, as a regularizer, leads to sparse solutions? We show some intuitions using a simple example.

Consider a minimization objective

$$f(x) = \frac{1}{2}x^2 - cx \,,$$

in which $c \in \mathbb{R}$ is a constant. If we add a regularizer $\lambda|x|$ ($\lambda\|x\|_1 = \lambda|x|$) to $f(x)$ and denote it as $g(x)$, the optimization problem becomes

$$\begin{aligned} x^\star &= \arg\min_x g(x) && (2)\\ &= \arg\min_x f(x) + \lambda|x| && (3)\\ &= \arg\min_x \frac{1}{2}x^2 - cx + \lambda|x| \,, && (4) \end{aligned}$$

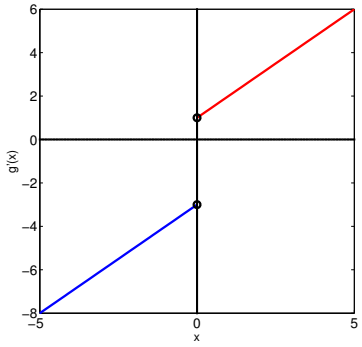
in which $\lambda > 0$. It is obvious that $g(-\infty) = \infty$, $g(\infty) = \infty$, and $g(0) = 0$. Because $|c| < \infty$ and $\lambda > 0$, there does not exist an x such that $g(x) = -\infty$. Hence, there is a global minimum for $g(x) = f(x) + \lambda|x|$.

$g(x)$ is differentiable except at $x = 0$. Its derivation is $x - c - \lambda$ when $x < 0$, and $x - c + \lambda$ when $x > 0$. When $x = 0$, the left derivative is $-c - \lambda$ and its right derivative is $-c + \lambda$. Hence, the optimal value $x^\star$ can only happen in one of the three cases:

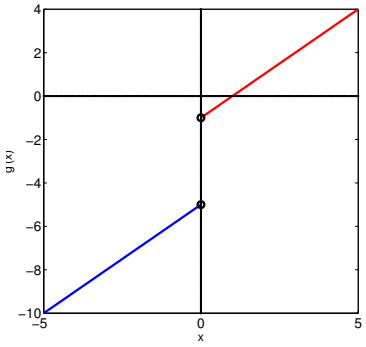
- $x^\star$ is at the non-differentiable point, i.e., $x^\star = 0$; or,
- when one of the two gradient formulas is zero, i.e., $x^\star = c+\lambda$ or $x^\star = c-\lambda$.

Because $\lambda > 0$, the point $\mathrm{P}_1 = (0, -c + \lambda)$ is always above the point $\mathrm{P}_2 = (0, -c - \lambda)$. There are three possible scenarios for the derivatives:

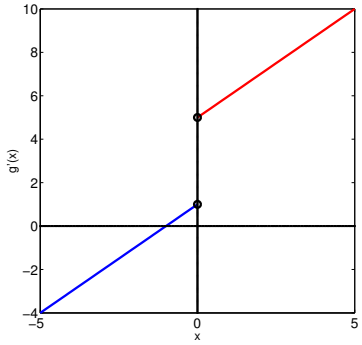
- P_1 is above or on the x-axis, but P_2 is on or below it, as shown in Figure 1a. This can only happen if $-c + \lambda \geq 0$ and $-c - \lambda \leq 0$, i.e., if $|c| \leq \lambda$.
- Both P_1 and P_2 are below the x-axis, as shown in Figure 1b. This can only happen when $-c + \lambda < 0$, or, $c > \lambda > 0$.



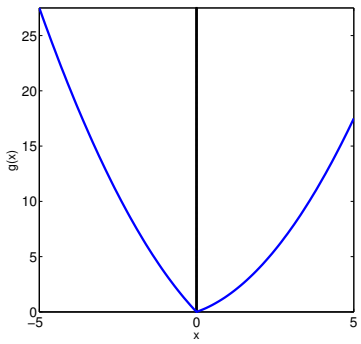
(a) $g'(x)$ when $|c| \leq \lambda$



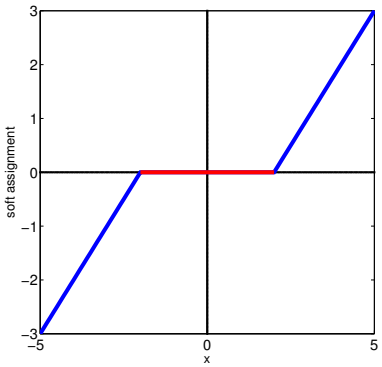
(b) $g'(x)$ when $c > \lambda > 0$



(c) $g'(x)$ when $c < -\lambda < 0$



(d) $g(x)$ when $|c| \leq \lambda$



(e) Soft thresholding

Figure 1: The soft thresholding solution. The first three plots are different cases for the gradient $g'(x)$, and the fourth one is the illustration of the function $g(x)$ when $|c| \leq \lambda$. The last figure shows the soft thresholding solution.

- Both P_1 and P_2 are above the x-axis, as shown in Figure 1c. This can only happen when $-c-\lambda > 0$, or $c < -\lambda < 0$.

In the first case, the only possible solution is $x^\star = 0$ and $g(x^\star) = 0$. Figure 1d shows an example curve of $g(x)$ when $|c| \le \lambda$.

In the second case, the line $x - c + \lambda$ has an intersection with the x-axis at $c - \lambda$ (which is positive), i.e., $g'(c-\lambda) = 0$. Simple calculations show that

$$g(c-\lambda) = -\frac{1}{2}(c-\lambda)^2 < 0 = g(0)\,.$$

Hence, when $c > \lambda$, $x^\star = c - \lambda$.

In the final case, the line $x - c - \lambda$ has an intersection with the x-axis at $c + \lambda$ (which is negative), i.e., $g'(c+\lambda) = 0$. Simple calculations show that

$$g(c+\lambda) = -\frac{1}{2}(c+\lambda)^2 < 0 = g(0)\,.$$

Hence, when $c < -\lambda$, $x^\star = c + \lambda$.

Putting these three cases together, the optimal x is determined by the following *soft thresholding* formula:[2]

$$x^\star = \operatorname{sign}(c)\,(|c| - \lambda)_+\,, \tag{5}$$

in which $\operatorname{sign}(\cdot)$ is the sign function; $x_+ = \max(0, x)$ is the hinge loss, which equals x if $x \ge 0$, and is always 0 if $x < 0$.

