Applied Linear Algebra and Big Data Course Book

A project by

Kabir Gandhi

Presented to

The Department of Applied Mathematics

In partial fulfillment of the honors requirements
for the degree of

Bachelor of Arts

Harvard College
Cambridge, Massachusetts

March 29, 2019

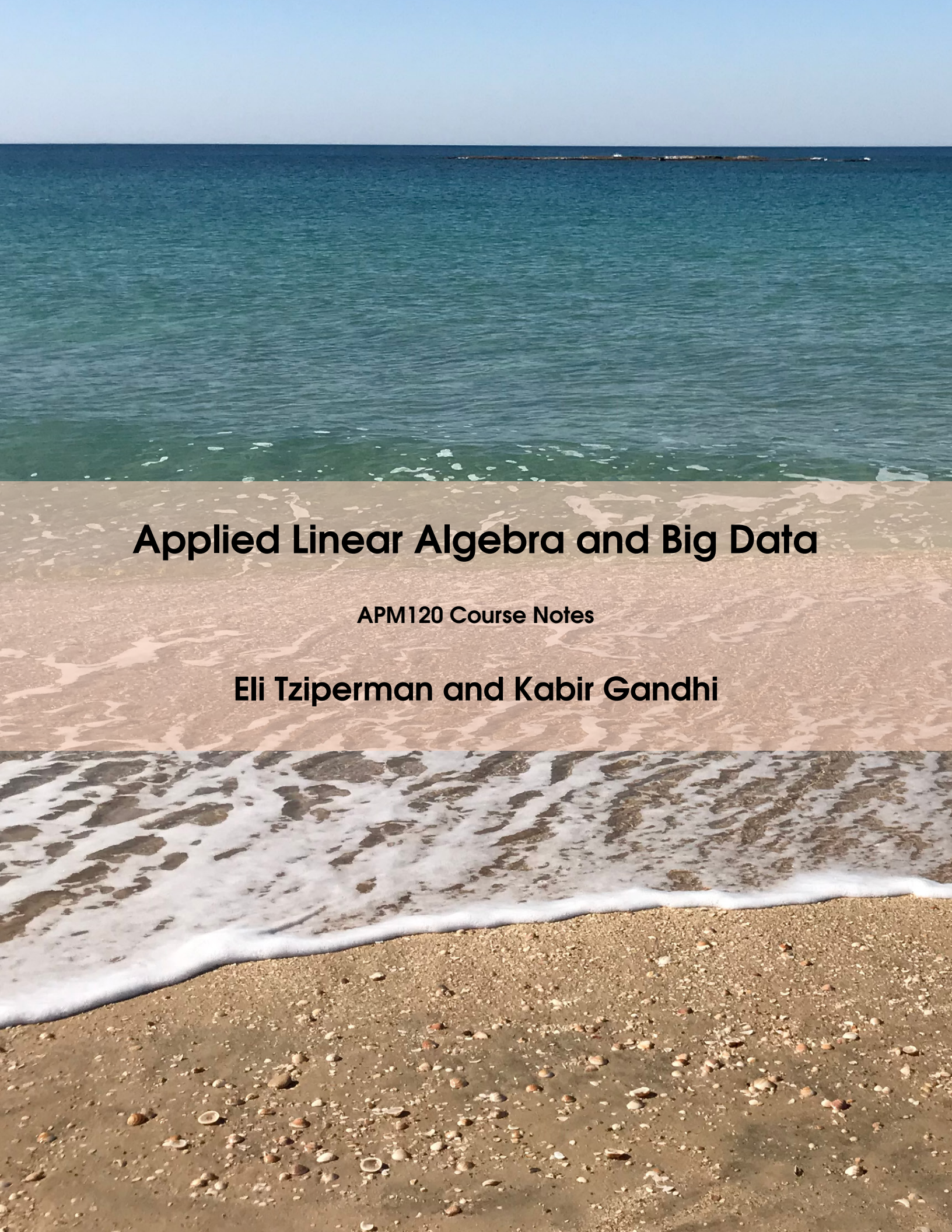
Abstract

This project involves the development of course notes for Harvard's Applied Mathematics 120: Applied Linear Algebra and Big Data. The course notes span eight chapters, beginning with a review of foundational concepts in linear algebra that are used throughout the course. Then, methods for solving linear equations are discussed, including the LU decomposition, iterative methods, the MapReduce algorithm and how to deal with large, sparse matrices that often come up in large-scale applications. Next, eigenvalues, eigenvectors and their respective applications are discussed, including Google's PageRank algorithm, spectral clustering, solutions to systems of linear ordinary differential equations, transient amplification and the Jordan form. Chapters 4 and 5 explore principal component analysis and singular value decomposition, and several applications of these techniques: image compression, the matrix norm, the condition number, polar decomposition, solving under-determined and over-determined linear equations, multivariate PCA, maximum covariance analysis, and SVD-based recommendation systems. Then, in chapter 6 we discuss the identification and analysis of frequent patterns and applications of similarity analyses. In chapter 7, multiple clustering algorithms are considered, analyzed and demonstrated, including hierarchical, k-means, and self-organizing maps. In addition to the clustering algorithms, the notes cover related issues such as unusually-shaped data (and the application of the Mahalanobis distance in these cases), the "curse of dimensionality", and techniques for clustering very large datasets such as the BFR and CURE algorithms. Finally, machine learning classification algorithms are discussed in chapter 8, including perceptrons, support vector machines and feedforward neural networks, as well as a discussion of over-fitting and neural network optimization. The notes include numerous images, graphics and numerical examples, generated using MATLAB and python, designed to clarify challenging concepts and improve the overall student experience in digesting complex course material. There was an emphasis in designing these notes to include step-by-step numerical examples for frequently arising problems. The course is designed to be predominantly applications-focused and proofs are provided only when they contribute to the understanding of important concepts.

Acknowledgements and Background

This project would not have been possible without the incredible guidance and encouragement of Professor Eli Tziperman. I approached Professor Tziperman with the idea of taking on this daunting project and have received invaluable feedback throughout the writing process. I am further grateful to the rest of the AM120 teaching staff, in particular Xiaoting Yang and Minmin Fu, for their opinions on how best to explain aspects of challenging course material. I would also like to thank all the students of AM120 past and present for their feedback on the course book and for pointing out areas of course content that required further clarification. Finally, my thanks to Mathias Legrand for his template used in designing the course notes.

Throughout this project, I applied both my previous experience as a student and teaching assistant in AM120 and experience from other courses throughout my Applied Mathematics education, in particular AM115 (Mathematical Modeling), to make the material as approachable and useful as possible. The motivation for this project emerged as a result of my work as a teaching assistant in Spring 2018. I received feedback from students in the course who wished there was a single, centralized resource for course content which motivated me to initiate this project. I constructed a full first draft of these course notes based on the extended course syllabus, my own notes, course references, and some existing partial notes, and worked with Professor Tziperman to edit and refine the notes for the current iteration of the course, with regular feedback from the outstanding students in AM120.



Applied Linear Algebra and Big Data

APM120 Course Notes

Eli Tziperman and Kabir Gandhi

Contents

Part One

1. Introduction

1.1 What is this course about?

Welcome to APM120! This course covers topics in linear algebra which arise frequently in applications, especially in the analysis of large data sets. The topics covered include linear equations, eigenvalue problems, linear differential equations, principal component analysis, singular value decomposition, data mining methods (such as frequent pattern analysis), clustering, classification, and machine learning methods (such as neural networks and support vector machines). Examples will be given from physical sciences, biology, climate, finance, the internet, image processing and more. The focus is on *applications* rather than rigorous proofs, and proofs will be provided only occasionally, when they provide important intuition about the subjects covered.

Please see the course syllabus for details regarding course administration, requirements, grading, and so on.

1.2 Notation and linear algebra review

Consider the following brief reminder and introduction to notation that will be used in the course. APM120 assumes you have already taken a basic linear algebra course. Homework #0 covers the material you need to know before starting this course, covered below.

Notation

Vectors: $\vec{x} = \mathbf{x} = x = x_i$; vector norm $|\mathbf{x}| = \sqrt{\sum_i x_i^2}$;

row vector $(\mathbf{x})_{1\times n}$; column vector $(\mathbf{x})_{n\times 1}$;

scalar product $\mathbf{x}\cdot\mathbf{y} = \mathbf{x}^T\mathbf{y} = \sum_i x_i y_i = x_i y_i = |\mathbf{x}|\,|\mathbf{y}|\cos\theta$;

Matrices: $\mathsf{A} = \underset{\sim}{\mathsf{A}} = a_{ij}$;

matrix multiplication $(\mathsf{C})_{n\times p} = (\mathsf{A})_{n\times m}(\mathsf{B})_{m\times p}$ where $c_{ik} = \sum_{j=1}^{m} a_{ij}b_{jk} = a_{ij}b_{jk}$. Matrix mul-

tiplication is *associative* but not *commutative*. This means that $(\mathsf{AB})\mathsf{C} = \mathsf{A}(\mathsf{BC}) \equiv \mathsf{ABC}$, but $\mathsf{AB} \not\equiv \mathsf{BA}$. For example,

$$\mathsf{LA} = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 \\ 4\frac{2}{3} & -5\frac{2}{3} & \frac{1}{3} \\ -3 & 6.5 & 1.5 \end{pmatrix}.$$

Note that $1/3$ of the first row of A has been added to the second and $1/2$ of the first row was subtracted from the third, an example of row manipulation via matrix multiplication.

Permutation matrices: These are used to exchange the rows of a given matrix, e.g.,

$$\mathsf{PA} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{pmatrix} = \begin{pmatrix} 4 & -6 & 0 \\ 2 & 1 & 1 \\ -2 & 7 & 2 \end{pmatrix},$$

Note that while PA exchanges rows of A, AP exchanges its columns.

Unit/identity matrix: $\mathsf{I} = \delta_{ij}$, $\mathsf{AI} = \mathsf{IA} = \mathsf{A}$;
I is a diagonal matrix with 1s along the diagonal. $\mathsf{I}^{-1} = \mathsf{I}$

Transpose: $\mathsf{A}^T = a_{ji}$, transpose of a product $(\mathsf{AB})^T = \mathsf{B}^T\mathsf{A}^T$, conjugate transpose $\mathsf{A}^\dagger = a^*_{ji}$ where $z^* = (a+ib)^* = a-ib$.

Inverse: $\mathsf{A}^{-1}\mathsf{A} = \mathsf{A}\mathsf{A}^{-1} = \mathsf{I}$.

Eigenvalues: λ_i, and the corresponding **eigenvectors**, $\mathbf{e}_i$, of a matrix A, will be further discussed in the next chapter. They satisfy the equation $\mathsf{A}\mathbf{e}_i = \lambda_i\mathbf{e}_i$, as shown in their calculation below.

Review examples

We now go through several examples of key linear algebra techniques that will proveful useful throughout the course. First, we define the following matrices that we will carry out operations on:

$$L = \begin{pmatrix} 1 & 4 & 1 \\ 4 & 2 & 8 \\ 3 & 2 & 1 \end{pmatrix} = [1,4,1;4,2,8;3,2,1];$$

$$d = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} = [1;2;4];$$

$$R = \begin{pmatrix} 1 & 4 \\ 2 & 6 \end{pmatrix} = [1,4;2,6]$$

Linear equations, Gaussian elimination and back substitution

Consider $Lx = d$. We denote the ith line of the combined matrix and RHS (L,d) as l_i, and the jth column as c_j. To solve, carry out the following Gaussian elimination steps:

$$(L,d) = \left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 4 & 2 & 8 & 2 \\ 3 & 2 & 1 & 4 \end{array}\right)$$

$$l_2 = l_2 - 4l_1$$

$$\left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & -14 & 4 & -2 \\ 3 & 2 & 1 & 4 \end{array}\right)$$

$$l_3 = l_3 - 3l_1$$

$$\left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & -14 & 4 & -2 \\ 0 & -10 & -2 & 1 \end{array}\right)$$

$$l_2 = l_2/(-14)$$

$$\left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & 1 & -2/7 & 1/7 \\ 0 & -10 & -2 & 1 \end{array}\right)$$

$$l_3 = l_3 + 10l_2$$

$$\left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & 1 & -2/7 & 1/7 \\ 0 & 0 & -4\frac{6}{7} & 2\frac{3}{7} \end{array}\right)$$

$$l_3 = l_3/(-4\frac{6}{7})$$

$$\left(\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & 1 & -2/7 & 1/7 \\ 0 & 0 & 1 & -1/2 \end{array}\right)$$

Finally, using back substitution we solve:

$$\begin{aligned} &x_3 = -1/2 \\ &x_2 - (2/7)(-1/2) = 1/7 \Rightarrow x_2 = 0; \\ &x_1 + 4\times 0 + 1(-1/2) = 1 \Rightarrow x_1 = 1.5 \end{aligned}$$

In Matlab, the inverse (or more conveniently the backslash operator) may be used: `x=inv(L)*d`, or `x=L\d`.

Determinants and linear independence of vectors

1. **Using row-expansion of cofactors:**
   The determinant of L is given by the expansion of cofactors along the ith row,

   $$|L| = \sum_{j=1}^{3} l_{ij} c_{ij}$$

   where l_{ij} are the elements of L and the cofactors are given by $c_{ij} = (-1)^{i+j}|M_{ij}|$. The matrix M_{ij} is obtained from L by eliminating the i row and j column. Expanding along $i = 1$,

   $$\begin{aligned} |L| &= 1\times(-1)^{1+1}\det([2,8;2,1]) + 4\times(-1)^{1+2}\det([4,8;3,1]) + 1\times(-1)^{1+3}\det([4,2;3,2]) \\ &= 1\times(1)\times(-14) + 4\times(-1)\times(-20) + 1\times(1)\times(2) \\ &= 68 \end{aligned}$$

   In Matlab, write `L=[1,4,1;4,2,8;3,2,1]; det(L)`.
2. **Using reduction to row-echelon form:**
   Using Gaussian elimination, zero elements under the diagonal,

   $$\begin{aligned} L &= \begin{pmatrix} 1 & 4 & 1 \\ 4 & 2 & 8 \\ 3 & 2 & 1 \end{pmatrix} \\ l_2 &= l_2 - 4l_1 \\ &= \begin{pmatrix} 1 & 4 & 1 \\ 0 & -14 & 4 \\ 3 & 2 & 1 \end{pmatrix} \\ l_3 &= l_3 - 3l_1 \\ &= \begin{pmatrix} 1 & 4 & 1 \\ 0 & -14 & 4 \\ 0 & -10 & -2 \end{pmatrix} \\ l_3 &= l_3 - (10/14)l_2 \\ &= \begin{pmatrix} 1 & 4 & 1 \\ 0 & -14 & 4 \\ 0 & 0 & -4.8571 \end{pmatrix} \end{aligned}$$

   The determinant is then the product of the diagonal elements,

   $$\det(L) = 1\times(-14)\times -4.8571 = 68.$$

Matrix inversion

1. **Using matrix of cofactors:**
   Again, the cofactors are given by $c_{ij} = (-1)^{i+j}|M_{ij}|$, where the matrix M_{ij} is obtained from L by eliminating the i row and j column. The inverse is then given by $L^{-1} = \frac{1}{|L|}C^T$. Specifically,

$$\begin{aligned} L^{-1} &= \frac{1}{|L|}\begin{pmatrix} (-1)^{1+1}(-14) & (-1)^{1+2}(-20) & (-1)^{1+3}(2) \\ (-1)^{2+1}(2) & (-1)^{2+2}(-2) & (-1)^{2+3}(-10) \\ (-1)^{3+1}(30) & (-1)^{3+2}(4) & (-1)^{3+3}(-14) \end{pmatrix}^T \\ &= \frac{1}{68}\begin{pmatrix} -14 & 20 & 2 \\ -2 & -2 & 10 \\ 30 & -4 & -14 \end{pmatrix}^T \\ &= \frac{1}{68}\begin{pmatrix} -14 & -2 & 30 \\ 20 & -2 & -4 \\ 2 & 10 & -14 \end{pmatrix} \end{aligned}$$

2. **Using row operations:**
   Start with the identity matrix augmented to the right of L:

$$\left(\begin{array}{ccc|ccc} 1 & 4 & 1 & 10 & 0 & \\ 4 & 2 & 8 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 & 0 & 1 \end{array}\right)$$

   Multiply the first line by -4 and add it to the second line:

$$\left(\begin{array}{ccc|ccc} 1 & 4 & 1 & 1 & 0 & 0 \\ 0 & -14 & 4 & -4 & 1 & 0 \\ 3 & 2 & 1 & 0 & 0 & 1 \end{array}\right)$$

   Then, similarly, multiply the first line by -3 and add it to the third line:

$$\left(\begin{array}{ccc|ccc} 1 & 4 & 1 & 1 & 0 & 0 \\ 0 & -14 & 4 & -4 & 1 & 0 \\ 0 & -10 & -2 & -3 & 0 & 1 \end{array}\right)$$

   Moving onto the second column, subtract (4/-14) times row 2 from row 1.

$$\left(\begin{array}{ccc|ccc} 1.0 & 0 & 2.1 & -0.14 & 0.29 & 0 \\ 0 & -14.0 & 4.0 & -4.0 & 1.0 & 0 \\ 0 & -10.0 & -2.0 & -3.0 & 0 & 1.0 \end{array}\right)$$

   Next, subtract (-10/-14) times row 2 from row 3:

$$\left(\begin{array}{ccc|ccc} 1.0 & 0 & 2.1 & -0.14 & 0.29 & 0 \\ 0 & -14.0 & 4.0 & -4.0 & 1.0 & 0 \\ 0 & 0 & -4.9 & -0.14 & -0.71 & 1.0 \end{array}\right)$$

   Finally, we zero out the third column by subtracting $(2.1/-4.9)$ times row three from row one and subtracting $(4/-4.9)$ times row three from row two:

$$\left(\begin{array}{ccc|ccc} 1.0 & 0 & 0 & -0.21 & -0.029 & 0.44 \\ 0 & -14.0 & 4.0 & -4.0 & 1.0 & 0 \\ 0 & 0 & -4.9 & -0.14 & -0.71 & 1.0 \end{array}\right)$$

and

$$\left(\begin{array}{ccc|ccc} 1.0 & 0 & 0 & -0.21 & -0.029 & 0.44 \\ 0 & -14.0 & 0 & -4.1 & 0.41 & 0.82 \\ 0 & 0 & -4.9 & -0.14 & -0.71 & 1.0 \end{array}\right)$$

Finally, we divide row 2 by -14 and row 3 by -4.9 to arrive at:

$$\left(\begin{array}{ccc|ccc} 1.0 & 0 & 0 & -0.21 & -0.029 & 0.44 \\ 0 & 1.0 & 0 & 0.29 & -0.029 & -0.059 \\ 0 & 0 & 1.0 & 0.029 & 0.15 & -0.21 \end{array}\right)$$

where the right hand side of the augmented matrix is the inverse. It is easy to verify that the three right columns are indeed the inverse of the original L matrix by multiplying them by L to arrive at the identity matrix (By definition, $L^{-1}L = I$).