The soft thresholding solutions are shown in Figure 1e. Note that the $x^\star$ for $c \in [-\lambda, \lambda]$ is all 0! However, without the regularizer $|x|$, if $x^\star = 0$ for $f(x) = \frac{1}{2}x^2 - cx$, the only possible case is $c = 0$ because $f(x) = \frac{1}{2}(x-c)^2 - \frac{1}{2}c^2$ attains its minimum at $x^\star = c$. That is, adding the ℓ_1 regularizer makes $x^\star = 0$ in many more cases. Hence, the ℓ_1 regularizer is indeed sparsity inducing. When $\boldsymbol{x}$ is a vector, we have reasons to expect the simple ℓ_1 regularization $\|\boldsymbol{x}\|_1$ is effective in making many elements of $\boldsymbol{x}^\star$ be 0.

We can also take a second look at the SVM primal problem, which is

$$\min_{\boldsymbol{w},b} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{n}\xi_i\,, \tag{6}$$

where

$$\xi_i = (1 - y_i f(\boldsymbol{x}_i))_+ = \big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)_+ \tag{7}$$

is a function of $\boldsymbol{w}$.

Because $\xi_i \ge 0$, we have $\sum_{i=1}^n \xi_i = \|\boldsymbol{\xi}\|_1$, where $\boldsymbol{\xi} = (\xi_1, \xi_2, \ldots, \xi_m)^T$. Hence, the ℓ_1 regularization is also in effect in SVM learning. And, the optimal solution is indeed sparse—$\xi_i \neq 0$ means $\boldsymbol{x}_i$ is a support vector, and support vectors are sparse!

[2] Soft thresholding is an essential part of FISTA, an efficient sparse learning solver. We will discuss more on FISTA in the exercise problems.

In this SVM example, we observe that instead of directly using $\|\boldsymbol{x}\|_1$ as a regularizer, we can use $\|h(\boldsymbol{x})\|_1$, in which $h(\boldsymbol{x})$ is a function of $\boldsymbol{x}$. Another example of enforcing sparsity constraint on transformed variables is the structured sparsity. Suppose X is an $m \times n$ matrix,

$$s_i = \sqrt{\sum_{j=1}^{n} x_{ij}^2}, \quad 1 \leq i \leq m$$

is the ℓ_2 norm of its i-th row, and

$$\boldsymbol{s} = (s_1, s_2, \ldots, s_m)^T .$$

Then, the regularizer $\|\boldsymbol{s}\|_1$ makes $\boldsymbol{s}$ sparse. That is, many rows in the optimal X will be all zeros, because $s_i = 0$ forces $x_{ij} = 0$ for $1 \leq j \leq n$! If we use $\boldsymbol{x}_{i:}$ to denote the i-th row, the mixed $\ell_{2,1}$ norm of the matrix X is

$$\sum_{i=1}^{m} \|\boldsymbol{x}_{i:}\| , \tag{8}$$

and a mixed $\ell_{2,1}$ regularization will encourage many rows of X to be $\mathbf{0}^T$. More generally, the mixed $\ell_{\alpha,\beta}$ norm of X is

$$\left(\sum_{i=1}^{m} \|\boldsymbol{x}_{i:}\|_\alpha^\beta \right)^{\frac{1}{\beta}} , \tag{9}$$

in which $\alpha \geq 1, \beta \geq 1$. We can also use the mixed $\ell_{\infty,1}$ norm to replace the $\ell_{2,1}$ norm.

If we know one row in X contains grouped variables that change simultaneously, the $\ell_{2,1}$ regularizer encourages sparsity in terms of the groups of variables rather than individual variables. Similarly, the regularizer

$$\sum_{j=1}^{n} \|\boldsymbol{x}_{:j}\|_2$$

is useful when every column is a group, in which $\boldsymbol{x}_{:j}$ is the j-th column. In addition to the row and column structures in a matrix, one can arbitrarily define structures in a vector or matrix by grouping variables together. These generalized sparsity constraints are called the group sparsity or structured sparsity.

1.3 Using an overcomplete dictionary

Sparse machine learning covers diverse topics. In this section, we introduce the formulation of the dictionary learning and sparse coding problem, which is also called compressive sensing or compressed sensing.

Consider the case where an input vector $\boldsymbol{x}$ is approximately connected to its new representation $\boldsymbol{\alpha}$ by a linear relationship. That is,

$$\boldsymbol{x} \approx D\boldsymbol{\alpha}\,, \tag{10}$$

in which $\boldsymbol{x} \in \mathbb{R}^p$, $D \in \mathbb{R}^p \times \mathbb{R}^k$, and $\boldsymbol{\alpha} \in \mathbb{R}^k$. If we denote D in a block matrix form, it can be written as

$$D = [\boldsymbol{d}_1|\boldsymbol{d}_2|\ldots|\boldsymbol{d}_k]\,,$$

where $\boldsymbol{d}_i$ is the i-th column in D, which has the same length as $\boldsymbol{x}$. Then, we have

$$\boldsymbol{x} \approx \sum_{i=1}^{k} \alpha_i \boldsymbol{d}_i\,,$$

in which α_i is the i-th element in $\boldsymbol{\alpha}$. In other words, $\boldsymbol{x}$ is approximated by a linear combination of the columns of D.

Given a training dataset $\{\boldsymbol{x}_i\}_{i=1}^n$, we can organize all training examples in a $p \times n$ matrix, as

$$X = [\boldsymbol{x}_1|\boldsymbol{x}_2|\ldots|\boldsymbol{x}_n]\,.$$

We denote the new representation for $\boldsymbol{x}_i$ as $\boldsymbol{\alpha}_i$, and they also form a matrix

$$A = [\boldsymbol{\alpha}_1|\boldsymbol{\alpha}_2|\ldots|\boldsymbol{\alpha}_n] \in \mathbb{R}^k \times \mathbb{R}^n\,.$$

The errors in this linear approximation for all n examples form a matrix $X - DA$ (with size $p \times n$).

To measure the approximation error, we can compute

$$\sum_{i=1}^{p}\sum_{j=1}^{n}[X - DA]_{ij}^2\,,$$

which can be written as $\|X - DA\|_F^2$. $\|X\|_F$ is the Frobenius norm of a matrix—for an $m \times n$ matrix X,

$$\|X\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2} = \sqrt{\mathrm{tr}(XX^T)}\,.$$

If D is known and we want the reconstruction to be sparse, i.e., $\boldsymbol{\alpha}_i$ is sparse, we can solve the following problem

$$\min_{\boldsymbol{\alpha}_i} \|\boldsymbol{x}_i - D\boldsymbol{\alpha}_i\|^2 + \lambda\|\boldsymbol{\alpha}_i\|_1\,.$$

However, in many applications D is unknown and it has to be learned using the data X.

D is called the dictionary in this context, i.e., we approximate any example using a weighted combination of dictionary items.[3] When $p > k$, there are

[3]Because D is used to denote the dictionary, we do not use D or d to denote the feature dimensionality in this section. p is used for this purpose.

more feature dimensions than dictionary items, and the dictionary is called undercomplete, because it is not enough to fully describe $\boldsymbol{x}_i$ if X is full-rank. For example, the E_d matrix in PCA is undercomplete and is not a suitable dictionary.

An overcomplete dictionary (i.e., $p < k$) is often used. For example, let X consists of frontal face images (stretched to a long vector using the "vec" operation) of 1000 different individuals and each individual has 100 images taken under different poses, illumination conditions, and facial expressions; hence, $n = 100,000$. If we use X itself as the dictionary (i.e., $D = X$, which is used in many face recognition sparse learning methods), we have $k = 100,000$. Suppose the face images are resized to 100×100 for recognition, it has 10,000 pixels, and an image is stretched to a vector with $p = 10,000$. The dictionary is then an overcomplete one.

During face recognition, we are presented with a testing example $\boldsymbol{x}$, which is taken from individual `id`, under pose `p`, illumination condition `i` and expression `e`. It is natural to guess that $\boldsymbol{x}$ can be approximated by a sparse linear combination of images from the same identity, pose, illumination and expression in X, and this guess has been proven by face recognition experiments! In an extreme situation, if $\boldsymbol{x}$ is in the training set X (and hence in the dictionary D), its corresponding $\boldsymbol{\alpha}$ just need one nonzero entry (itself), and the reconstruction error is 0.

In other words, if we enforce the sparsity on the reconstruction coefficient $\boldsymbol{\alpha}$ for $\boldsymbol{x}$, we expect only a few nonzero entries in α_i, and further expect the identity of these nonzero dictionary items (also training images because $D = X$ in this application) to be the same as $\boldsymbol{x}$. Hence, face recognition might, for example, be done by a voting of these nonzero entries. That is, sparse machine learning is useful in both learning representations and recognition.

In many applications, we cannot simply set $D = X$, but has to learn a good dictionary. The dictionary learning problem can be formulated as

$$\min_{D,A} \quad \sum_{i=1}^{n} \left(\|\boldsymbol{x}_i - D\boldsymbol{\alpha}_i\|_F^2 + \lambda \|\boldsymbol{\alpha}_i\|_1 \right) \tag{11}$$

$$\text{s.t.} \quad \|\boldsymbol{d}_j\| \leq 1, \forall\, 1 \leq j \leq k\,. \tag{12}$$

As in PCA, we cannot let the dictionary item go unbounded, hence the constraint $\|\boldsymbol{d}_j\| \leq 1$ (or $\|\boldsymbol{d}_j\| = 1$) is added. It is also worthwhile to note that although the optimal $\boldsymbol{\alpha}_i$ can be found for different i independently when D is fixed, the dictionary D has to been learned using all training examples altogether. Let $\text{vec}(A)$ be the result of vectorizing the matrix A, i.e.,

$$[\text{vec}(A)]_{i \times n + j} = A_{ij} \in \mathbb{R}^{kn}\,,$$

we can also write this optimization as

$$\min_{D,A} \quad \|X - DA\|_F^2 + \lambda \|\text{vec}(A)\|_1 \tag{13}$$

$$\text{s.t.} \quad \|\boldsymbol{d}_j\| \leq 1, \forall\, 1 \leq j \leq k\,. \tag{14}$$

Dictionary learning is difficult because its objective is non-convex. We will not introduce its optimization process in detail. However, ample sparse learning packages are publicly available.

1.4 A few other related topics

A problem whose formulation is simpler than sparse coding is the Lasso (Least absolute shrinkage and selection operator) problem. The Lasso is a linear regression model, but requires the regression parameters to be sparse. Given the training dataset $\{\boldsymbol{x}_i\}_{i=1}^n$ with $\boldsymbol{x}_i \in \mathbb{R}^p$ and organized into a matrix $X = [\boldsymbol{x}_1|\boldsymbol{x}_2|\dots|\boldsymbol{x}_n]$, the regression target values $\{y_i\}_{i=1}^n$ with $y_i \in \mathbb{R}$ and organized into a vector $\boldsymbol{y}$, Lasso wants to learn regression parameters $\boldsymbol{\beta}$ such that $X\boldsymbol{\beta}$ approximates $\boldsymbol{y}$, and $\boldsymbol{\beta}$ is sparse. This objective is described as the following mathematical optimization problem:

$$\min_{\boldsymbol{\beta}} \frac{1}{n}\|\boldsymbol{y} - X^T\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1 \,, \tag{15}$$

in which λ is a hyperparameter that determines the tradeoff between approximation quality and sparsity.

The LARS (Least-Angle Regression) solver for Lasso generates a complete regularization path, i.e., optimization results for a set of λ values in the reasonable range. The regularization path facilitates the choice of the hyperparameter λ.

A sparse linear learner (e.g., Lasso) has an additional benefit: if a dimension in the linear boundary $\boldsymbol{w}$ is zero (nonzero), we can interpret this fact as this feature is useless (useful). Hence, a sparse linear classifier can also be used as a feature selection mechanism. For example, we can add the sparse inducing regularizer $\|\boldsymbol{w}\|_1$ to logistic regression or linear SVM.

Previously, we have established that we can write the binary SVM problem as an unconstrained optimization problem

$$\min_{\boldsymbol{w},b} \ \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^n \big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)_+ \,. \tag{16}$$

Note that $y_i \in \{+1, -1\}$. We can replace $\boldsymbol{w}^T\boldsymbol{w} = \|\boldsymbol{w}\|^2$ by $\|\boldsymbol{w}\|_1$, which leads to the sparse support vector classification:

$$\min_{\boldsymbol{w},b} \ \|\boldsymbol{w}\|_1 + C\sum_{i=1}^n \big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)_+^2 \,. \tag{17}$$

Note that the sparse support vector classification has changed the empirical loss from $\big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)_+$ to $\big(1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\big)_+^2$. This sparse linear classifier formulation can be used as a feature selection tool.