Eigenvalues and eigenvectors

First, we find the "characteristic polynomial" of R and the roots of this polynomial (the eigenvalues of R):

$$\begin{aligned} \det(R-\lambda I) &= \det\begin{pmatrix} 1-\lambda & 4 \\ 2 & 6-\lambda \end{pmatrix} \\ &= (1-\lambda)(6-\lambda)-8 = \lambda^2-7\lambda-2 \quad \text{(the characteristic polynomial)} \\ \Rightarrow \lambda &= 7/2 \pm \sqrt{57}/2 = -0.2749, 7.2749 \end{aligned}$$

Consider two examples of calculating eigenvectors. First, a simple 2×2 example using the eigenvalues we just calculated for the matrix R, and second, a 3×3 example solved using "Gaussian elimination".

Given the eigenvalues found above we solve in general,

$$\begin{pmatrix} 1-\lambda & 4 \\ 2 & 6-\lambda \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \lambda \begin{pmatrix} x \\ y \end{pmatrix}$$
$$\begin{pmatrix} 1-7.2749 & 4 \\ 2 & 6-7.2749 \end{pmatrix} = 7.2749 \begin{pmatrix} x \\ y \end{pmatrix}$$
$$\begin{pmatrix} -6.2749 & 4 \\ 2 & -1.2749 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Note that the two equations derived from this system are **not** linearly independent – we just multiply the first by $-2/6.2749$ to get the second. As a result, there is an infinite number of solutions $-6.2749x + 4y = 0$ or $y = 1.5687x$. Choose $x = 1$ to find,

$$(x,y) = (1, 1.5687).$$

Then, normalize so that the eigenvector has unit length,

$$\mathbf{e} = (x,y)/\sqrt{x^2+y^2} = (0.5375, 0.8432).$$

It may be verified that this is an eigenvector (upto a minus sign) using Matlab,

```
[V,D]=eig([1,4;2,6])
V =
   -0.9528   -0.5375
    0.3037   -0.8432
D =
   -0.2749         0
         0    7.2749
```

Next, consider the 3×3 example in which we transform to reduced row-echelon form to solve:

$$L = \begin{pmatrix} 1 & 4 & 1 \\ 4 & 2 & 8 \\ 3 & 2 & 1 \end{pmatrix}$$

Using Matlab, the eigenvalues are:

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} 8.4820 \\ -2.2410 + 1.7305i \\ -2.2410 - 1.7305i \end{pmatrix}$$

Let's calculate the eigenvector $\mathbf{e}_1 = (e_1, e_2, e_3)^T$ corresponding to the first eigenvalue, $\lambda_1 = 8.4820$:

$$(L - \lambda_1 I)\mathbf{e}_1 = 0$$

which we can write explicitly as,

$$\begin{aligned} B &= L - \lambda_1 I \\ &= \begin{pmatrix} -7.4820 & 4.0000 & 1.0000 \\ 4.0000 & -6.4820 & 8.0000 \\ 3.0000 & 2.0000 & -7.4820 \end{pmatrix} \end{aligned}$$

Now, to solve $(L - \lambda_1 I)\mathbf{e} = 0$ we add a column of zeros to the RHS of the matrix and perform Gaussian elimination on rows to bring the equation to a reduced row-echelon form. Because the RHS is of zeros only, it does not change and we do not write it here explicitly. After several row reductions to ensure zero below the diagonal, we find,

$$B = \begin{pmatrix} 1 & 0 & -1.1841 \\ 0 & 1 & -1.9649 \\ 0 & 0 & 0.0001 \end{pmatrix}$$

Since we have done our calculations using 4 or 5 significant digits, we must consider 0.0001 on the last row to be zero – a round-off error. Given this, e_3 can be chosen arbitrarily so we **choose** $e_3 = 1$. Then, the second equation tells us that $e_2 - 1.9649e_3 = 0$, so $e_2 = 1.9649$. The first equation tells us that $e_1 - 1.1841e_3 = 0$ so that $e_1 = 1.1841$. Therefore, $\mathbf{e}_1 = (1.1841, 1.9649, 1)$. We normalize by the norm of the vector, $\|\mathbf{e}_1\| = \sqrt{e_1^2 + e_2^2 + e_3^2} = 2.5026$, to find $\mathbf{e}_1 = (0.4731, 0.7851, 0.3996)$. This is in fact equal to Matlab's solution:

```
[U,D]=eig(L)
U = [0.4732 + 0.0000i   0.6865 + 0.0000i   0.6865 + 0.0000i
     0.7851 + 0.0000i  -0.5190 + 0.3747i  -0.5190 - 0.3747i
     0.3996 + 0.0000i  -0.1492 - 0.3109i  -0.1492 + 0.3109i]
D = [8.4820 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
     0.0000 + 0.0000i  -2.2410 + 1.7305i   0.0000 + 0.0000i
     0.0000 + 0.0000i   0.0000 + 0.0000i  -2.2410 - 1.7305i]
```

Matrix diagonalization

Given a matrix U whose columns are the eigenvectors of R, we may need to calculate,

$$U^{-1}RU = \begin{pmatrix} -0.9528 & -0.5375 \\ 0.3037 & -0.8432 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 4 \\ 2 & 6 \end{pmatrix} \begin{pmatrix} -0.9528 & -0.5375 \\ 0.3037 & -0.8432 \end{pmatrix}$$

and show that this is diagonal (i.e. only has values along the diagonal). To do this, we find the the eigenvectors of R and insert them into a matrix U as the columns. We then find the inverse of this matrix of eigenvectors, U^{-1}, and multiply $U^{-1}RU$ to show the result is diagonal:

$$U^{-1}RU = \begin{pmatrix} -0.2749 & 0 \\ 0 & 7.2749 \end{pmatrix}$$

We see that $U^{-1}RU$ is indeed diagonal and the diagonal elements are the eigenvalues of R.

Gram-Schmidt orthogonalization

In order to orthonormalize the columns of L (that is, make the columns orthogonal to one another with unit norm), carry out the following steps:

$$L = \begin{pmatrix} 1 & 4 & 1 \\ 4 & 2 & 8 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 \end{pmatrix}$$

Then,

$$
\begin{aligned}
L &= \begin{pmatrix} 1 & 4 & 1 \\ 4 & 2 & 8 \\ 3 & 2 & 1 \end{pmatrix} \\
\mathbf{c}_1 &= \mathbf{c}_1/|\mathbf{c}_1| \\
L &= \begin{pmatrix} 0.1961 & 4 & 1 \\ 0.7845 & 2 & 8 \\ 0.5883 & 2 & 1 \end{pmatrix} \\
\mathbf{c}_2 &= \mathbf{c}_2 - \mathbf{c}_1(\mathbf{c}_2^T\mathbf{c}_1) \\
L &= \begin{pmatrix} 0.1961 & 3.3077 & 1.0000 \\ 0.7845 & -0.7692 & 8.0000 \\ 0.5883 & -0.0769 & 1.0000 \end{pmatrix} \\
\mathbf{c}_2 &= \mathbf{c}_2/|\mathbf{c}_2| \\
L &= \begin{pmatrix} 0.1961 & 0.9738 & 1.0000 \\ 0.7845 & -0.2265 & 8.0000 \\ 0.5883 & -0.0226 & 1.0000 \end{pmatrix} \\
\mathbf{c}_3 &= \mathbf{c}_3 - \mathbf{c}_1(\mathbf{c}_3^T\mathbf{c}_1) - \mathbf{c}_2(\mathbf{c}_3^T\mathbf{c}_2) \\
L &= \begin{pmatrix} 0.1961 & 0.9738 & 0.4533 \\ 0.7845 & -0.2265 & 2.2667 \\ 0.5883 & -0.0226 & -3.1733 \end{pmatrix} \\
\mathbf{c}_3 &= \mathbf{c}_3/|\mathbf{c}_3| \\
L &= \begin{pmatrix} 0.1961 & 0.9738 & 0.1155 \\ 0.7845 & -0.2265 & 0.5774 \\ 0.5883 & -0.0226 & -0.8083 \end{pmatrix}
\end{aligned}
$$

In Matlab, the same process is carried out using the code:

```
% the input matrix:
L=[1,4,1;4,2,8;3,2,1];
% orthonormalize columns:
L(:,1)=L(:,1)/norm(L(:,1));
L(:,2)=L(:,2)-L(:,1)*(L(:,2)'*L(:,1));
L(:,2)=L(:,2)/norm(L(:,2));
L(:,3)=L(:,3)-L(:,1)*(L(:,3)'*L(:,1))-L(:,2)*(L(:,3)'*L(:,2));
L(:,3)=L(:,3)/norm(L(:,3));
L
% test outcome columns for orthonormality:
L1_norm=L(:,1)'*L(:,1),L2_norm=L(:,2)'*L(:,2),L3_norm=L(:,3)'*L(:,3),
L1_times_L2=L(:,1)'*L(:,2)
L1_times_L3=L(:,1)'*L(:,3)
L2_times_L3=L(:,2)'*L(:,3)
```

Null space of a matrix

Suppose we need to calculate the null space of the matrix A,

$$\mathsf{A} = \begin{pmatrix} 1 & 3 & 2 & 5 \\ 2 & 3 & 1 & 2 \\ 3 & 6 & 3 & 7 \end{pmatrix}$$

The null space of the matrix A is defined by the vectors satisfying $\mathsf{A}\mathbf{x} = 0$. Start by adding the rhs to the above,

$$\mathsf{A}_1 = \left(\begin{array}{cccc|c} 1 & 3 & 2 & 5 & 0 \\ 2 & 3 & 1 & 2 & 0 \\ 3 & 6 & 3 & 7 & 0 \end{array} \right)$$

Using Gaussian elimination, discussed above, we arrive at the reduced matrix,

$$\mathsf{A}_1 = \left(\begin{array}{cccc|c} 1 & 3 & 2 & 5 & 0 \\ 0 & -3 & -3 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right)$$

This result implies the equations,

$$\begin{aligned} x + 3y + 2z + 5w &= 0 \\ -3y - 3z - 8w &= 0, \end{aligned}$$

which we can solve for x, y in terms of z and w:

$$\begin{aligned} y &= -z - (8/3)w \\ x &= -(3y + 2z + 5w) \\ &= -(3(-z - (8/3)w) + 2z + 5w) \\ &= z + 3w. \end{aligned}$$

Thus, we can write that in general,

$$\begin{bmatrix} z+3w \\ -z-(8/3)w \\ z \\ w \end{bmatrix} = \begin{bmatrix} z \\ -z \\ z \\ 0z \end{bmatrix} + \begin{bmatrix} 3w \\ -(8/3)w \\ 0w \\ w \end{bmatrix} = z \begin{bmatrix} 1 \\ -1 \\ 1 \\ 0 \end{bmatrix} + w \begin{bmatrix} 3 \\ -(8/3) \\ 0 \\ 1 \end{bmatrix}$$

and the null space is given by the two vectors that are multiplied by z and w, correspondingly. Normalizing these two vectors by their norm (i.e. magnitude) so that we can compare our results with Matlab. In Matlab, we find

```
a1=[1,-1 1 0]; a2=[3 -8/3 0 1];
a1=a1/norm(a1);a2=a2/norm(a2);
my_fprintf_array(a1);my_fprintf_array(a2);
a1=[ 0.57735  -0.57735  0.57735  0 ];
a2=[ 0.725241  -0.644658  0  0.241747 ];
% check that these are, indeed null vectors of A:
A*a1'
A*a2'
% and they are both zero, as expected
```

Gradient

Consider the function $J(x,y) = 2x+y+3xy+x^2/2+y^2/2$. The *gradient* of this function is the following vector,

$$\begin{aligned}\nabla J(x,y) &= \left(\frac{\partial J}{\partial x}, \frac{\partial J}{\partial y}\right) \\ &= (2+3y+x, 1+3x+y)\end{aligned}$$

Evaluating the gradient at a specific point $\mathbf{x} = (x,y) = (1,2)$, we find

$$\nabla J(x,y) = (2+3\times 2+1, 1+3\times 1+2) = (9,6),$$

while the value of the function itself at this point is

$$J(1,2) = 2*1+2+3*1*2+1^2/2+2^2/2 = 12.5.$$

Consider a small vector $\delta\mathbf{x} = (\delta x, \delta y) = 0.001 \times (9,6)$ in the same direction of the gradient at this specific point and let's evaluate the change to the function J at the point $\mathbf{x}+\delta\mathbf{x}$. The first two terms in the Taylor expansion of the multivariate function J are

$$\begin{aligned}J(\mathbf{x}+\delta\mathbf{x}) &\approx J(\mathbf{x}) + \frac{\partial J}{\partial x}\delta x + \frac{\partial J}{\partial y}\delta y \\ &= J(\mathbf{x}) + \nabla J \cdot \delta\mathbf{x}.\end{aligned}$$

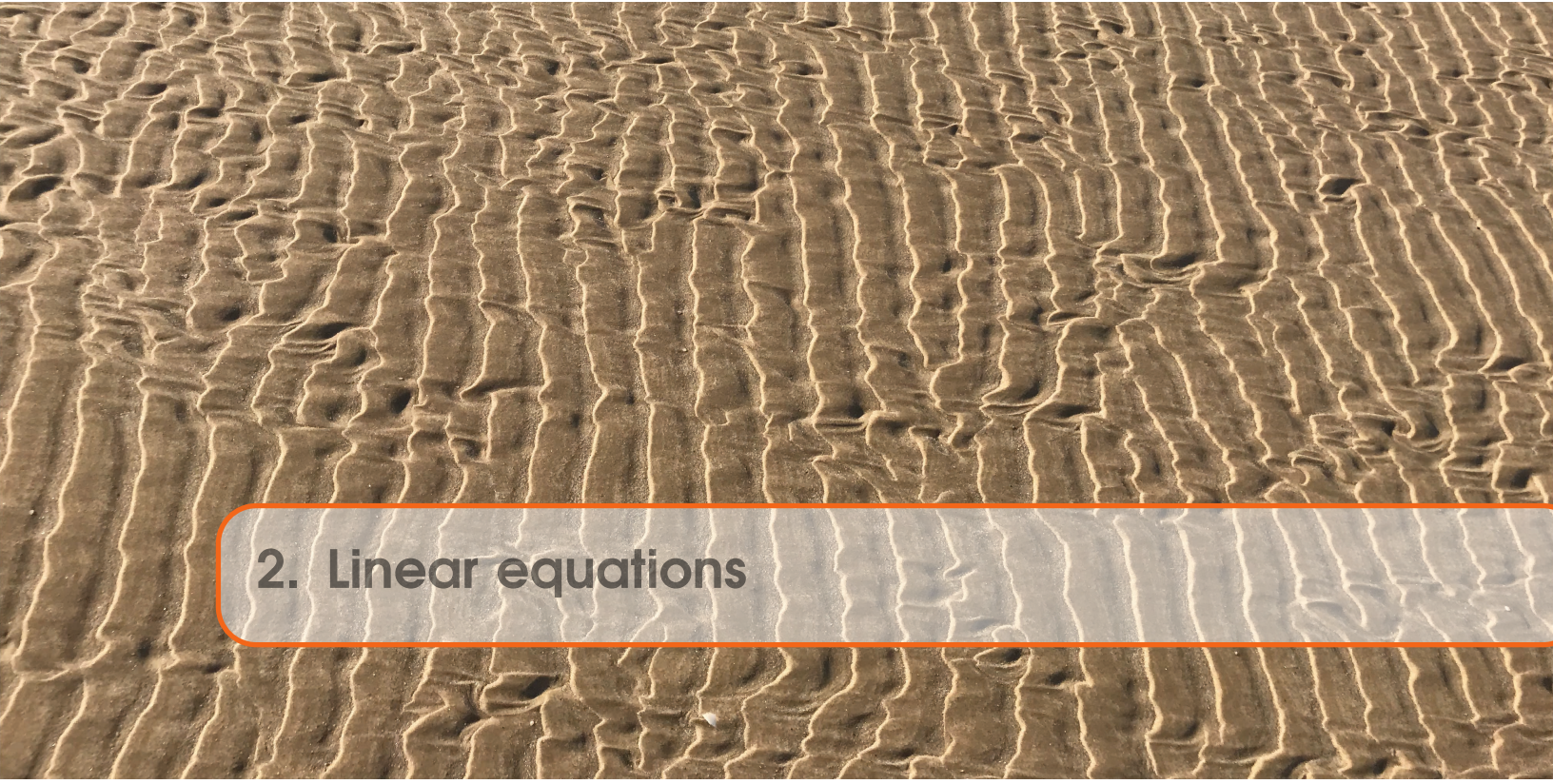
Substituting the numerical values for $\mathbf{x}$ and $\delta\mathbf{x}$, using Matlab notations for the scalar product

$J(\mathbf{x}+\delta\mathbf{x}) \approx$ `12.5+[9;6]'*0.001*[9;6]=12.617`.