We have used $y_i \in \{0, 1\}$ to denote the labels in logistic regression. If we use $y_i \in \{+1, -1\}$, then logistic regression estimates the probability

$$\Pr(y_i = 1|\boldsymbol{x}_i) \approx \frac{1}{1 + \exp(-\boldsymbol{x}^T\boldsymbol{w})}$$

and

$$\Pr(y_i = -1|\boldsymbol{x}_i) \approx 1 - \frac{1}{1+\exp(-\boldsymbol{x}^T\boldsymbol{w})} = \frac{\exp(-\boldsymbol{x}^T\boldsymbol{w})}{1+\exp(-\boldsymbol{x}^T\boldsymbol{w})}\,.$$

When $y_i = 1$, its ($\boldsymbol{x}_i$'s) negative log-likelihood is

$$\ln(1+\exp(-\boldsymbol{x}^T\boldsymbol{w}))\,,$$

which approximates $-\ln(\Pr(y_i = 1|\boldsymbol{x}_i))$; when $y_i = -1$, its negative log-likelihood is

$$-\ln\left(1 - \frac{1}{1+\exp(-\boldsymbol{x}^T\boldsymbol{w})}\right) = \ln(1+\exp(\boldsymbol{x}^T\boldsymbol{w}))\,.$$

Then, the negative log-likelihood in these two cases can be unified as

$$\ln(1+\exp(-y_i\boldsymbol{x}^T\boldsymbol{w}))\,.$$

The maximum likelihood estimation for $\boldsymbol{w}$ can be achieved by minimizing the negative log-likelihood, and adding a sparse regularizer leads to the sparse logistic regression problem:

$$\min_{\boldsymbol{w}} \quad \|\boldsymbol{w}\|_1 + C\sum_{i=1}^{n} \ln\left(1+\exp(-y_i\boldsymbol{x}_i^T\boldsymbol{w})\right)\,. \tag{18}$$

Sparse logistic regression is also useful for feature selection.

A lot more can be talked about sparse learning methods. For example, in sparse PCA (cf. Section 1.1) we require that D is sparse (while sparse coding requires that A is sparse); when our data are matrices, low rank is the corresponding concept as sparsity in vectors; there are a lot of works on how to optimize sparse learning problems. However, as an introductory book, we will stop at here.

2 Dynamic time warping

From now on, we will move away from the scenario in which a feature vector is a real-valued vector and the same dimension always has the same meaning in different feature vectors. In real-world applications, the raw data are usually more complex than this simple setup. For example, temporal or spatial relationships can be important. In face recognition, even though we can stretch a face image into a vector, the pixel at a given face location, e.g., nose tip, will appear at different vector locations because of the variations in face shape, pose, expression, etc. Another example is stock price progression. We can sample the price of one particular stock in a day using a fixed time step, such that we represent the price change of this stock as a fixed length vector per day. However, it is hard to say that the same dimension in two different days' data (i.e., two days' feature vectors) mean the same thing.

Deep neural networks are very effective in dealing with these relationships. The convolutional neural network (CNN) is a very popular deep learning method

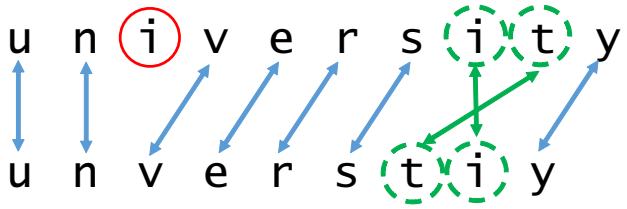


Figure 2: Alignment for `university` versus `unverstiy`.

and is to be introduced in Chapter 15. In this chapter, we mainly discuss a type of data that can be aligned, i.e., after the alignment operation, the same dimension in any two vectors do mean (or, approximately mean) the same thing. After the alignment, we can compute the similarity or distance between two examples as usual, and can apply our favorite learning and recognition methods.

Sometimes images require alignments. For example, if we want to compute the similarity or distance between two face images, they should be aligned frontal faces. If one face is slightly turned to the left, and another has the head lifted, directly calculating distances between pixels at the same location is problematic—we may compare eyebrow pixels with forehead pixels, or cheek pixels with nose pixels. Alignment, however, usually requires domain knowledge about the data and different alignment techniques are required for different types of data.

We mainly discuss the alignment of sequential data. Note that not all sequential data can be aligned, e.g., it seems very difficult (if not impossible) to align the stock price data.

2.1 Misaligned sequential data

By sequential data, we mean the type of data which comprises of an array of *ordered* elements. In a linear SVM classifier, if we rearrange the dimensions of every example and the classification boundary in the same way, the classification result will not change. For example, if we denote $\boldsymbol{x} = (x_1, x_2, x_3, x_4)^T$, $\boldsymbol{w} = (w_1, w_2, w_3, w_4)^T$, $\boldsymbol{x}' = (x_3, x_1, x_4, x_2)^T$, and $\boldsymbol{w}' = (w_3, w_1, w_4, w_2)^T$, we will always have $\boldsymbol{x}^T\boldsymbol{w} = \boldsymbol{x}'^T\boldsymbol{w}'$. Hence, the feature dimensions for SVM (and nearest neighbor, logistic regression, etc.) are unordered.

In the real world, many data are ordered. For example, an English word `university` is different from the word `unverstiy`, and the latter one is in fact a typo. This typo (or, misalignment) is easy to be fixed (or, aligned), and our brain can immediately produce the following alignment in Figure 2:

We use red and green circles to denote mismatches of characters, i.e., misalignments in Figure 2. If an alignment can be established between the typo `unverstiy` and any other word, e.g., `universe`, we can use the number of misalignments as a measure of distance. For this example, the word `university` is

the one with the smallest distance, and a spell checker software can then remind the writer to correct this typo.

The dynamic time warping (DTW) algorithm to be introduced in this section is a good tool to handle such misalignments: when the few misalignments are corrected by DTW, we can find a one-to-one correspondence between two sequences, and can use usual distance metrics or similarity measures to compare the aligned sequences. For example, the discrete metric is used in the above spell checker example, which returns 1 if two characters are different and 0 if they are the same.

Before we introduce the details of DTW, we want to note that comparing two text sequences (i.e., strings) have wider applications than building a spell checker. For example, string matching is very important in bioinformatics research.

The structure of DNA comprises of four bases: thymine (T), adenine (A), cytosine (C), and guanine (G), and DNA sequencing determines their ordering, e.g., being sequenced as ATGACGTAAATG$\cdots$. String matching is important in DNA sequencing and its applications such as molecular biology and medicine. Many algorithms and systems (e.g., parallel design and implementation of these algorithms) have been produced for string matching.

2.2 The idea (or, criteria)

DTW is a simple algorithm if one understands the dynamic programming strategy. What we want to emphasize in this chapter is, however, the way the method is formulated and solved.

One major scenario where DTW is useful is when speed is involved in generating the sequences. Two people can finish a golf swing at different speeds, hence using different time. The same action will result in video clips of different lengths, meaning the input to action recognition is a sequence of frames with variable length. The same also happens in speech recognition, because human speaks at different speeds. However, we expect the same action or same speech to be similar even if they are produced by different people, i.e., they *can be aligned*.

Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$. A few things are worth mentioning about these notations. First, sequences do not necessarily have the same length, i.e., $n \neq m$ is possible. Second, x_i $(1 \leq i \leq n)$ or y_j $(1 \leq j \leq m)$ can be a categorical value, a real value, an array, a matrix, or even more complex data types. For example, if $\boldsymbol{x}$ is a video clip, x_i is a video frame, which is a matrix of pixel values (and a pixel can be a single value or a RGB triplet). Third, we assume there is a distance metric that compares any x_i and y_j as $d(x_i, y_j)$. Hence, after $\boldsymbol{x}$ and $\boldsymbol{y}$ are aligned, we can compute the distance between them as the sum of distances of all matched elements in $\boldsymbol{x}$ and $\boldsymbol{y}$. For example, $d(x_i, y_j)$ can be the sum of Euclidean distances between all matched pixel pairs if x_i and y_j are frames in two videos.

What is a good match? Based on Figure 2, the following assumptions can be made:

1 If x_i is matched to y_j, then $d(x_i, y_j)$ should be small;

2 Some elements in $\boldsymbol{x}$ and/or $\boldsymbol{y}$ can be skipped in the matching process, e.g., the character `i` in the red circle in Figure 2; and,

3 We shall choose the match that leads to the smallest total distance.

However, these criteria are not suitable for an optimization. First, if every element is skipped, the total distance is zero, but this matching is obviously non-optimal. The DTW remedy to this issue is to request that every element in $\boldsymbol{x}$ ($\boldsymbol{y}$) has a matched element in $\boldsymbol{y}$ ($\boldsymbol{x}$). Second, if two pairs of matched elements are out of order, the optimization will have a large search space and hence be very difficult. For example, if $x_i \leftrightarrow y_j$ (where $\leftrightarrow$ means a matched pair) and $x_{i+1} \leftrightarrow y_k$, but $j > k$, then they lead to an out-of-order match (as the two green lines shown in Figure 2). In order to avoid this difficulty, DTW forbids out of order matched elements. In short, DTW uses the following two criteria to replace the second criterion in the above:

2.1 Every x_i $(1 \leq i \leq n)$ and y_j $(1 \leq j \leq m)$ has to have a matched element;

2.2 The matching has to be in order, i.e., if $x_i \leftrightarrow y_j$ and $x_{i+1} \leftrightarrow y_k$, then $j \leq k$;

Note that we use $j \leq k$ instead of $j < k$ in criterion 2.2, because when $m \neq n$ there must be an element matched to more than one elements.

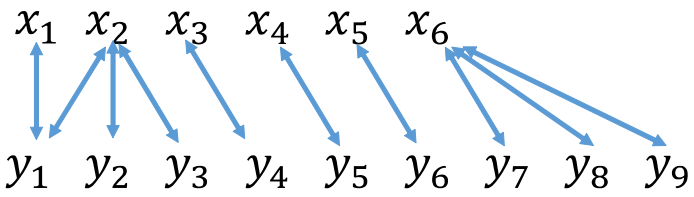
2.3 Visualization and formalization

Visualization is a good tool to intuitively understand these criteria. In the visualization in Figure 3, the matching in Figure 3a is translated into the blue path in Figure 3b.

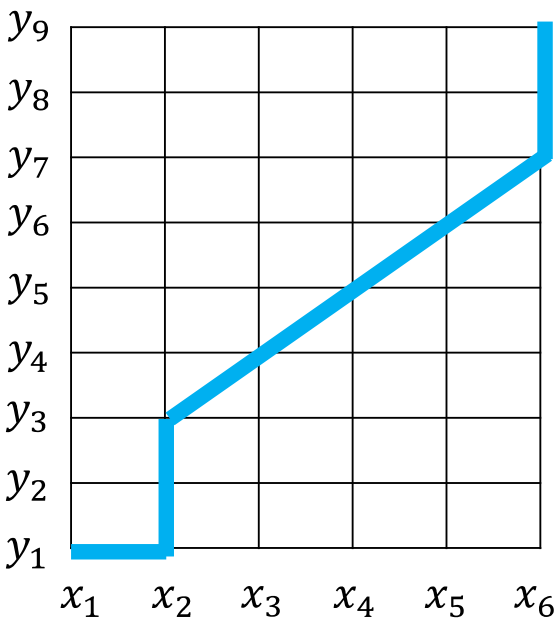
In fact, we can translate the criteria for the match into requirements for the path. A path is fully described by a series of coordinates (r_k, t_k) $(k = 1, 2, \ldots, K)$. Then,

- "Every element is in the match" is partly translated into: the first coordinate is $(1, 1)$, i.e., $r_1 = 1$ and $t_1 = 1$; and, the last coordinate is (n, m), i.e., $r_K = n$ and $t_K = m$.
- "Matching is in order" plus "Every element is in the match" are translated to a constraint between (r_k, t_k) and (r_{k+1}, t_{k+1}). There are only 3 possible cases:

$$\#1:\ r_{k+1} = r_k, \quad t_{k+1} = t_k + 1 \quad \text{(The path moves upward)}; \tag{19}$$
$$\#2:\ r_{k+1} = r_k + 1, t_{k+1} = t_k \quad \text{(The path moves rightward)}; \tag{20}$$
$$\#3:\ r_{k+1} = r_k + 1, t_{k+1} = t_k + 1 \quad \text{(The path moves upright)}; \tag{21}$$



(a) The match



(b) The path

Figure 3: Visualizing a match between two sequences as a path.

- The translation of "Choose the path with the smallest total distance" requires a little more notations. We denote $\boldsymbol{r} = (r_1, r_2, \ldots, r_K)$ and $\boldsymbol{t} = (t_1, t_2, \ldots, t_K)$, and a path is denoted as $(\boldsymbol{r}, \boldsymbol{t})$. Let Ω be the set of all paths that satisfy these translated criteria, then to minimize the total matching distance is translated as

$$D(n, m) = \min_{(\boldsymbol{r},\boldsymbol{t})\in\Omega} \sum_{k=1}^{K_{(\boldsymbol{r},\boldsymbol{t})}} d(x_{r_k}, y_{t_k}) \,. \tag{22}$$

  Note that for different paths, K may have different values, hence we use the notation $K_{(\boldsymbol{r},\boldsymbol{t})}$. $D(n, m)$ is the smallest total matching distance.

2.4 Dynamic programming

The number of paths that satisfy these constraints is enormous. If we only allow moving rightward or upward, as illustrated in Figure 4a, the number of paths is

$$\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1},$$

the proof of which we leave as an exercise. When n or m is not too small (e.g., when $n \approx m$), this number increases exponentially with n (or m). Hence, it is not feasible to enumerate all paths and find the best one.