The gradient at a given point indicates the direction of fastest increase of the function. Indeed, when we moved in the direction of the gradient the function increased. We can check directly what the value of the function is at the new point to find,

$$J(\mathbf{x}+\delta\mathbf{x}) = J((1,2)+0.001(9,6)) = J(1.09,2.06) = 12.6172,$$

which is remarkably close to the Taylor approximation.



2. Linear equations

2.1 Motivation

Linear equations arise in the analysis of electrical networks, chemical reactions, network analysis, Leontief economic models, ranking of sports teams, numerical finite difference solution of PDEs, medical tomography, and much more.

In this chapter, we discuss several ways to efficiently solve large sets of linear equations. In some applications, it is necessary to solve systems of the form $\mathsf{A}\mathbf{x} = \mathbf{b}$ for many right hand side $\mathbf{b}$ vectors. This is done using the "LU decomposition" which deconstructs a matrix A such that the linear equations $\mathsf{A}\mathbf{x} = \mathbf{b}$ can be efficiently solved for many right hand sides. We will see, however, that LU decomposition does not work well for sparse matrices (with many zero entries, as occurs in many applications) and thus we then cover iterative methods, which are often better for solving equations involving sparse matrices.

Throughout this section, and in particular for LU and iterative methods, a good reference is Strang (2006).

2.1.1 Medical tomography

Medical tomography serves as a fascinating example of the power of applied linear algebra. This type of computerized imaging uses detection of the intensity of X-rays fired through tissue to determine the density of tissues in question. With these densities, we can generate images for medical expert analysis. As we'll see, this kind of imaging, at scale, relies on the solution to millions of linear equations and may lead to either under-determined or over-determined systems, that is, to more unknowns than equations, or more equations than unknowns.

The objective is to calculate tissue density (brain, bone and so on) in a 2d section from multiple X-rays through the tissue. The idea is that the X-ray intensity transmitted via a distance s, $I(s)$, is a function of tissue density along the ray path, $\rho(x,y)$. If the density were constant, ρ_0, the intensity would simply

be an exponentially decaying function of the distance traveled within the tissue, $I(s) = I(0)\exp(-\mu\rho_0 s)$, where μ is some appropriate constant depending on the tissue composition. When the tissue density is not constant, however, this is replaced by an integral along the path as follows.

Let each path be described by $(x(u), y(u))$ with u going from 0 to s; Some 2d examples are:

$(x,y) = (u,0)$	a path along the y-axis
$(x,y) = (0,u)$	a path along the x-axis
$(x,y) = (u,u)$	a diagonal path

Intensity is related to tissue density as ,

$$I(s) = I(0)\exp\left(-\int_0^s \mu\rho(x(u),y(u))du\right),$$

or

$$\log(I(s)/I(0)) = -\int_0^s \mu\rho(x(u),y(u))du.$$

Represent the density values on a grid in each two dimensional section, $\rho(x_i, y_j) = \rho_{ij}$,

$$(\rho_{11}, \rho_{12}, \ldots, \rho_{1n}, \rho_{21}, \rho_{22}, \ldots, \rho_{mn})$$

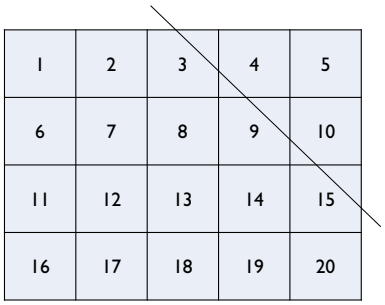
which we write as a vector $\boldsymbol{\rho}$,

$$\boldsymbol{\rho} = \{\rho_1, \ldots, \rho_{mn}\}, \quad (k = 1, m\times n)$$

and write the integral as a sum,

$$-\int_0^s \mu\rho(x(u),y(u))du = -\sum_{k=1}^{mn} \mu B_{lk}\rho_k \Delta u$$

where B_{lk} is one when X-ray l goes through location $\mathbf{x}_k = (x_i, y_j)$ and zero otherwise. This leads to a linear system $\mathsf{A}\boldsymbol{\rho} = \mathbf{b}$. The rhs $\boldsymbol{b}$ is made of $\log(I(s)/I(0))$ entries and the matrix A is $(\mu\Delta u)\mathsf{B}$, where each row represents one trajectory through the tissue. Here is an example of one ray and the corresponding LHS of the equation,



In this example of a ray passing through grid cells 3, 4, 9, 10 and 15, the corresponding LHS of the equation is,

$$\int \rho\, du = \rho_3\Delta u_3 + \rho_4\Delta u_4 + \rho_9\Delta u_9 + \rho_{10}\Delta u_{10} + \rho_{15}\Delta u_{15} = RHS = \frac{-1}{\mu}\log(I(s)/I(0))$$

and in matrix form this is,

$$\begin{pmatrix} & & & \dots & & & & & & \\ 0 & 0 & \Delta u_3 & \Delta u_4 & 0\dots0 & \Delta u_9 & \Delta u_{10} & 0\dots0 & \Delta u_{15} & \dots0 \\ & & & \dots & & & & & & \\ & & & \dots & & & & & & \end{pmatrix} \begin{pmatrix} \rho_1 \\ \vdots \\ \vdots \\ \rho_{20} \end{pmatrix} = R\vec{H}S$$

Note that a given location $\mathbf{x}_k = (x_i, y_j)$ is encountered by many rays. The dimension of the matrix A is $L \times K$ =(number of X-ray trajectories)×(number of grid points). Increasing the resolution of (x_i, y_j) to get a detailed image, we would always have $K > L$, i.e., more unknowns $\rho(x_i, y_j)$ than equations, leading to an under-determined system which can be solved using SVD, as discussed in sections 5.2.5 and 5.2.6.

While the above derivation conveys the essence of medical tomography, in practice the calculation is often done using a "Radon transform".

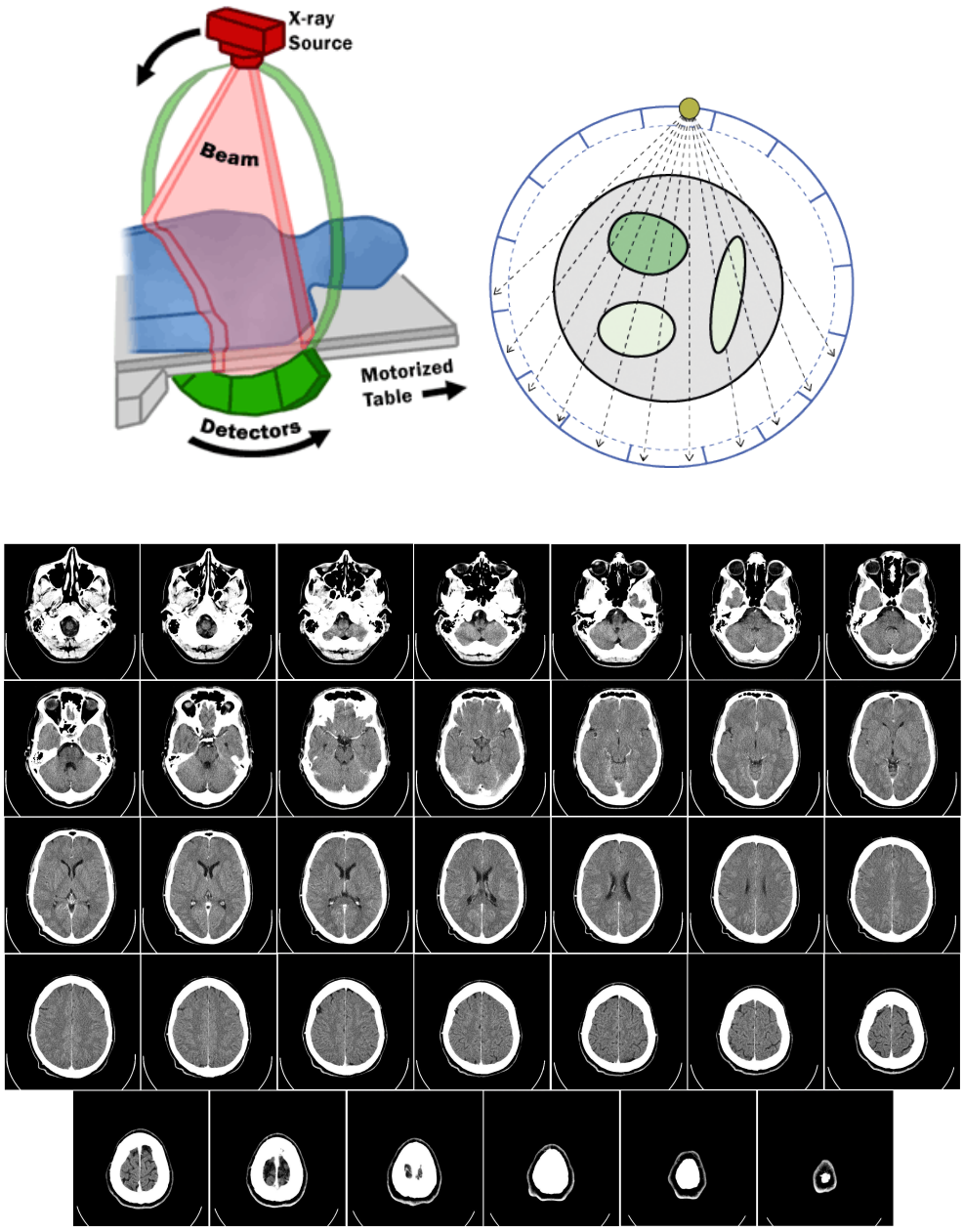


Figure 2.1: Tomography schematics and brain scan images from here, here and here.

2.2 Geometric interpretations for linear equations

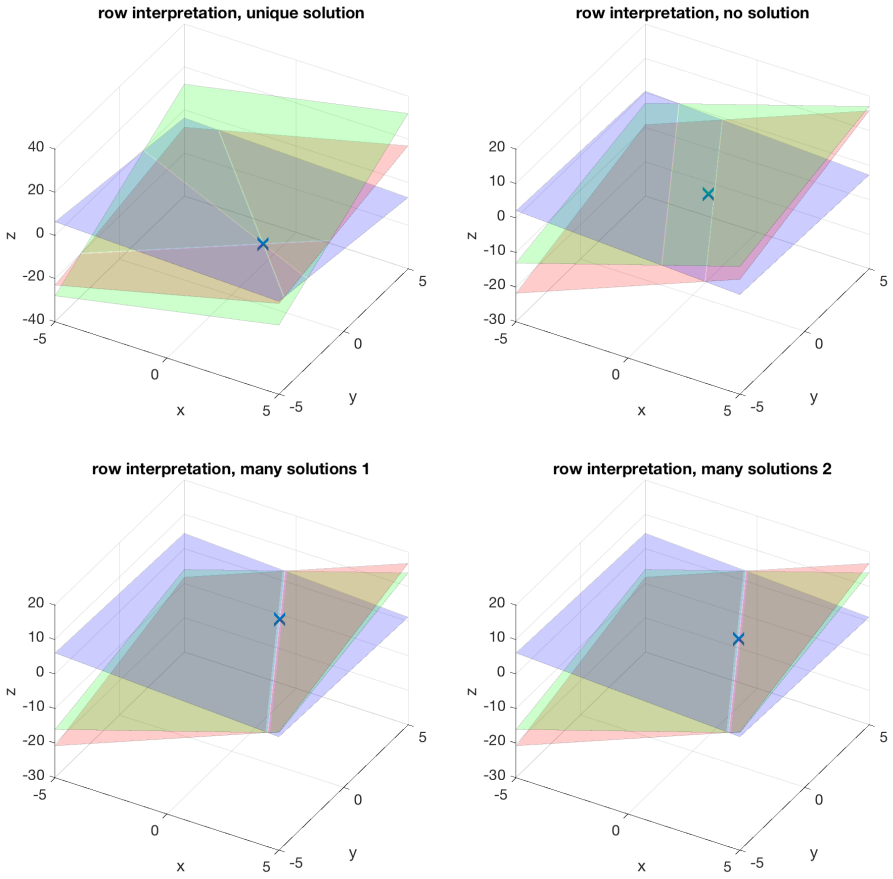
A linear system of equations, $A\mathbf{x} = \mathbf{b}$, can be interpreted using the rows or the columns of A to help understand why there may be no solutions, a unique solution, or an infinite number of solutions. For example, consider

$$\begin{aligned} 5x+3y-2z &= 6 \\ 2x+4y-z &= -2 \\ x+3y+3z &= -1 \end{aligned} .$$

This system of equations can be re-written in matrix form as $A\mathbf{x} = \mathbf{b}$,

$$\begin{bmatrix} 5 & 3 & -2 \\ 2 & 4 & -1 \\ 1 & 3 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -2 \\ -1 \end{bmatrix} .$$

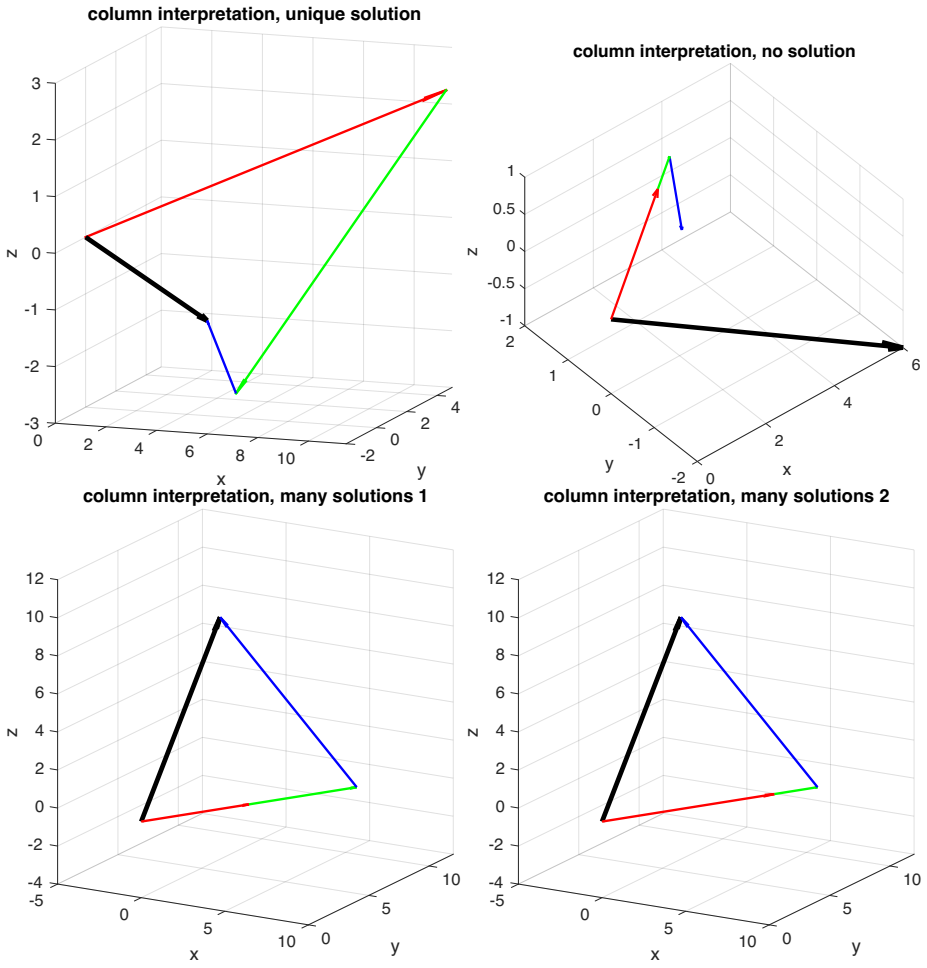
Let's start by examining the *rows* of this system. Each equation represents a plane in 3D space. Row 1: $5x+3y-2z=6$ is equivalent to an equation for the height of the plane $z = 2.5x+1.5y-3$, and similarly for row 2: $2x+4y-z=-2$ and row 3: $x+3y+3z=-1$. Plotting these three planes, we see that they intersect at a particular point which is the solution to the system of equations. When two or more of the planes are parallel (but not identical), or when two pairs of planes intersect along parallel lines, there is no intersection of all three planes and therefore no solution. When the three plans intersect along a *line* instead of at a point, there is an infinite number of solutions.



Alternatively, we can interpret the *columns* of the system geometrically. We can rewrite the system of equations as a sum over three column vectors that is equal to the column vector on the RHS,

$$x\begin{bmatrix}5\\2\\1\end{bmatrix}+y\begin{bmatrix}3\\4\\3\end{bmatrix}+z\begin{bmatrix}-2\\-1\\3\end{bmatrix}=\begin{bmatrix}6\\-2\\-1\end{bmatrix}.$$

We are looking for the three numbers x, y, z such that the linear combination above equals the RHS. We plot each of the above column vectors such that each starts at the end of the previous one. We then plot the RHS column vector and find that it ends at the same point as the third vector from the LHS. The x, y, z for which the sum of the LHS vectors and the RHS vector meet is the solution to the linear system of equations. When the three vectors on the LHS are not linearly independent, it may not be possible to find a combination of these vectors that is equal to the RHS, and in this case there is no solution. An infinite number of solutions may occur if, say, two of the LHS vectors are parallel and the RHS vector is such that it can still be expressed in terms of the LHS vectors. In this case, one can find different combinations of the two parallel LHS vectors that can be part of a combination that is equal to the RHS, yielding an infinite number of solutions.



2.3 Direct solution to linear equations by LU decomposition

In this section, we describe the LU decomposition algorithm, demonstrate it with a specific example, show how it is used to efficiently solve systems of linear equations with many right hand sides, discuss its computational cost and explain why the algorithm works.

2.3.1 Gaussian elimination

As a reminder, consider the following example, $\mathbf{A}\mathbf{x} = \mathbf{b}$,

$$\begin{bmatrix} -2 & -1/3 & -2\frac{2}{3} \\ 6 & 5 & 4 \\ 3 & 4.5 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -10\frac{2}{3} \\ 28 \\ 21 \end{bmatrix}.$$

(1) We start by exchanging the first two equations such that the coefficient of the first variable in the first equation is as large as possible,

$$\begin{bmatrix} 6 & 5 & 4 \\ -2 & -1/3 & -2\frac{2}{3} \\ 3 & 4.5 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 28 \\ -10\frac{2}{3} \\ 21 \end{bmatrix}$$

(2) Now, subtract $-2/6$ times the first equation from the second, subtract $3/6$ of the first equation from the third,

$$\begin{bmatrix} 6 & 5 & 4 \\ 0 & 1\frac{1}{3} & -1\frac{1}{3} \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 28 \\ -1\frac{1}{3} \\ 7 \end{bmatrix}$$

(3) Exchange the last two equations again, to make sure the pivot (coefficient of y in second equation) is largest,

$$\begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 1\frac{1}{3} & -1\frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}. = \begin{bmatrix} 28 \\ 7 \\ -1\frac{1}{3} \end{bmatrix}$$

(4) Subtract $1\frac{1}{3}/2$ times the second equation from the third,

$$\begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 28 \\ 7 \\ -6 \end{bmatrix}$$

which we now solve by back substitution:

$$\begin{aligned} -2z &= -6 && \Rightarrow z = 3 \\ 2y + 3 \times 1 &= 7 && \Rightarrow y = 2 \\ 6x + 5 \times 2 + 4 \times 3 &= 28 && \Rightarrow x = 1 \end{aligned}$$

2.3.2 LU decomposition algorithm

The algorithm for LU decomposition of a matrix A follows a similar approach to Gaussian elimination. The objective is to write the matrix as a product of an upper triangular and a lower triangular matrix. Because we want to avoid having zero or small pivots, we normally need to multiply the matrix by a permutation matrix before it can be decomposed such that $\mathsf{LU} = \mathsf{PA}$. We will see that this decomposition is useful as it helps us solve $\mathsf{A}\mathbf{x} = \mathbf{b}$ efficiently for many RHS vectors $\mathbf{b}$.

We note first that the each step in the above Gaussian elimination example could be expressed using permutation and manipulation matrices,

(1) is obtained by multiplying A by the permutation matrix

$$\mathsf{P}_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(2) is obtained by multiplying A by the manipulation matrix,

$$\mathsf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 2/6 & 1 & 0 \\ -3/6 & 0 & 1 \end{bmatrix}$$

and step (3) is again a permutation using

$$\mathsf{P}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Step (4) uses another manipulation matrix,

$$\mathsf{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2/3 & 1 \end{bmatrix}.$$

Combining all steps, the manipulation bringing A to an upper diagonal matrix can therefore be written as,

$$\mathsf{U} = \mathsf{L}_2\mathsf{P}_2\mathsf{L}_1\mathsf{P}_1\mathsf{A} = \begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 0 & -2 \end{bmatrix}$$

and therefore,

$$\mathsf{A} = \mathsf{P}_1^{-1}\mathsf{L}_1^{-1}\mathsf{P}_2^{-1}\mathsf{L}_2^{-1}\mathsf{U}.$$

It turns out that, magically (section 2.3.5), this can also be written as

$$\mathsf{A} = \mathsf{P}^{-1}\mathsf{LU}$$

where,

$$\mathsf{U} = \begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 0 & -2 \end{bmatrix}, \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/3 & 2/3 & 1 \end{bmatrix}, \quad \mathsf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Note the relationship with the original $\mathsf{L}_i, \mathsf{P}_i$, which is explained to some degree by the relation between L_1 and its inverse,

$$\mathsf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix}, \quad \mathsf{L}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}.$$

The LU algorithm calculates these U, L and P as follows:

1. Initialize $\mathsf{U} = \mathsf{A}$, and initialize L and P to the identity matrix.
2. Starting with the first column, exchange rows in U such that the entry in the column with the largest absolute value becomes the diagonal value (the pivot). Similarly, exchange the same rows in P, and similarly exchange them in L *but only below the diagonal*.
3. Zero the column under the pivot in U by subtracting from each row by a factor of the pivot value. Insert this "subtraction factor" in the appropriate entry of L.
4. Move to the next column of U and repeat the above steps until you arrive at an upper diagonal U and a lower diagonal L matrix.

2.3.3 Example of LU decomposition

■ **Example 2.1** Consider the following 3×3 matrix A,

$$\mathsf{A} = \begin{bmatrix} -2 & -1/3 & -2\frac{2}{3} \\ 6 & 5 & 4 \\ 3 & 4.5 & 3 \end{bmatrix}.$$

Initialize,

$$\mathsf{U} = \begin{bmatrix} -2 & -1/3 & -2\frac{2}{3} \\ 6 & 5 & 4 \\ 3 & 4.5 & 3 \end{bmatrix} \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathsf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Now exchange rows 2 and 1 so that the pivot becomes the entry in the column with the largest magnitude. Reflect this exchange in P,

$$\mathsf{U} = \begin{bmatrix} 6 & 5 & 4 \\ -2 & -1/3 & -2\frac{2}{3} \\ 3 & 4.5 & 3 \end{bmatrix} \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathsf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

To zero out column 1, rows 2 and 3, subtract from row 2 of U $-2/6 = -1/3$ times row 1, and from row 3 $3/6 = 1/2$ times row 1. Put these subtraction factors in the first column below the diagonal in L,

$$\mathsf{U} = \begin{bmatrix} 6 & 5 & 4 \\ 0 & 1\frac{1}{3} & -1\frac{1}{3} \\ 0 & 2 & 1 \end{bmatrix} \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} \quad \mathsf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Again we have to swap rows 2 and 3 so that the largest magnitude value in column 2 is the pivot. We also swap the elements below the diagonal of L and adjust the permutation matrix accordingly,

$$\mathsf{U} = \begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 1\frac{1}{3} & -1\frac{1}{3} \end{bmatrix} \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/3 & 0 & 1 \end{bmatrix} \quad \mathsf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

To zero out column 2 below the pivot, subtract $1\frac{1}{3}/2 = 2/3$ times row 2 from row 3 and place the factor below the diagonal in L,

$$\mathsf{U} = \begin{bmatrix} 6 & 5 & 4 \\ 0 & 2 & 1 \\ 0 & 0 & -2 \end{bmatrix} \quad \mathsf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/3 & 2/3 & 1 \end{bmatrix} \quad \mathsf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

It is now easy to check that $\mathsf{LU} = \mathsf{PA}$, as desired. ∎

2.3.4 Solving linear equations using the LU decomposition for multiple RHS

LU is useful when we need to solve $\mathsf{A}\mathbf{x} = \mathbf{b}$ for many RHS $\mathbf{b}$. First, consider the cost of Gaussian elimination. If we call each division or multiplication-subtraction one operation, it takes n operations to achieve zeros below the first entry in the first column of the matrix. With $n-1$ rows below the first row, we have $n(n-1) = n^2 - n$ operations. If we reduce the elimination down to k equations, $k^2 - k$ operations are needed to clear out the column below the pivot. As such the total operations become,

$$\begin{aligned}(1^2 + \ldots + n^2) - (1 + \ldots + n) &= \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} \\ &= \frac{n^3 - n}{3}\end{aligned}$$

For large n, this is approximately $n^3/3$.

Given the LU decomposition, we can now solve $\mathsf{A}\mathbf{x} = \mathbf{b}$ efficiently for many right hand sides $\mathbf{b}$, using only $O(n^2)$ steps for each RHS. We use the LU Decomposition to write $\mathsf{LU}\mathbf{x} = \mathsf{P}\mathbf{b}$ and define $\mathsf{U}\mathbf{x} = \mathbf{c}$ so that we have $\mathsf{L}\mathbf{c} = \mathsf{P}\mathbf{b}$. We calculate the RHS $\mathsf{P}\mathbf{b}$, and efficiently solve the lower diagonal system $\mathsf{L}\mathbf{c} = \mathsf{P}\mathbf{b}$ for the vector $\mathbf{c}$ using forward substitution. Then, we efficiently solve the upper diagonal system $\mathsf{U}\mathbf{x} = \mathbf{c}$ using back substitution to find $\mathbf{x}$. This can be repeated for many $\mathbf{b}$ in $O(n^2)$ steps for each $\mathbf{b}$, which is far less expensive than the $n^3/3$ steps that would be required for a full Gaussian elimination (section 2.3.4).

2.3.5 Why does the LU decomposition algorithm work?

(Optional) Each step of the LU algorithm involves a row permutation followed by a row manipulation to set values under the diagonal to zero. This can be written as a multiplication by a permutation matrix and then by a manipulation matrix. For example, the first step in the above example may be written as,

$$\mathsf{U}_1 = \mathsf{E}_1\mathsf{P}_1\mathsf{A},$$

where

$$\mathsf{P}_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathsf{E}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -0.3 & 1 & 0 & 0 \\ -0.1 & 0 & 1 & 0 \\ 0.2 & 0 & 0 & 1 \end{bmatrix} \quad \mathsf{U}_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 2.5 & -1.8 & 8.1 \\ 0 & 5 & 6 & 7 \\ 0 & -2 & 5.6 & 6.2 \end{bmatrix}.$$

The final result of the Gaussian elimination is therefore,

$$\mathsf{U} = \mathsf{E}_3\mathsf{P}_3\mathsf{E}_2\mathsf{P}_2\mathsf{E}_1\mathsf{P}_1\mathsf{A}$$

which we may write as,

$$\mathsf{U} = \mathsf{E}_3(\mathsf{P}_3\mathsf{E}_2\mathsf{P}_3^T)(\mathsf{P}_3\mathsf{P}_2\mathsf{E}_1\mathsf{P}_2^T\mathsf{P}_3^T)(\mathsf{P}_3\mathsf{P}_2\mathsf{P}_1)\mathsf{A}.$$

Define $\mathsf{E}'_3 = \mathsf{E}_3$, $\mathsf{E}'_2 = \mathsf{P}_3\mathsf{E}_2\mathsf{P}_3^T$ and $\mathsf{E}'_1 = \mathsf{P}_3\mathsf{P}_2\mathsf{E}_1\mathsf{P}_2^T\mathsf{P}_3^T$. It turns out that these new matrices have the same form as the original E_i matrices: 1 along the diagonal and non-zero values only under the diagonal in the ith column. Thus we may write,

$$\mathsf{U} = (\mathsf{E}'_3\mathsf{E}'_2\mathsf{E}'_1)(\mathsf{P}_3\mathsf{P}_2\mathsf{P}_1)\mathsf{A}.$$

The inverse of each of the E'_i is obtained by changing the signs of the elements under the diagonal. Multiplying by these inverses and defining $\mathsf{P} = \mathsf{P}_3\mathsf{P}_2\mathsf{P}_1$ we find,

$$(\mathsf{E}'^{-1}_3\mathsf{E}'^{-1}_2\mathsf{E}'^{-1}_1)\mathsf{U} = \mathsf{P}\mathsf{A}.$$

Define $\mathsf{L} = (\mathsf{E}'^{-1}_3\mathsf{E}'^{-1}_2\mathsf{E}'^{-1}_1)$, and it turns out that because of the special structure of these matrices, their values are just added under the diagonal, leading to the familiar form of the L matrix. We therefore have $\mathsf{L}\mathsf{U} = \mathsf{P}\mathsf{A}$ as required.

For more details, see here, as well as chapters 20, 21 of Trefethen and Bau III (1997).

2.4 Iterative methods

When solving large systems, Gaussian elimination and LU decomposition may not be appropriate for several reasons. If the problem needs to be solved for a single RHS, it may take too many operations ($n^3/3$ is large and thus, elimination is computationally expensive). Also, if the matrix is highly sparse, its LU decomposition can be very dense (section 2.6.1), making LU operations unnecessarily expensive. Both issues can be resolved by using iterative methods that can be used to obtain an *approximate* solution for $\mathbf{x}$. The iterations are terminated when the solution has converged closely enough to the actual solution such that additional iterations yield the same solution. Because multiplying sparse matrices is inexpensive, this solution method can be significantly more efficient than Gaussian elimination. For a good resource on this subject, see Strang (2006),§7.4. To create an iterative scheme, we write the equation matrix as the difference between two matrices to be specified, $\mathsf{A} = \mathsf{S} - \mathsf{T}$. Thus, $\mathsf{A}\mathbf{x} = \mathbf{b}$ becomes $\mathsf{S}\mathbf{x} = \mathsf{T}\mathbf{x} + \mathbf{b}$. This is used to create an iteration scheme from k to $k+1$ by writing,

$$\mathsf{S}\mathbf{x}_{k+1} = \mathsf{T}\mathbf{x}_k + \mathbf{b}. \tag{2.1}$$

At every iteration, the RHS is known and we need to solve $\mathsf{S}\mathbf{x}_{k+1} = rhs$ for $\mathbf{x}_{k+1}$, so a convenient choice for S that allows an efficient solution is required. Of course, the iterative solution $\mathbf{x}_{k+1}$ needs to converge to the true value of $\mathbf{x}$, and to examine if it does, consider the error at iteration k, defined as $\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k$. Subtracting the equation $\mathsf{S}\mathbf{x} = \mathsf{T}\mathbf{x} + \mathbf{b}$ for the actual solution $\mathbf{x}$ from the above iteration scheme, we find the error equation,

$$\mathsf{S}\mathbf{e}_{k+1} = \mathsf{T}\mathbf{e}_k \qquad \text{Error Equation,} \tag{2.2}$$

or, equivalently, $\mathbf{e}_{k+1} = \mathsf{S}^{-1}\mathsf{T}\mathbf{e}_k$. An iterative scheme is *convergent* if the error goes to zero as the number of iterations increases. This occurs when every eigenvalue of $\mathsf{S}^{-1}\mathsf{T}$ satisfies $|\lambda_i| < 1$. The rate of convergence depends on the size of $\max_i |\lambda_i|$ — the smaller this value is, the faster the solution should converge to the true value of $\mathbf{x}$. This maximum absolute value eigenvalue, $\max_i |\lambda_i|$, is termed

the *spectral radius*. The reason the rate of convergence is governed by the absolute value of the largest eigenvalue is seen by expanding the initial guess in terms of the eigenvectors of $\mathsf{S}^{-1}\mathsf{T}$, $\mathbf{v}_i$, and denoting the corresponding eigenvalues λ_i. Thus, we may write the error initially, after one and then after k iterations as,

$$\begin{aligned}\mathbf{e}_0 &= c_1\mathbf{v}_1 + \ldots + c_n\mathbf{v}_n \\ \mathbf{e}_1 &= \mathsf{S}^{-1}\mathsf{T}\mathbf{e}_0 = c_1\lambda_1\mathbf{v}_1 + \ldots + c_n\lambda_n\mathbf{v}_n \\ \mathbf{e}_k &= (\mathsf{S}^{-1}\mathsf{T})^k\mathbf{e}_0 = c_1\lambda_1^k\mathbf{v}_1 + \ldots + c_n\lambda_n^k\mathbf{v}_n.\end{aligned}$$

The error will therefore clearly be dominated by the largest eigenvalue, which explains the above convergence criterion. Below we consider two specific simple iterative schemes based on different choices for S and T.