The *divide and conquer* strategy is useful in reducing the complexity for DTW. As illustrated in Figure 4a, if an oracle tells us that node $A = (x_2, y_3)$ is in the optimal path, then the optimal total distance is the sum of distances of the optimal path from O to A and the optimal path from A to T. The original large problem is split into two smaller ones, and smaller problems are usually easier to solve than a big one. For example, in the Fast Fourier Transform (FFT) algorithm, a size n problem is divided into two smaller problems with size $\frac{n}{2}$, whose solutions can be combined to solve the original problem. In DTW, we can use this strategy, too.

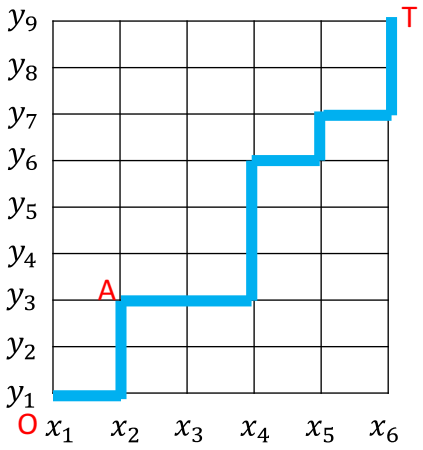
There are, however, two issues remaining for solving DTW using divide-and-conquer. First, we do not have an oracle to tell us "A is in the optimal path (or not in the optimal path)". Second, it is not clear how we should divide a large problem into smaller ones. The *dynamic programming* strategy solves the first issue by *enumerating all candidates for* A, and the second issue (usually) by splitting a large problem into a very small problem (which is trivial to solve) and another one.

In DTW, if we want to compute the optimal $D(n, m)$, there are three candidates for the intermediate point A when the path from A to T only contains one movement:

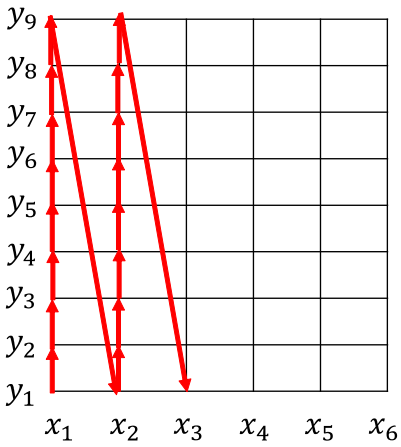
- $A = (x_{n-1}, y_{m-1})$ and the movement is upright (i.e., diagonal). Then, we have

$$D(n, m) = D(n-1, m-1) + d(x_n, y_m) \,,$$

  in which $D(n-1, m-1)$ is the optimal distance from O to (x_{n-1}, y_{m-1}).



(a) Divide and Conquer



(b) Dynamic Programming

Figure 4: The dynamic programming strategy for DTW.

- $A = (x_n, y_{m-1})$ and the movement is upward. Then, we have

$$D(n, m) = D(n, m-1) + d(x_n, y_m)\,.$$

- $A = (x_{n-1}, y_m)$ and the movement is rightward. Then, we have

$$D(n, m) = D(n-1, m) + d(x_n, y_m)\,.$$

To combine these three cases, we just need to find the smallest value among $D(n-1, m-1)$, $D(n, m-1)$ and $D(n-1, m)$. In other words, for any integer

$1 < u \leq n$ and $1 < v \leq m$, we have

$$D(u,v) = d(x_u, y_v) + \min\left\{D(u-1,v), D(u,v-1), D(u-1,v-1)\right\} . \quad (23)$$

Equation 23 is a *recursive* relationship between a large problem and two smaller problems. One smaller problem is trivial to solve ($d(x_u, y_v)$), and there are three candidates for the other smaller problem. This kind of recursive relationship is the key in all dynamic programming methods. In order to find $D(n,m)$, we need to find three other optimal values $D(n-1,m-1)$, $D(n,m-1)$ and $D(n-1,m)$.

To find $D(n-1,m-1)$, we need to find three more optimal values $D(n-2,m-2)$, $D(n-1,m-2)$ and $D(n-2,m-1)$. And $D(n-1,m-2)$, $D(n,m-2)$, $D(n-1,m-1)$ are requested to find $D(n,m-1)$. We have the following observations for this recursive expansion:

- We can write a simple recursive program to compute $D(n,m)$, e.g., in C++, as

```
double D(int n, int m, const double **dist)
{
  if (m == 1 && n == 1) return dist[1][1];
  if (n == 1)  return dist[n][m] + D(n, m-1, dist);
  if (m == 1)  return dist[n][m] + D(n-1, m, dist);

  return dist[n][m] +
         min( min(D(n-1, m-1, dist), D(n, m-1, dist)),
              D(n-1, m, dist));
}
```

  in which `dist` is a two dimensional array, and `dist(i,j)` $= d(x_i, y_j)$. The number of function evaluations is increasing exponentially with n or m. We can build an expansion tree to illustrate the recursive computation, as illustrated in Figure 5. It is obvious that many computations are *redundant*, e.g., $D(n-1,m-1)$ are repeated 3 times, and $D(n-1,m-2)$ and $D(n-2,m-1)$ two times in the partial expansion of Figure 5.

- To compute $D(n,m)$, we need to compute all $D(u,v)$ for $1 \leq u \leq n$ and $1 \leq v \leq m$ in the expansion tree. There are in total nm optimal values to compute, including $D(n,m)$.

In fact, we just need to compute nm such $d(\cdot,\cdot)$ values if a proper evaluation order is designed. A good evaluation order should satisfy that $D(u-1,v-1)$, $D(u,v-1)$ and $D(u-1,v)$ are already computed when we compute $D(u,v)$. Figure 4b specifies such an order: start from $(1,1)$ and move upward until $(1,m)$; then move to the next column $(2,1)$ and finish the second column till $(2,m)$; then move to the next column and so on. Finally, when the last column is traversed, we have computed $D(n,m)$ successfully. Now, we have all components of the DTW algorithm, which is described in Algorithm 1. Note that $D(i,j)$ in Algorithm 1 is the (i,j)-th element in the two dimensional array D, which is no longer a function call.