2.4.1 Jacobi

In Jacobi's method, we define S to be the diagonal part of A and then set $\mathsf{T} = \mathsf{S} - \mathsf{A}$. For example, we define A as,

$$\mathsf{A} = \begin{bmatrix} 0.5 & 0.25 \\ 0.1 & 0.1 \end{bmatrix}.$$

We would define S and T matrices as,

$$\mathsf{A} = \mathsf{S} - \mathsf{T} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.1 \end{bmatrix} - \begin{bmatrix} 0 & -0.25 \\ -0.1 & 0 \end{bmatrix}.$$

Using these matrices we would iterate as above using the equation,

$$\mathbf{x}_{k+1} = \mathsf{S}^{-1}(\mathsf{T}\mathbf{x}_k + \mathbf{b}) \tag{2.3}$$

for a given RHS vector $\mathbf{b}$, starting with an initial guess for $\mathbf{x}_0$. The guess for $\mathbf{x}_0$ should have a non-zero projection on the solution, and often a random vector or a vector of ones, $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ is appropriate. The eigenvalues of $\mathsf{S}^{-1}\mathsf{T}$ in this case are 0.7071 and -0.7071. Thus, max $|\lambda_i| < 1$, so the iterative scheme should converge to the true solution $\mathbf{x}$.

2.4.2 Gauss-Seidel

Another iterative scheme is Gauss-Seidel, where S is the lower triangular part of A and T is the remainder. For example, given

$$\mathsf{A} = \begin{bmatrix} 0.5 & 0.25 \\ 0.1 & 0.1 \end{bmatrix},$$

define S and T matrices as

$$\mathsf{A} = \mathsf{S} - \mathsf{T} = \begin{bmatrix} 0.5 & 0 \\ 0.1 & 0.1 \end{bmatrix} - \begin{bmatrix} 0 & -0.25 \\ 0 & 0 \end{bmatrix},$$

and carry out iterations as instructed above. The eigenvalues of $\mathsf{S}^{-1}\mathsf{T}$ are 0 and 0.5, so again the solution converges to the true value of $\mathbf{x}$ since max $|\lambda_i| < 1$, and it converges faster than in the Jacobi scheme as the largest eigenvalue (in absolute value) is smaller.

2.5 Existence of solutions and sensitivity to noise

There are two things to beware of when solving real-world problems. (1) The solution may be overly sensitive to "noise" because the matrix governing the system could be ill-conditioned. This could be measurement noise affecting the matrix or RHS, or noise introduced by round-off error due to the finite accuracy of a computer. We will discuss later the "condition number" of a matrix which allows us to prejudge if the solution is expected to be sensitive to such noise. (2) Even a problem that is not ill-conditioned could be sensitive to noise if not solved correctly, specifically if row exchanges are not properly used during the Gaussian elimination.

Consider $\mathsf{A}\mathbf{x}_1 = \mathbf{b}$ vs $\mathsf{A}\mathbf{x}_2 = \mathbf{b} + \delta\mathbf{b}$, using the following values,

$$\mathsf{A} = \begin{bmatrix} 4.67 & -3.03 \\ 6.87 & -4.67 \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} -5.56 \\ -8.31 \end{bmatrix}$$
$$\delta\mathbf{b} = \begin{bmatrix} -0.08 \\ 0.06 \end{bmatrix}$$

to find

$$\mathbf{x}_1 = [-0.83,\ 0.56]^T$$
$$\mathbf{x}_2 = [-1.39,\ -0.27]^T.$$

The small noise added to the RHS led to a very large difference in the solution. The two equations are shown as two lines for each system, with their solutions (crossing point of the lines), with and without the noise $\delta\mathbf{b}$, in Fig. 2.2. One can see that in this case the sensitivity to noise is large because the matrix is nearly singular. The two equations are therefore nearly identical, and thus their lines are nearly parallel – a small amount of noise can shift their crossing line significantly.

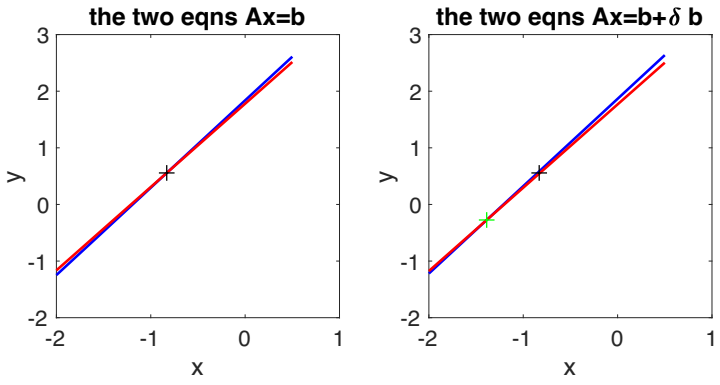


Figure 2.2: Solutions to two linear equations with and without noise. The small noise led to a large shift of the solution from the black to the green cross.

As for the importance of row exchanges (pivoting) and the effect of round-off errors, consider the following example of an equation $\mathsf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x} = (x, y)^T$,

$$\mathsf{A} = \begin{bmatrix} 0.0001 & -1 \\ 2 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}.$$

A direct solution gives

$$\mathbf{x} = \begin{bmatrix} 2.49988 \\ -2.99975 \end{bmatrix}.$$

Using Gaussian elimination, the matrix and RHS become

$$[\mathsf{A}, \mathbf{b}] = \begin{bmatrix} 0.0001 & -1 & 3 \\ 0 & 20001 & -59998 \end{bmatrix},$$

and back substitution results in,

$$\mathbf{x} = \begin{bmatrix} 2.49988 \\ -2.99975 \end{bmatrix}.$$

Next, using a limited precision, the Gaussian elimination stage gives,

$$[\mathsf{A}, \mathbf{b}] = \begin{bmatrix} 0.0001 & -1 & 3 \\ 0 & 20000 & -60000 \end{bmatrix},$$

and the back substitution therefore gives a wrong answer with a large error,

$$\mathbf{x} = \begin{bmatrix} 0 \\ -3 \end{bmatrix}.$$

Finally, consider the above problem, but this time with appropriate row pivoting. Start with the matrix and RHS written as,

$$\begin{bmatrix} 0.0001 & -1 & 3 \\ 2 & 1 & 2 \end{bmatrix},$$

exchange rows,

$$\begin{bmatrix} 2 & 1 & 2 \\ 0.0001 & -1 & 3 \end{bmatrix}.$$

Add the first row times $-0.0001/2$ to the second,

$$\begin{bmatrix} 2 & 1 & 2 \\ 0 & -1.0001 & 2.9999 \end{bmatrix},$$

and assume finite accuracy,

$$\begin{bmatrix} 2 & 1 & 2 \\ 0 & -1 & 3 \end{bmatrix}.$$

The second equation gives $y = -3$, and substituting in the first, we have $2x - 3 = 2 \Rightarrow x = 2.5$. The solution is therefore consistent with that which was obtained using full accuracy.

2.6 Dealing with huge systems

2.6.1 Sparse matrices

A sparse matrix is a matrix in which most of the entries are zero. By contrast, a dense matrix is one in which most of the entries are non-zero. We define the sparsity or density of a matrix as the number of non-zero entries in the matrix divided by the total number of entries.

Often, we can save enormous amounts of memory by storing only the non-zero entries of the matrix. There are several standards for storing a sparse matrix, and a common one is to store it as a list with row number, column number, and the value of the entry. For example, the matrix,

$$A = \begin{bmatrix} 0 & 0 & 0 & 4 & 0 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 & 0 \\ 0 & 0 & 7 & 0 & 0 \end{bmatrix}$$

can be stored as,

$$\text{list} = \begin{pmatrix} \text{row:} & 1 & 2 & 2 & 4 & 4 & 5 \\ \text{col:} & 4 & 2 & 3 & 1 & 4 & 3 \\ \text{val:} & 4 & 1 & 2 & 3 & 5 & 7 \end{pmatrix}$$

so that we can index into the row and column of the matrix to return a specific value. We've stored 18 digits instead of the original 25, and for very sparse matrices of a large dimension this method can lead to substantial memory savings.

When solving problems involving sparse matrix systems we often find that direct methods such as $PA = LU$ decomposition may not be the most efficient strategy. The reason is that the lower triangular L and upper triangular U matrices that result from the decomposition of a sparse matrix are substantially denser than the original sparse matrix.

To see why, note that the first step of the LU decomposition involves adding the first rows to the following ones to eliminate the entries in the first column, so assuming the rows are very large and sparse (say 0.1%), the density of the resulting second to last rows is expected to be roughly twice the original one (0.2%). That's because we are likely adding zeros to non-zero entries. The next step is adding the second row to all the rows below it, so the density of the third to last rows is now about 0.4%, etc. At this rate the density doubles every step. Eventually, the density increases sufficiently that we cannot assume that non-zero values are necessarily added to zero values and vice versa during the LU decomposition calculation, and in turn the approximation of the density doubling at every step breaks down. But in general, if the matrix is large enough the density of consecutive rows becomes larger quickly and therefore the LU decomposition is not expected to be sparse.

Fig. 2.3 shows an example of a sparse system yielding a dense LU decomposition.

As a result, iterative methods may be better for arriving at an approximate solution to huge sparse linear systems. Iterative methods also often work best (converge well) for equations based on diagonally-dominant, and/or symmetric positive definite matrices.

2.6.2 MapReduce

Dealing with massive data problems can lead to computational, storage and memory limitations. This can be addressed for some problems using the Google MapReduce algorithm, which splits up the problem across multiple processors. The steps of the algorithm are as follows,

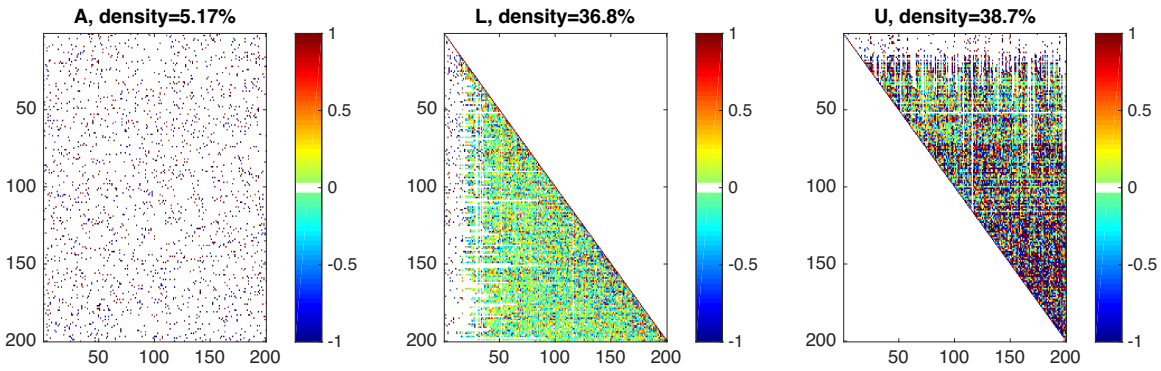


Figure 2.3: LU decomposition of a sparse matrix leads to dense L and U.

Split: The (large) data set is split up into manageable chunks across several processors and disks. Consider a problem involving 200,000 rows of data. We could, say, split up the data into 10 chunks of 20,000 rows each across 10 processors and disks. Each chunk is now of a manageable size that can fit into the computer memory and be analyzed.

Map: Once split up, each row of data is then *mapped* to a {KEY, VALUE} pair. You can have multiple values stored for a given key. For example, if calculating average flight delay and average flight time by the day of the week, then for each flight, the KEY would be the day of the week and the VALUE(s) would be the flight's delay and its duration.

Shuffle: Once all the data have been mapped to {KEY,VALUE} pairs, the data is then grouped according to KEY across processors. So, in the above example, MapReduce would group all the Monday flights from all the sub-data sets together, all the Tuesday flights together etc., *shuffling* the data across processors.

Reduce: With the data now sorted and grouped by key, we carry out the reduce stage. At this point, the MapReduce algorithm has provided us with all desired flight information information, grouped together for a given KEY. This allows us to finalize the computation for that KEY in the reduce step based on our initial objective. That is, in the reduce step we conduct a final aggregation or computational step to arrive at some desired summary statistic for a given KEY. For example, say we are given all the Monday flights first and their corresponding delays. These data can be used in the reduce step to calculate the average flight delay for Monday. Then we can do the same reduce for all the Tuesday through Sunday flights to find the average flight delay for these KEYS respectively.

■ **Example 2.2 Word counting example:** Fig. 2.4 shows an example illustrating how the MapReduce algorithm is used to count the number of words in a large file of text. ■

■ **Example 2.3 Matrix multiplication example:** Of the many applications of the MapReduce algorithm in computation we consider here matrix multiplication $\mathsf{A} = \mathsf{BC}$, or $a_{ik} = \sum_j b_{ij} c_{jk}$. This is further discussed in section 2.3.9 of Leskovec et al. (2014) with additional refinements and consideration of efficiency. Assume the matrices in question are so large that they cannot fit in a computer memory in order to be multiplied together. Start by defining a Map function that creates sets of the matrix elements in B and C required to compute a given element of A.

Map Function: In order to calculate a_{ik}, define the KEY as (i,k). The VALUE is then all entries b_{ij} that are used to calculate a_{ik}, written as $(\text{“B”}, j, b_{ij})$ for all j. Here “B” is a label denoting the

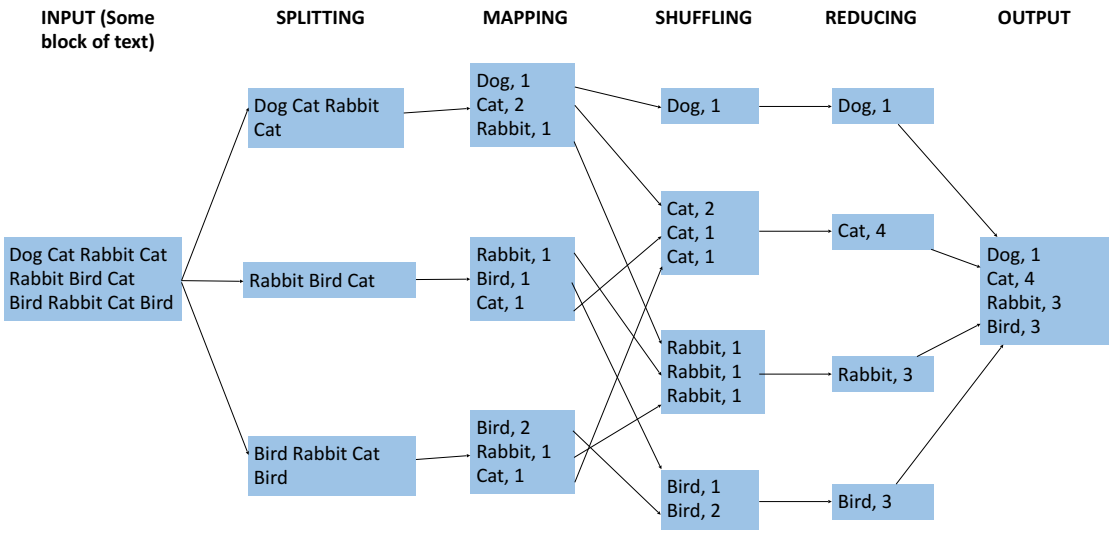


Figure 2.4: A word counting MapReduce example. The map step calculates the number of occurrences of each word in the sub-files and returns a {KEY,VALUE} pair corresponding to the (word, number of occurrences in sub-data). This is further analyzed in the reduce step, to calculate the total number of occurrences of a each word in the entire data.

matrix. The {KEY,VALUE} pairs are therefore $((i,k),(\text{“B”},j,b_{ij}))$. Similarly, the entries of C are mapped into {KEY,VALUE} pairs as $((i,k),(\text{“C”},j,c_{jk}))$. Note that any given element in B and C will be used multiple times for computing different elements of A, and thus each element of B and C will be mapped to multiple {KEY,VALUE} pairs.