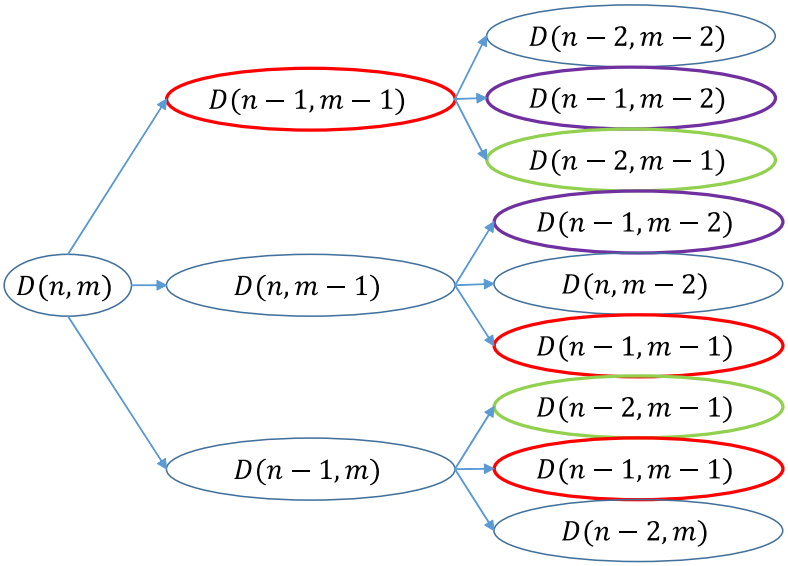


Figure 5: The (partial) expansion tree for recursive DTW computations.

It is obvious that $d(x_i, y_j)$ is evaluated nm times in Algorithm 1, which is more efficient than the recursive code. The reason is that all redundant computations have been eliminated by a proper evaluation order. Algorithm 1 evaluates $D(u, v)$ in a column-based ordering. Other orderings are also possible. For example, we can evaluate the first row, then the second row, until $D(n, m)$ is computed. Note that with this ordering, when we need to compute $D(i, j)$, the values in $D(x, y)$ $(1 \leq x \leq i,\ 1 \leq y \leq j)$ have all been computed.

In Algorithm 1 there are two special cases: the first column $D(1, j)$ and the first row $D(i, 1)$. We can pad a column to the left of the first column, and pad a row under the first row with appropriate values (0 or ∞). Then, we can use Equation 23 to process all $D(u, v)$ in a unified way.

One frequently appearing characteristic of dynamic programming is that: instead of computing one single optimal value (e.g., $D(n, m)$), dynamic programming methods compute the optimal values of many more problems (e.g., $D(u, v)$ for $1 \leq u \leq n$ and $1 \leq v \leq m$). These additional problems, however, lead to efficient solution of the original problem.

Dynamic programming has a wide range of applications, and the recursive relationships may be much more complex than Equation 23. The way to split a large problem into smaller ones has a lot of variations, and the number of smaller problems can be two, three, or even more. The order of evaluation does not necessarily start from the smallest problem. In more complex problems, the recursive relationship may involve two (or even more) sets of variables. We will see more examples of dynamic programming in the hidden Markov model (HMM), a tool to handle another type of non-vector data. The basics of HMM

Algorithm 1 The Dynamic Time Warping Method

Input: Two sequences $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$
$D(1,1) = d(x_1, y_1)$
for $j = 2$ to m **do**
  $D(1,j) = d(x_1, y_j) + D(1, j-1)$
end for
for $i = 2$ to n **do**
  $D(i,1) = d(x_i, y_1) + D(i-1, 1)$
  for $j = 2$ to m **do**
    $D(i,j) = d(x_i, y_j) + \min\{D(i-1, j-1), D(i, j-1), D(i-1, j)\}$
  end for
end for

will be introduced in the next chapter.

Exercises

1. (Soft thresholding) Let $\lambda > 0$. Show that the solution for

$$\arg\min_{\boldsymbol{x}} \|\boldsymbol{x} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{x}\|_1 \tag{24}$$

is the soft thresholding strategy applied to every dimension of $\boldsymbol{y}$ with a shrinkage parameter $\frac{\lambda}{2}$. That is,

$$\boldsymbol{x}^\star = \operatorname{sign}(\boldsymbol{y}) \left(|\boldsymbol{y}| - \frac{\lambda}{2} \right)_+ , \tag{25}$$

in which the sign function, the absolute value function, the minus operation, the $(\cdot)_+$ thresholding function, and the multiplication are all applied element-wise. If we denote the shrinkage-thresholding operator as

$$\mathcal{T}_\lambda(\boldsymbol{x}) = \operatorname{sign}(\boldsymbol{x}) \left(|\boldsymbol{x}| - \lambda \right)_+ , \tag{26}$$

the solution can be represented as $\boldsymbol{x}^\star = \mathcal{T}_{\frac{\lambda}{2}}(\boldsymbol{y})$.

2. (ISTA) ISTA, or *iterative shrinkage-thresholding algorithms* are a family of methods, which can solve the following problem when the dictionary D and $\boldsymbol{x}$ are known:

$$\arg\min_{\boldsymbol{\alpha}} \|\boldsymbol{x} - D\boldsymbol{\alpha}\|^2 + \lambda\|\boldsymbol{\alpha}\|_1 .$$

Let $f(\boldsymbol{\alpha})$ and $g(\boldsymbol{\alpha})$ be two functions of $\boldsymbol{\alpha}$, in which f is a smooth convex function, g is a continuous convex function. However, g is not necessarily smooth, which makes the optimization of $\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) + g(\boldsymbol{\alpha})$ difficult.

ISTA solves $\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) + g(\boldsymbol{\alpha})$ in an iterative manner, and each iteration is a shrinkage-thresholding step. Hence, it is named as the iterative shrinkage-thresholding algorithm (ISTA). In this problem, we only consider the simple case of $f(\boldsymbol{\alpha}) = \|\boldsymbol{x} - D\boldsymbol{\alpha}\|^2$ and $g(\boldsymbol{\alpha}) = \lambda\|\boldsymbol{\alpha}\|_1$ ($\lambda > 0$).

(a) One additional requirement in ISTA is that f is continuously differentiable with Lipschitz continuous gradient $L(f)$, i.e., there exists an f-dependent constant $L(f)$, such that for any $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$

$$\|\nabla f(\boldsymbol{\alpha}_1) - \nabla f(\boldsymbol{\alpha}_2)\| \leq L(f)\|\boldsymbol{\alpha}_1 - \boldsymbol{\alpha}_2\| , \tag{27}$$

in which ∇f is the gradient of f.

For our choice of f, show that its corresponding $L(f)$ (or simply abbreviated as L) is two times the largest eigenvalue of $D^T D$, which is called the *Lipschitz constant* of ∇f.