Reduce Function: For each key (i,k) the reduce function receives two arrays, $(\text{“B”},j,b_{ij})$ and $(\text{“C”},j,c_{jk})$ for all values of j. The reduce function then sorts by j, multiplies b_{ij} by c_{jk}, and accumulates the products to find a_{ik}.

■

For further examples, see the course Matlab/python demonstrations of the algorithm used to analyze data from airline flights.

3. Eigenproblems

3.1 Motivation

We now move onto our discussion of eigenvectors and eigenvalues. We will explore a series of applications of these powerful tools such as Google's PageRank algorithm, partioning of graphs/networks, solving systems of linear differential equations, and even the explosive development of weather systems. Be sure to review the calculation of eigenvectors and eigenvalues in Section 1.2.

3.2 Google's PageRank

3.2.1 Introduction and explanation

Google originally created the PageRank algorithm to rank search results according to their importance, and this algorithm serves as a wonderful example of the utility of eigenvectors.

Consider the Internet as a network of websites with pages containing links to other pages, with an example depicted in Figure 3.1. Nodes represent the websites and the arrows represent links. The PageRank algorithm is based on the assumption that websites which have more incoming links from other important sites are more important, leading to a recursive definition. In addition, the value of an outgoing link from a given page is weighted according the number of outlinks that page has — if a page has 50 outlinks, the weight of each outlink is $1/50$.

Mathematically, the above definition can be expressed as follows. For n pages P_i, $i = 1, 2, \ldots, n$ the PageRank of page i is defined as,

$$r_i = \sum_{j \in L_i} \frac{r_j}{N_j}, \tag{3.1}$$

where N_j is the number of outlinks from page P_j and L_i are the pages that link to page P_i. We discuss a matrix formulation of the PageRank algorithm in the next section, and a method to solve for the rankings r_i.

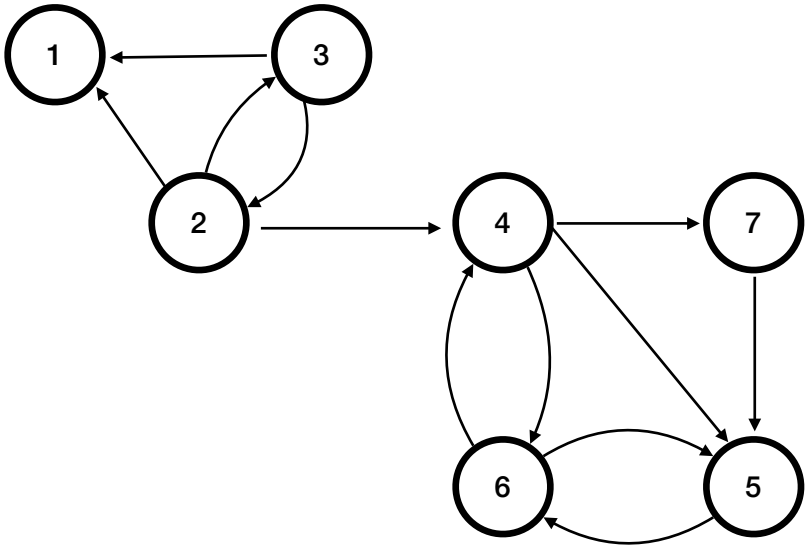


Figure 3.1: Network of 7 pages for demonstrating PageRank.

3.2.2 Matrix model of PageRank

Define a "transition matrix" of a given network, Q,

$$q_{ij} = \begin{cases} 1/N_i & \text{if } P_i \text{ links to } P_j \\ 0 & \text{otherwise,} \end{cases} \tag{3.2}$$

such that the ith row represents the outgoing links from page i. Writing this matrix explicitly for the example network in Fig. 3.1,

$$\mathsf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Using this matrix, we can write equation (3.1) for the PageRank $\mathbf{r}$ in matrix form,

$$\mathbf{r}^T = \mathbf{r}^T \mathsf{Q} \tag{3.3}$$

which, after transposing both sides, gives $\mathsf{Q}^T \mathbf{r} = \mathbf{r}$. We can see this is an eigenproblem for an eigenvalue that is equal to one. The matrix Q is "column-stochastic" (i.e., a square matrix whose elements are positive, its columns sum to one on each row and that is also irreducible; see Wikipedia for the theorem, reproduced again below, and for stochastic matrices). Such a matrix is indeed known to have a largest eigenvalue that is equal to one, according to the Perron-Frobenius theorem. In order to calculate the eigenvector corresponding to the unit eigenvalue, which gives the PageRank of each web page, we turn this eigenproblem into an iteration scheme for calculating iterate $k+1$ from iterate k,

$$\mathbf{r}_{k+1} = \mathsf{Q}^T \mathbf{r}_k.$$

We will see below that repeating this iteration leads to an improving approximation for the eigenvector representing the PageRank. But first, some refinements to the matrix Q are needed.

Theorem 3.2.1 — Perron-Frobenius. Assuming an irreducible matrix, T, the theorem states that:

1. T has a positive (real) eigenvalue $\lambda_{\max}$ such that all other eigenvalues of T satisfy,
$$|\lambda| \leq \lambda_{\max}.$$
2. Furthermore, $\lambda_{\max}$ has algebraic and geometric multiplicity of one, and has an eigenvector $\mathbf{x}$ with $\mathbf{x} > 0$.
3. Any non-negative eigenvector is a multiple of $\mathbf{x}$
4. More generally, if $\mathbf{y} \geq 0$, $\mathbf{y} \neq 0$ is a vector and μ is a number such that
$$T\mathbf{y} \leq \mu\mathbf{y}$$
then
$$y > 0, \quad \text{and } \mu \geq \lambda_{\max}$$
with $\mu = \lambda_{\max}$ if and only if $\mathbf{y}$ is a multiple of $\mathbf{x}$
5. If $0 \leq \mathsf{S} \leq \mathsf{T}, \mathsf{S} \neq \mathsf{T}$ then every eigenvalue σ of S satisfies
$$|\sigma| < \lambda_{\max}.$$
In particular, all the diagonal minors $\mathsf{T}_{(i)}$ obtained from T by deleting the ith row and column have eigenvalues all of which have absolute value $< \lambda_{\max}$.
6. If T is primitive, then all other eigenvalues of T satisfy
$$|\lambda| < \lambda_{\max}$$

3.2.3 Refinements to the matrix model

To explain the needed refinements to the PageRank algorithm, consider an alternative interpretation for it. Imagine a random walker surfing from page to page following the links. When more than one outgoing link is available, the walker chooses among them randomly. Define now the importance (PageRank) of a page to be the probability that a random walker is at that page; the random walker is more likely to end up at a page with more in-links, consistent with our previous definition. Consider some issues the random walker may run into.

Stuck at Node: The first issue presents itself at page (node) 1 in the example of Figure 3.1. Once arriving at page 1, the walker will get stuck there because Page 1 has no outlinks. To resolve this, allow the random walker arriving at this page to jump to any random page with equal probability. Mathematically, this is done by defining a new matrix,

$$\hat{\mathsf{Q}} = \mathsf{Q} + \frac{1}{n}\mathbf{e}\mathbf{d}^T,$$

where $\mathbf{d}$ is a column vector of 1s and $\mathbf{e}$ is a column vector with 1 in each entry corresponding to

rows in Q which are all zeros. In the above specific example, define **d** and **e** as,

$$\mathbf{d} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \qquad \mathbf{e} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

and $\hat{\mathsf{Q}}$ becomes

$$\hat{\mathsf{Q}} = \mathsf{Q} + \frac{1}{7} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Note that this addition changes a line of zeros in the transition matrix to a line whose sum of columns is one, as required by the Perron-Frobenius theorem.

Stuck at Subgraph: The second potential problem is the random walker getting stuck in a subgraph. In Figure 3.1 the network can be divided into two subgraphs: pages {1,2,3} and pages {4,5,6,7}. If the random walker moves to page 4 from page 2, there is zero probability that the walker will return to pages {1,2,3} – the walker is stuck in subgraph {4,5,6,7}. To resolve this issue, allow the random walker to jump from any site to any other site with some small probability ("teleportation"), at any time. This is done by further adjusting the transition matrix, defining $\hat{\hat{Q}}$ (and thus making the transition matrix "irreducible", again required by the theorem),

$$\begin{aligned} \hat{\hat{\mathsf{Q}}} &= \alpha \hat{\mathsf{Q}} + (1-\alpha)\frac{1}{n}\mathbf{d}\mathbf{d}^T \\ &= \alpha \mathsf{Q} + \alpha \frac{1}{n}\mathbf{e}\mathbf{d}^T + (1-\alpha)\frac{1}{n}\mathbf{d}\mathbf{d}^T \end{aligned} \tag{3.4}$$

with **d** defined again as a column of 1s, and α as the teleportation factor, chosen to be $\alpha \approx 0.85$. Note that $\mathbf{d}\mathbf{d}^T$ is a matrix whose elements are all 1. The smaller the α, the faster the convergence of the iterations for finding the PageRank is, as it determines the magnitude of the second-largest eigenvalue of the matrix (see next section for the explanation of how the convergence rate is determined from these eigenvalues). However, a small α also lowers the contribution of the

transition matrix Q which represents the network. In this example, calculate $\hat{\hat{\mathrm{Q}}}$ to be,

$$\hat{\hat{\mathrm{Q}}} = 0.85 \begin{bmatrix} \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} + (1-0.85)\frac{1}{7} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{32}{105} & \frac{3}{140} & \frac{32}{105} & \frac{32}{105} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} \\ \frac{25}{56} & \frac{25}{56} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} \\ \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{32}{105} & \frac{32}{105} & \frac{32}{105} \\ \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{61}{70} & \frac{3}{140} \\ \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{25}{56} & \frac{25}{56} & \frac{3}{140} & \frac{3}{140} \\ \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{3}{140} & \frac{61}{70} & \frac{3}{140} & \frac{3}{140} \end{bmatrix}$$

3.2.4 Calculating PageRank using the power method

Several variants of the "power method" for efficiently calculating eigenvalues and eigenvectors of large matrices will be covered in the next section, and one is demonstrated here for the case of calculating the PageRank for an $n \times n$ transition matrix. The steps are,

1. Initialize an $n \times 1$, non-zero starting vector $\mathbf{x}_0$ either by ones or by random numbers.
2. Calculate the next iterate: $\hat{\mathbf{x}}_k = \hat{\hat{\mathrm{Q}}}^T \mathbf{x}_{k-1}$
3. Normalize: $\mathbf{x}_k = \hat{\mathbf{x}}_k / \|\hat{\mathbf{x}}_k\|$
4. Iterate for $k = 0, 1, 2, \ldots$ until the solution converges (changes between iteration results become sufficiently small) indicating that an appropriate approximation of the desired eigenvector was found.

As for why the algorithm works and when may it fail, see the next section.

3.3 The power method

The objective of the simplest power method is to calculate the largest eigenvalue (in absolute value) and the corresponding eigenvector of a matrix. For large, and in particular large and sparse matrices, this method is especially efficient. As a result, it is very appropriate when all we need is the largest or smallest few eigenvalues/vectors, as when solving for a network's PageRank.

3.3.1 Gram-Schmidt orthogonalization

Reminder: In an orthogonal basis, vectors are orthogonal to one another (that is for any two vectors $\mathbf{q}_1$ and $\mathbf{q}_2$, $\mathbf{q}_1 \cdot \mathbf{q}_2 = \mathbf{q}_1^T \mathbf{q}_2 = 0$). The basis is orthonormal if the vectors are also normalized to one,

$$\mathbf{q}_i^T \mathbf{q}_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

To make a basis orthonormal, use the Gram-Schmidt orthogonalization briefly summarized as follows. Consider three independent vectors $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$. We are looking for three orthonormal vectors

$\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$ that span the same space. Start by normalizing $\mathbf{a}_1$,

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|}.$$

Next, define a new vector $\mathbf{A}_2$ that is orthogonal to $\mathbf{q}_1$, by subtracting any component of $\mathbf{a}_2$ that is in the direction of $\mathbf{q}_1$. Then we normalize $\mathbf{A}_2$ to get $\mathbf{q}_2$,

$$\mathbf{A}_2 = \mathbf{a}_2 - (\mathbf{q}_1^T \mathbf{a}_2)\mathbf{q}_1, \qquad \mathbf{q}_2 = \frac{\mathbf{A}_2}{\|\mathbf{A}_2\|}.$$

Finally, repeat the procedure for $\mathbf{q_3}$ removing any components of $\mathbf{a}_3$ in the direction of $\mathbf{q}_1$ and $\mathbf{q}_2$ and normalize,

$$\mathbf{A}_3 = \mathbf{a}_3 - (\mathbf{q}_1^T \mathbf{a}_3)\mathbf{q}_1 - (\mathbf{q}_2^T \mathbf{a}_3)\mathbf{q}_2, \qquad \mathbf{q}_3 = \frac{\mathbf{A}_3}{\|\mathbf{A}_3\|}.$$

It will be useful later to write this Gram-Schmidt process as a matrix decomposition, such that,

$$\begin{aligned} \mathsf{A} = (\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3) &= (\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3) \begin{pmatrix} \mathbf{q}_1^T \mathbf{a}_1 & \mathbf{q}_1^T \mathbf{a}_2 & \mathbf{q}_1^T \mathbf{a}_3 \\ 0 & \mathbf{q}_2^T \mathbf{a}_2 & \mathbf{q}_2^T \mathbf{a}_3 \\ 0 & 0 & \mathbf{q}_3^T \mathbf{a}_3 \end{pmatrix} \\ &= \mathsf{QR}. \end{aligned} \tag{3.5}$$

3.3.2 The power method: calculating the largest eigenvalue/vector

The simplest power method attempts to calculate the largest (in absolute value) eigenvalue and the corresponding eigenvector of a matrix Q. The eigenproblem $\mathsf{Q}\mathbf{e} = \lambda \mathbf{e}$ is used to create an iteration scheme, calculating the $k+1$ approximation for $\mathbf{e}$ from the kth approximation as follows. Start from an initial guess, $\mathbf{v}_0$, then iterate and normalize at each step,

$$\begin{aligned} \hat{\mathbf{v}}_{k+1} &= \mathsf{Q}\mathbf{v}_k \\ \mathbf{v}_{k+1} &= \hat{\mathbf{v}}_{k+1} / \|\hat{\mathbf{v}}_{k+1}\|. \end{aligned}$$

If the matrix is sparse, the multiplication by a vector is inexpensive, making this an efficient scheme. As for why this works, consider the eigenvectors $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ of the matrix Q, expand the initial guess in terms of these eigenvectors, and consider the next iterations ignoring the normalization step for a moment,

$$\begin{aligned} \mathbf{v}_0 &= c_1 \mathbf{e}_1 + \ldots + c_{n-1}\mathbf{e}_{n-1} + c_n \mathbf{e}_n \\ \mathbf{v}_1 = \mathsf{Q}\mathbf{v}_0 &= c_1 \lambda_1 \mathbf{e}_1 + \ldots + c_{n-1}\lambda_{n-1}\mathbf{e}_{n-1} + c_n \lambda_n \mathbf{e}_n \\ \mathbf{v}_k &= c_1 \lambda_1^k \mathbf{e}_1 + \ldots + c_{n-1}\lambda_{n-1}^k \mathbf{e}_{n-1} + c_n \lambda_n^k \mathbf{e}_n. \end{aligned}$$

Assuming the eigenvalues are arranged by their absolute value magnitude such that λ_n is the largest, $|\lambda_1| \leq |\lambda_2| \leq \ldots \leq |\lambda_{n-1}| < |\lambda_n|$, the last term in the above sum dominates as k increases until eventually $\mathbf{v}_k \approx c_n \lambda_n^k \mathbf{e}_n$. Adding the normalization, we find $\mathbf{v}_k \approx \mathbf{e}_n$. The largest eigenvalue is then calculated from the ratio of, say, the first element of $\mathbf{v}_k$ and $\mathbf{v}_{k+1}$.

We can now understand why the above scheme may fail . First, if the initial guess $\mathbf{v}_0$ does not project on $\mathbf{e}_n$, the scheme cannot work because the desired eigenvalue must be represented in the expansion of the initial conditions. Next, the convergence to the last eigenvector depends on the ratio $|\lambda_{n-1}|/|\lambda_n|$. If this ratio is close to 1, convergence is very slow as the second to last term is not becoming smaller than the last term sufficiently rapidly, and if it equals 1 convergence may not even occur. If $\lambda_{n-1} = \lambda_n$, the iterations seem to converge, but $\mathbf{v}_k$ will contain contributions from these the two eigenvectors corresponding to the two largest eigenvalues. These problems are resolved using the block power method introduced below. If $\lambda_{n-1} = -\lambda_n$, the iterations will not converge as the contribution of one of the corresponding two eigenvectors with a negative eigenvalue keeps changing sign, cancelling out the dominant terms.

3.3.3 Block power method

Expanding on the basic power method, the block power method allows us to calculate the largest p eigenvalues/eigenvectors of a matrix A. Start by initializing p orthonormal vectors with random numbers, placing them in an $n \times p$ matrix U_0 and multiplying them by A. We then orthonormalize the resulting matrix using Gram-Schmidt, and proceed to the next iteration,

$$\begin{aligned} \hat{\mathsf{U}}_{k+1} &= \mathsf{A}\mathsf{U}_k \\ \mathsf{U}_{k+1} &= \text{Gram-Schmidt of } \hat{\mathsf{U}}_{k+1} \end{aligned}$$

The reason this approach converges is essentially identical to that of the regular power method, and the rate of convergence is determined in this case by the ratio $|\lambda_{n-p}|/|\lambda_{n-p+1}|$. The block power method should work only for normal matrices, whose eigenvectors are orthogonal, otherwise the orthogonalization step destroys the structure of the eigenvectors. If the matrix is non-normal, the iterations may lead to a mixing among the non-orthogonal eigenvectors. An example showing the convergence of the iteration in a simple case is shown in Fig. 3.2.

Demos: Calculating the largest p eigenvalues/vectors using the block power method, and demonstrating different scenarios in which the block power method fails. These scenarios are important to understand when using Matlab/python to calculate a few largest/smallest eigenmodes: demo;

3.3.4 Inverse power method

Finally, the inverse power method allows to calculate the smallest eigenvalue and corresponding eigenvector for a matrix Q. Use the inverse of Q for the iterative scheme, using the fact that $\mathsf{Q}\mathbf{e}_k = \lambda_k \mathbf{e}_k$ may be written as $\mathsf{Q}^{-1}\mathbf{e}_k = \lambda_k^{-1}\mathbf{e}_k$, to derive the iteration scheme,

$$\begin{aligned} \hat{\mathbf{v}}_{k+1} &= \mathsf{Q}^{-1}\mathbf{v}_k \\ \mathbf{v}_{k+1} &= \hat{\mathbf{v}}_{k+1}/\|\hat{\mathbf{v}}_{k+1}\|. \end{aligned}$$

In practice, calculating the inverse Q^{-1} is undesired as it may lead to numerical issues for large and noisy matrices that are not well-conditioned. Instead, multiply the above iteration scheme by Q on both side to find $\mathsf{Q}\hat{\mathbf{v}}_{k+1} = \mathbf{v}_k$. This is seen as a set of linear equations that needs to be solved for $\hat{\mathbf{v}}_{k+1}$. As iterations proceed, the dominant term in $\mathbf{v}_k$ will be the one involving λ_1^{-1} where λ_1 is the smallest eigenvalue of Q, and the corresponding eigenvector $\mathbf{e}_1$. The inverse power method converges to the correct smallest eigenvalue/vector only if $|\lambda_1| < |\lambda_2|$. It is now simple to generalize this to an inverse block power method for calculating the smallest p eigenvalues and corresponding eigenvectors.

You are encouraged to read about the more efficient "shifted power method" for calculating eigenvectors/values in Strang §7.3 pp 396-397.

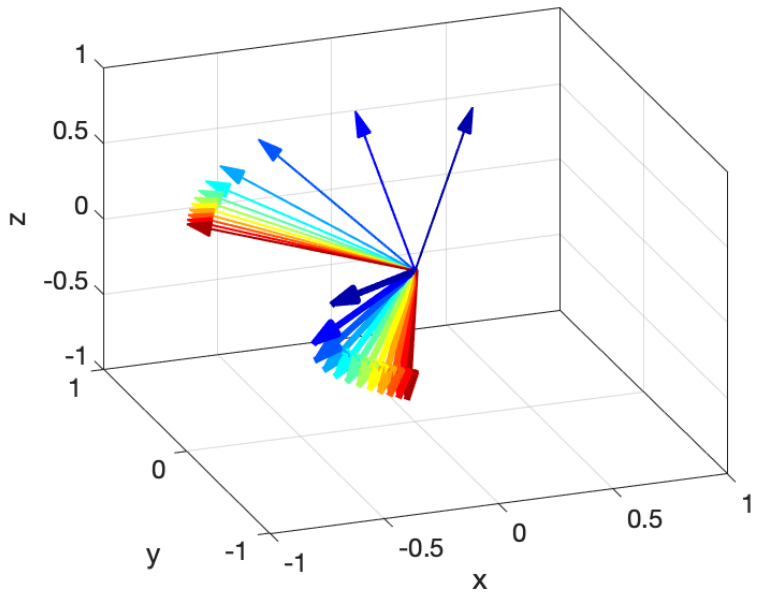


Figure 3.2: Iterations of the block power method, calculating the two eigenvectors corresponding to the two largest eigenvalues of a 3×3 matrix. The colors from blue to red denote progressing iterations.

3.4 Spectral clustering (partitioning) of networks

Eigenvectors can also be used to partition networks. An example would be a social network, where different individuals are linked (friended), forming groups, and the objective of the analysis is to identify these groups. This is a special case of the more general spectral clustering algorithm to be discussed later in the course. In order to get an insight into how the algorithm works, consider first the "network", with four nodes, connected in pairs as shown in Fig. 3.3. Note that the network is non-directed, as there is no direction to the links between two nodes, unlike the case analyzed in the discussion of the PageRank algorithm. Consider a vector $\mathbf{x} = (x_1, x_2, x_3, x_4)^T$ corresponding to the four nodes, which we would like to be the classification measure, such that if the network is divided into two parts, the two corresponding groups of elements of $\mathbf{x}$ are positive and negative, respectively.

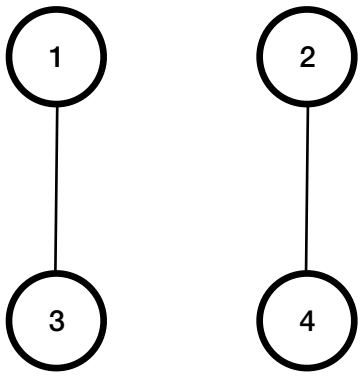


Figure 3.3: an example network

We first define an Adjacency matrix, A, to have an element at the (i, j) location if nodes i and j are

connected. The Degree matrix, D, is a diagonal matrix that sums the rows of the Adjacency matrix, A, and the Laplacian matrix is defined as $\mathsf{L} = \mathsf{D} - \mathsf{A}$. For the above example,

$$\mathsf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad \mathsf{D} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathsf{L} = \mathsf{D} - \mathsf{A} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}.$$

The partition (clustering) vector $\mathbf{x}$ turns out to be the eigenvector corresponding to the second smallest eigenvalue of L. To see this, consider the scalar quadratic form $\mathbf{x}^T \mathsf{L} \mathbf{x}$, which is seen to be in this case,

$$\mathbf{x}^T \mathsf{L} \mathbf{x} = (x_1 - x_3)^2 + (x_2 - x_4)^2.$$

This is precisely the sum of differences corresponding to connected pairs of nodes and is always the case – for a proof, see the spectral clustering chapter of the course to see that this is the basis for the classification algorithm. If we find $\mathbf{x}$ which minimizes this quadratic form, we expect in the above example x_1 and x_3 to be as similar to each other as possible, and x_2 and x_4 to be similar as well, such that linked pairs have similar magnitudes.

Let's consider how to find the minimum of $\mathbf{x}^T \mathsf{L} \mathbf{x}$. First, this needs to be a constrained minimization, by requiring, for example, that the magnitude of $\mathbf{x}$ is one, or else the minimum would just be $\mathbf{x} = 0$. Second, remembering that we wish to use the $\mathbf{x}$ for classification of the network into two groups based on their signs, it makes sense to require that the mean of their values is zero, forcing the elements to have both positive and negative signs. So the constrained optimization problem becomes,

$$\begin{aligned} &\text{find } \min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathsf{L} \mathbf{x} \\ &\text{subject to } \frac{1}{2} \mathbf{x}^T \mathbf{x} = 1 \\ &\text{and } \mathbf{x}^T \mathbf{1} = 0, \end{aligned}$$

where $\mathbf{1} = (1,1,1,1)$ in our example. Ignoring the constraint $\mathbf{x}^T \mathbf{1} = 0$ for a moment, we write this constrained optimization using Lagrange multipliers,

$$\min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathsf{L} \mathbf{x} + \lambda (1 - \frac{1}{2} \mathbf{x}^T \mathbf{x}) = \frac{1}{2} \sum_{ij} x_i \mathsf{L}_{ij} x_j + \lambda \left(1 - \frac{1}{2} \sum_i x_i^2 \right).$$

At the minimum, the derivative with respect to any component x_p should vanish,

$$\begin{aligned} 0 &= \frac{d}{dx_p} \left[\frac{1}{2} \sum_{ij} x_i \mathsf{L}_{ij} x_j + \lambda \left(1 - \frac{1}{2} \sum_i x_i^2 \right) \right] \\ &= \frac{1}{2} \sum_i x_i \mathsf{L}_{ip} + \frac{1}{2} \sum_j \mathsf{L}_{pj} x_j - \lambda x_p = \sum_i x_i \mathsf{L}_{ip} - \lambda x_p = \mathsf{L} \mathbf{x} - \lambda \mathbf{x}, \end{aligned}$$

using the fact that the matrix L is symmetric in the last step. From the above, we conclude that

$$\mathsf{L} \mathbf{x} = \lambda \mathbf{x}$$

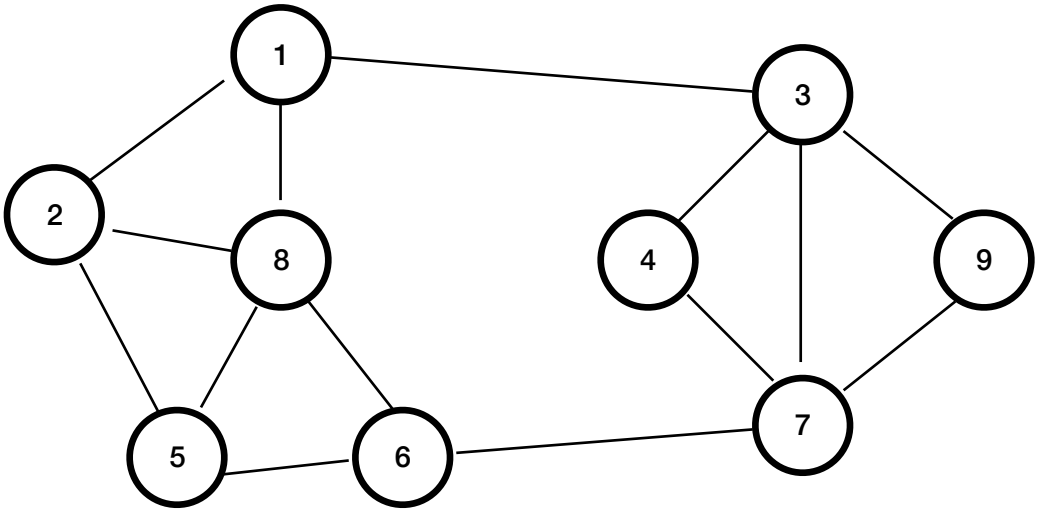


Figure 3.4: Network for demonstrating spectral partitioning.

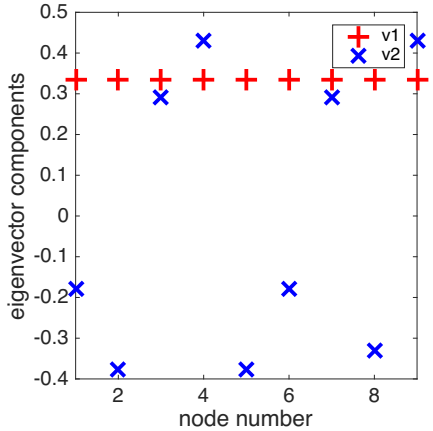
so that the $\mathbf{x}$ that minimizes the quadratic form is an eigenvector of the Laplacian matrix L. To see which of the eigenvectors, consider the quadratic form again,

$$\mathbf{x}^T \mathsf{L}\mathbf{x} = \mathbf{x}^T \lambda \mathbf{x} = \lambda \mathbf{x}^T \mathbf{x} = \lambda .$$

We are therefore looking for the eigenvector whose eigenvalue is as small as possible, but non-zero. Because the smallest eigenvalue is zero and it corresponds to a vector of 1s. We required above that $\mathbf{x}$ be perpendicular to the vector of 1s, and therefore we choose $\mathbf{x}$ to be the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix.

As an example, consider the network shown in Fig. 3.4. The adjacency matrix and the first and second eigenvectors of the corresponding Laplacian matrix are shown here, showing that the second eigenvector clearly well-classifies this particular network.

$$\mathsf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



(Optional) Finally, why is the matrix called a Laplacian matrix? If we derive the Laplacian matrix for a network made of a one dimensional line of connected nodes, it turns out to be the same as the finite difference representation of a second derivative. Consider a function $y(x)$, represented on a grid $x_i = 0, \Delta x, 2\Delta x, \ldots$ The second derivative of $y(x)$ at a point $x_i = i\Delta x$ is approximated as

$y''(x) = \partial y/\partial x^2 \approx (y_{i+1} - 2y_i + y_{i-1})/\Delta x^2$. The vector of second derivatives at all grid points may be written in matrix form as $\mathsf{L}(y_1, \ldots, y_n)^T$, where the matrix L has -2 on the diagonal, and 1 on the two diagonals on either side. The Laplacian matrix for the above mentioned one-dimensional connected series of nodes has the same shape. Now, consider the corresponding "Sturm-Liouville" problem $\mathsf{L}\mathbf{y} = \lambda\mathbf{y}$, or in a differential equation form, $y'' + \lambda y = 0$ with $y'(0) = y'(2\pi) = 0$. The solutions (eigenfunctions) are $y_n(x) = \cos(nx)$, $\lambda_n = n$. They become more oscillatory for higher n, corresponding to larger eigenvalues. Therefore, the eigenvectors corresponding to smaller eigenvalues correspond to a coarser division of the network into sub-components. This provides some further intuition into the choice of the second smallest eigenvector for the partitioning the network into two parts.

See demo network_classification_example.m/py.

3.5 Generalized eigenvalue problems

Generalized eigenproblems of the form $\mathsf{A}\mathbf{x} = \lambda\mathsf{B}\mathbf{x}$ arise in both the solution of partial differential equations and in classification problems (see later in the course). In order to solve them, we could multiply B^{-1} to obtain a standard eigenproblem $\mathsf{B}^{-1}\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$ and solve for $\mathbf{x}$ and λ. However, if A, B are symmetric, it is not a good idea to because $\mathsf{B}^{-1}\mathsf{A}$ is not necessarily symmetric, and we lose a very helpful property of the original problem. Instead, it is best to transform the generalized problem to a symmetric regular eigenvalue problem using "Cholesky decomposition", $\mathsf{B} = \mathsf{R}\mathsf{R}^\dagger$ where $\dagger$ indicates complex conjugate transpose, or just $\mathsf{B} = \mathsf{R}\mathsf{R}^T$ in the case of real matrices. To implement this procedure, we carry out the following steps.