(b) ISTA first initializes $\boldsymbol{\alpha}$ (e.g., by ignoring $g(\boldsymbol{\alpha})$, and solving the ordinary least square regression). Then, in each iteration, the following problem is solved:

$$p_L(\boldsymbol{\beta}) \stackrel{\text{def}}{=} \arg\min_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}) + \frac{L}{2} \left\| \boldsymbol{\alpha} - \left(\boldsymbol{\beta} - \frac{1}{L}\nabla f(\boldsymbol{\beta}) \right) \right\|^2 , \tag{28}$$

Algorithm 2 FISTA: A Fast iterative shrinkage-thresholding algorithm

1: {This version has a constant step size, i.e., when the Lipschitz constant $L = \nabla f$ is known}
2: {$\boldsymbol{\alpha}_0$ is the initial value for $\boldsymbol{\alpha}$, which can be obtained, e.g., through OLS.}
3: **Initialization**: $t \leftarrow 1$, $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha}_0$, $\boldsymbol{\beta} \leftarrow \boldsymbol{\alpha}$
4: **Iterate**: Repeat the following steps until convergence.

$$\boldsymbol{\alpha}' \leftarrow p_L(\boldsymbol{\beta})\,, \tag{29}$$

$$t' \leftarrow \frac{1+\sqrt{1+4t^2}}{2}\,, \tag{30}$$

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\alpha}' + \left(\frac{t-1}{t'}\right)(\boldsymbol{\alpha}' - \boldsymbol{\alpha})\,, \tag{31}$$

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha}'\,, \tag{32}$$

$$t \leftarrow t'\,. \tag{33}$$

in which L is the Lipschitz constant and $\boldsymbol{\beta}$ is a parameter. In the t-th iteration, ISTA updates the solution by

$$\boldsymbol{\alpha}_{t+1} = p_L(\boldsymbol{\alpha}_t)\,.$$

For our choice of f and g, solve this optimization problem. Explain why sparsity is induced by every step in ISTA.

3. (FISTA) FISTA, which stands for fast ISTA, is an improved ISTA method. The FISTA method is shown as pseudo-code in Algorithm 2. In terms of algorithmic details, the difference between FISTA and ISTA is the introduction of an intermediate variable $\boldsymbol{\beta}$ and a variable t that controls the update of $\boldsymbol{\alpha}$. These simple changes, however, greatly improves the convergence speed of ISTA (hence the name FISTA).

   Use the following sample Matlab / Octave code to generate a dictionary D with 300 items (each being 150 dimensional). One example $\boldsymbol{x}$ is generated by a linear combination of 40 dictionary items (out of 300, hence is sparse). Given a noise contaminated example $\boldsymbol{x}$, we will use the simple FISTA algorithm in Algorithm 2 to find an $\boldsymbol{\alpha}$, such that $\|\boldsymbol{x} - D\boldsymbol{\alpha}\|^2 + \lambda\|\boldsymbol{\alpha}\|_1$ is minimized. We use $\lambda = 1$.

```
p = 150; % dimensions
k = 300; % dictionary size
D = randn(p,k); % dictionary
% normalize dictionary item
for i=1:k
    D(:,i) = D(:,i)/norm(D(:,i));
end
% x is a linear reconstruction of 40 dictionary items
truealpha = zeros(k,1);
```

```
truealpha(randperm(k,40)) = 30 * (rand(40,1)-0.5);
% add noise, and generate x
noisealpha = truealpha + .1*randn(size(truealpha));
x = D * noisealpha;
% set lambda=1
lambda = 1;
```

(a) Find the ordinary least square optimal solution (i.e., as if $\lambda = 0$). How many nonzero entries are there in the solution for $\boldsymbol{\alpha}$? Is it sparse?

(b) Write a Matlab / Octave program to implement Algorithm 2. Run 100 iterations before termination. Use the OLS solution to initialize (i.e., as $\boldsymbol{\alpha}_0$). Are the FISTA solution sparse? In the FISTA solution, how many of its nonzero entry happen to be nonzero in the true $\boldsymbol{\alpha}$ vector that is used to generate the data $\boldsymbol{x}$ (i.e., the `truealpha` variable)? Do you think FISTA gives a good solution?

(c) If you want the solution to be sparser (or denser), how will you change λ? Explain your choice and make your argument based specifically on the FISTA method.

Finally, we want to add a cautious note. The FISTA method is more complex than Algorithm 2. For example, it may be computationally infeasible to find the Lipschitz constant, and the termination criterion may be very complex. Algorithm 2 is only an illustration of the simplest possible implementation of FISTA. Albeit simple, it retains the core idea in FISTA.

4. Given an $n \times m$ grid, find the number of paths that move from the bottom-left corner to the top-right corner. A movement in the path can only move rightward or upward, and one move can only traverse one cell in the grid. An example is the blue path shown in Figure 4a, which contains 5 rightward and 8 upward movements.

5. (Integral image) The *integral image* is a data structure that has wide usage in computer vision, especially in applications which require fast processing speed, e.g., real-time face detection. It has appeared in other names in earlier literature, as the *summed area table* in the computer graphics community.

   (a) Let A be an $n \times m$ single-channel image (or equivalently, an $n \times m$ matrix). We denote the integral image of A as B, which has the same size as A. For any $1 \le i \le n$ and $1 \le j \le m$, $B(i, j)$ is defined as

$$B(i,j) = \sum_{u=1}^{i} \sum_{v=1}^{j} A(u,v)\,. \tag{34}$$

   On the left of Table 1, an example input image A is shown as a 3×3 matrix. Fill the values of B into the right side of Table 1.

Table 1: An example of the integral image. A is the input image, and B is the integral image.

$$\begin{pmatrix} 1 & 1 & -1 \\ 2 & 3 & 1 \\ 3 & 4 & 6 \end{pmatrix} \quad \Longrightarrow \quad \left(\begin{array}{c|c|c} & & \\ \hline & & \\ \hline & & \end{array}\right)$$

A $\qquad\qquad$ B

(b) Find a method to compute B from A, whose complexity is $\mathcal{O}(nm)$, i.e., linear in the number of pixels. (Hint: use dynamic programming.)

(c) The main usage of an integral image B is to find the sum of all elements in *any* rectangular image patch inside A. Let (i_1, j_1) and (i_2, j_2) be two coordinates in the image ($1 \leq i_1 < i_2 \leq n$, $1 \leq j_1 < j_2 \leq m$), which defines the top-left and bottom-right corners of a rectangle. The task is to compute the rectangular sum

$$S = \sum_{i_1 \leq u \leq i_2} \sum_{j_1 \leq v \leq j_2} A(u, v)\,.$$

Let B be the integral image of A. Show that any such rectangular sum can be computed in $\mathcal{O}(1)$ time.