1. First, write the LU decomposition

   $$\mathsf{B} = \mathsf{L}_1\mathsf{U}_1$$

   as a $\mathsf{B} = \mathsf{L}\mathsf{D}\mathsf{U}$ decomposition, where D is diagonal and both L and U now have 1 on their diagonal, not only on the diagonal of L as in the standard LU decomposition. This is done by calculating the usual $\mathsf{B} = \mathsf{L}_1\mathsf{U}_1$, then setting D to be the diagonal of U_1, and calculating $\mathsf{U} = \mathsf{D}^{-1}\mathsf{U}_1$ (in Matlab: `U=D\U1`). The inverse of D is trivial because it is diagonal. Note that the LDU decomposition of a symmetric matrix B is

   $$\mathsf{B} = \mathsf{L}\mathsf{D}\mathsf{L}^T.$$

   To see that this is the case (as shown in Strang **1N** p 57), write $\mathsf{B} = \mathsf{L}\mathsf{D}\mathsf{U}$ so $\mathsf{B}^T = \mathsf{U}^T\mathsf{D}\mathsf{L}^T$ which is a lower×diagonal×upper decomposition. However, $\mathsf{B} = \mathsf{B}^T$ so also $\mathsf{B}^T = \mathsf{L}\mathsf{D}\mathsf{U}$. Because $\mathsf{L}\mathsf{D}\mathsf{U}$ is unique $\mathsf{L}^T = \mathsf{U}$ and that $\mathsf{B} = \mathsf{L}\mathsf{D}\mathsf{L}^T$.
2. Next, write this as a Cholesky decomposition $\mathsf{B} = \mathsf{R}\mathsf{R}^T$ with $\mathsf{R} = \mathsf{L}\sqrt{\mathsf{D}}$.
3. Transform the generalized eigenproblem as follows. Write $\mathsf{A}\mathbf{x} = \lambda\mathsf{B}\mathbf{x}$ as

   $$\mathsf{A}\mathbf{x} = \lambda\mathsf{R}\mathsf{R}^T\mathbf{x},$$

   multiply on left by R^{-1} to find $\mathsf{R}^{-1}\mathsf{A}\mathbf{x} = \lambda\mathsf{R}^T\mathbf{x}$, or $\mathsf{R}^{-1}\mathsf{A}(\mathsf{R}^T)^{-1}\mathsf{R}^T\mathbf{x} = \lambda\mathsf{R}^T\mathbf{x}$; define $\mathbf{y} = \mathsf{R}^T\mathbf{x}$ and

   $$\mathsf{A}' = \mathsf{R}^{-1}\mathsf{A}(\mathsf{R}^T)^{-1} = \mathsf{R}^{-1}\mathsf{A}(\mathsf{R}^{-1})^T.$$

   We have transformed to a standard eigenproblem $\mathsf{A}'\mathbf{y} = \lambda\mathbf{y}$ where A' is symmetric.
4. We can now solve for $\mathbf{y}$ and can then efficiently solve $\mathbf{y} = \mathsf{R}^T\mathbf{x}$ for $\mathbf{x}$.

For a demo, consider Generalized_eigenvalue_problem.m/py.

3.6 Linear ordinary differential equations and matrix exponentiation

Eigenvalues and eigenvectors are essential for solving systems of linear, constant-coefficient ordinary differential equations which arise in numerous applications. Consider, for example, the equations,

$$\frac{dx_1}{dt} = 4x_1 - x_2$$
$$\frac{dx_2}{dt} = 2x_1 + x_2$$

with initial conditions $x_1(t=0) = 1$, $x_2(t=0) = -2$. We can write these equations in matrix form,

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} 4 & -1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \mathbf{x}(t=0) = \begin{bmatrix} 1 \\ -2 \end{bmatrix}.$$

In general, given a set of linear ODEs written in matrix form, $d\mathbf{x}/dt = \mathsf{A}\mathbf{x}$, substituting into this equation a solution assuming an exponential time dependence, $\mathbf{x} = \mathbf{x}_0 e^{\lambda t}$, we find $d\mathbf{x}/dt = \lambda \mathbf{x}_0 e^{\lambda t} = \mathsf{A}\mathbf{x}_0 e^{\lambda t}$, or $\mathsf{A}\mathbf{x}_0 = \lambda \mathbf{x}_0$. Therefore, the constant λ determining the time dependence is an eigenvalue of A and the amplitude $\mathbf{x}_0$ is the corresponding eigenvector. Note that there are several eigenvalues and eigenvectors, and therefore several possible solutions. Given that if two functions solve these linear equations, so too does their sum, and thus the general solution is a sum over these eigen-solutions,

$$\mathbf{x}(t) = \sum_{i=1}^{n} c_i \mathbf{e}_i e^{\lambda_i t}, \tag{3.6}$$

where $\{\mathbf{e}_i, \lambda_i\}$ are the eigenvector, eigenvalue pairs, and the constants c_i are determined from the initial conditions. At $t = 0$, this solution may be written in matrix form as $\mathbf{x}(t=0) = \mathbf{x}_0 = \mathsf{S}\mathbf{c}$, where the vector $\mathbf{c}$ contains the constants c_i, and S is a matrix whose columns contain the eigenvectors $\mathbf{e}_i$. Thus, the required coefficients are given by the initial conditions via $\mathbf{c} = \mathsf{S}^{-1}\mathbf{x}_0$. The solution (3.6) may now be written in matrix form as,

$$\mathbf{x}(t) = \mathsf{S} \begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} \mathsf{S}^{-1}\mathbf{x}_0. \tag{3.7}$$

For the above example, the eigenvalues and vectors are $\lambda_1 = 2, \mathbf{v}_1 = \begin{bmatrix} 0.4472 & 0.8944 \end{bmatrix}^T$ and $\lambda_2 = 3, \mathbf{v}_2 = \begin{bmatrix} 0.7071 & 0.7071 \end{bmatrix}^T$. The solution to this system of differential equations is, therefore,

$$\mathbf{x}(t) = c_1 e^{\lambda_1 t}\mathbf{e}_1 + c_2 e^{\lambda_2 t}\mathbf{e}_2$$

We can write this solution in matrix form using (3.7),

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 0.4472 & 0.7071 \\ 0.8944 & 0.7071 \end{bmatrix} \begin{bmatrix} e^{2t} & 0 \\ 0 & e^{3t} \end{bmatrix} \begin{bmatrix} 0.4472 & 0.7071 \\ 0.8944 & 0.7071 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ -2 \end{bmatrix}.$$

Note that the eigenvalues and eigenvectors may be complex, but if the initial conditions and the matrix are real, the solution must also be real. This is achieved by calculating appropriate coefficients c_i. In the case of complex eigenvalues, writing $\lambda = \lambda_R \pm i\lambda_I$ and eigenvectors $\mathbf{e} = \mathbf{e}_R \pm i\mathbf{e}_I$, and using $e^{\lambda t} = e^{\lambda_R t}(\cos(\lambda_I t) + i\sin(\lambda_I t))$, the solution may be written as,

$$\mathbf{x}(t) = c_1 e^{\lambda_R t}(\mathbf{e}_R \cos\lambda_I t - \mathbf{e}_I \sin\lambda_I t) + c_2 i e^{\lambda_R t}(\mathbf{e}_I \cos\lambda_I t + \mathbf{e}_R \sin\lambda_I t)$$

The above discussion assumes that the matrix has n independent eigenvectors. If that's not the case, some adjustments are needed. See the discussion of the Jordan form of a matrix in Section 3.8.

3.6.1 Higher order, linear, constant coefficient ODEs

So far we have dealt with a system of first order constant-coefficient linear differential equations. A higher order system of constant-coefficient ODEs can be transformed to a set of first order ODEs and solved using the above approach. Consider a second order, homogeneous, constant coefficient ODE,

$$\ddot{x} = a\dot{x} + bx,$$

and define $y = \dot{x}$, writing the above equation instead as a set of two first order ODEs,

$$\begin{aligned}\dot{y} &= ay + bx \\ \dot{x} &= y.\end{aligned}$$

Similarly, a set of n linear, constants coefficient ODEs can be converted to a single nth order ODE, assuming no pathologies.

3.6.2 Matrix exponentiation

The above solution (3.7) for a set of linear, constant coefficient ODEs can be elegantly written in terms of the exponent of the matrix A. Following the formula for the Taylor expansion of e^x, the exponent of a matrix A is defined as,

$$e^{\mathsf{A}} = \mathsf{I} + \mathsf{A} + \frac{\mathsf{A}^2}{2!} + \frac{\mathsf{A}^3}{3!} + \cdots \tag{3.8}$$

Assuming we can diagonalize A using its eigenvectors as $\mathsf{A} = \mathsf{S}\Lambda\mathsf{S}^{-1}$, where Λ is a diagonal matrix containing the eigenvalues, we can plug this formula into the equation for the matrix exponential and we find that

$$\begin{aligned} e^{\mathsf{A}t} &= \mathsf{I} + \mathsf{A}t + \frac{\mathsf{A}^2t^2}{2!} + \frac{\mathsf{A}^3t^3}{3!} + \cdots && (3.9)\\ &= \mathsf{I} + \mathsf{S}\Lambda\mathsf{S}^{-1}t + \frac{(\mathsf{S}\Lambda^2\mathsf{S}^{-1}t)(\mathsf{S}\Lambda^2\mathsf{S}^{-1}t)}{2!} + \frac{(\mathsf{S}\Lambda^2\mathsf{S}^{-1}t)^3}{3!} + \ldots \\ &= \mathsf{I} + \mathsf{S}\Lambda\mathsf{S}^{-1}t + \frac{\mathsf{S}\Lambda^2\mathsf{S}^{-1}t^2}{2!} + \frac{\mathsf{S}\Lambda^3\mathsf{S}^{-1}t^3}{3!} + \ldots \\ &= \mathsf{S}(\mathsf{I} + \Lambda t + \frac{(\Lambda t)^2}{2!} + \frac{(\Lambda t)^3}{3!} + \ldots)\mathsf{S}^{-1} = \mathsf{S}e^{\Lambda t}\mathsf{S}^{-1}. && (3.10)\end{aligned}$$

Because Λt is diagonal, its exponent is a diagonal matrix with $e^{\lambda_i t}$ along the diagonal. Thus, solution (3.6) to the ODEs may be written using (3.10) and (3.7) as,

$$\mathbf{x}(t) = e^{\mathsf{A}t}\mathbf{x}_0. \tag{3.11}$$

It is not difficult to see from the first line of the expression (3.10) for $e^{\mathsf{A}t}$ that its derivative is given by

$$\frac{d}{dt}e^{\mathsf{A}t} = \mathsf{A}e^{\mathsf{A}t},$$

and from there,

$$\frac{d\mathbf{x}(t)}{dt} = \frac{d}{dt}e^{\mathsf{A}t}\mathbf{x}_0 = \mathsf{A}e^{\mathsf{A}t}\mathbf{x}_0 = \mathsf{A}\mathbf{x}(t),$$

so that (3.11) indeed solves the equation. In Matlab/python, `expm(A)` gives the exponent of a matrix A.

3.6.3 Stability of solutions to linear ODEs

In applications it is important to know if the solution to a set of linear ODEs grows to infinity, decays to zero, oscillates, etc. This behavior depends on the nature of the eigenvalues. For linear homogeneous ODEs, $\mathbf{x}(t) = \mathbf{0}$ is always an equilibrium solution: starting there, the solution to $d\mathbf{x}/dt = \mathsf{A}\mathbf{x}$ will remain at $\mathbf{x} = \mathbf{0}$. The question, though, is whether the solution will return to this equilibrium point if perturbed slightly – this is referred to as the stability of the equilibrium point. Using the eigenvalues we can determine whether the origin is a stable or unstable equilibrium point, and also determine in which way the solution either returns to the stable origin or departs from the unstable origin. The eigenvectors and eigenvalues are also used to construct graphical phase portraits of the system of ODEs, providing further insight into the stability.

To determine the stability, write each eigenvalue in the form: $\lambda = \lambda_R + i\lambda_I$, and from this make the following conclusions about the behavior of the solution,

- If all eigenvalues have a negative real part, the solution is stable, and decays to zero.
- If one or more of the eigenvalues have a positive real part, the solution is unstable and a small perturbation away from zero will grow to infinity.
- If one or more of the eigenvalues has a non-zero imaginary part, the solution will oscillate.

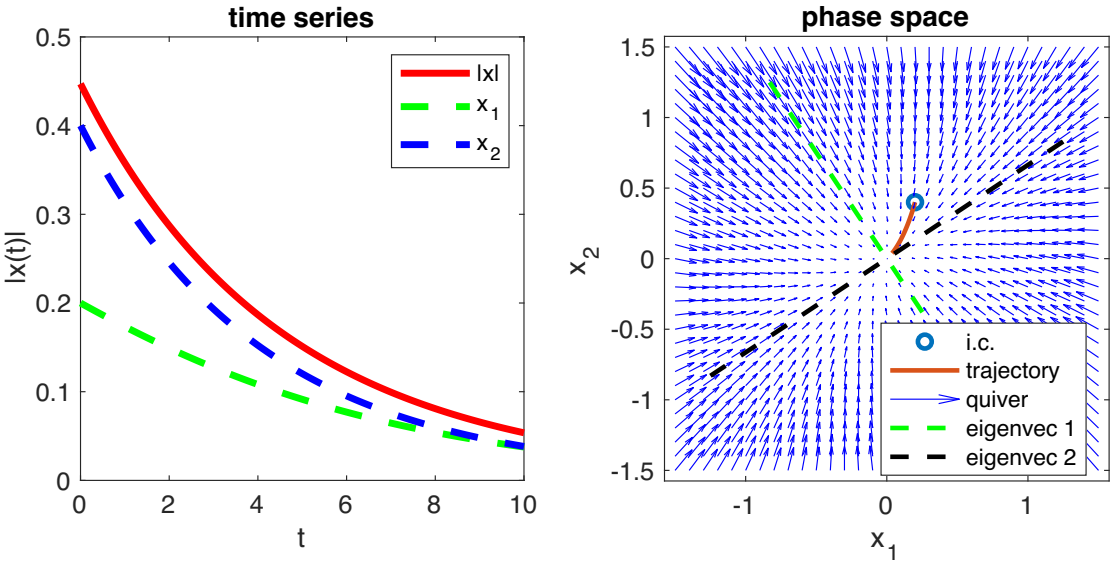
It is helpful to plot and analyze the solution to a set of two dimensional ODEs in a "phase space" whose coordinates are the variables. That is, given $d\mathbf{x}/dt = \mathsf{A}\mathbf{x}$ with $\mathbf{x}(t) = (x_1(t), x_2(t))^T$, the coordinates will be (x_1, x_2). At each point in this phase space, we can draw an arrow pointing in the direction of $(\dot{x}_1, \dot{x}_2)$, with the corresponding magnitude. Similarly, the solution $\mathbf{x}(t)$ corresponds to a curve in this phase space, emanating from the initial conditions. In addition, the directions of the two eigenvectors (when they are real) are also shown by the dash black and green lines.

Consider the following two-dimensional examples of ODEs in the form $d\mathbf{x}/dt = \mathsf{A}\mathbf{x}$. For each example, we provide the matrix A, its eigenvalues λ_i and its eigenvectors as the columns of a matrix S.

Real eigenvalues, decaying solution: Consider

$$A = \begin{bmatrix} -0.2306 & 0.0461 \\ 0.0461 & -0.2694 \end{bmatrix}; \quad \lambda_i = -3, -2; \quad S = \begin{bmatrix} -0.5532 & -0.8330 \\ 0.8330 & -0.5532 \end{bmatrix}.$$

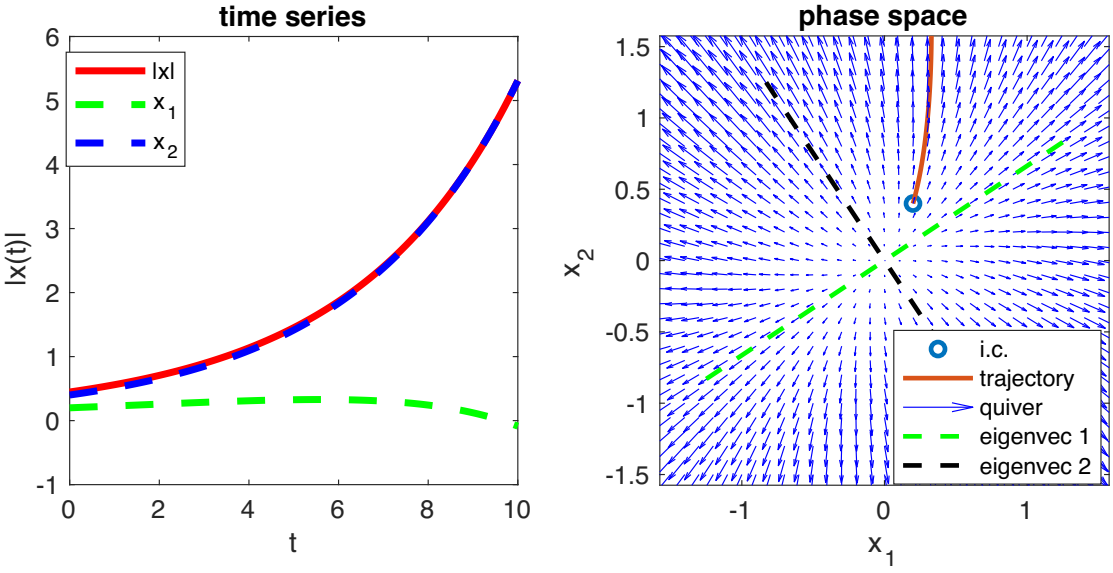
Using the linear ODE Matlab/python demo, the phase portrait and solution of this system is shown below. The vectors on the phase portrait are all pointing toward the origin, showing it is a *stable equilibrium* and that the solution is decaying.



Real eigenvalues, growing solution: Next, consider

$$A = \begin{bmatrix} 0.2306 & -0.0461 \\ -0.0461 & 0.2694 \end{bmatrix}; \quad \lambda_i = 2, 3; \quad S = \begin{bmatrix} -0.8330 & -0.5532 \\ -0.5532 & 0.8330 \end{bmatrix}.$$

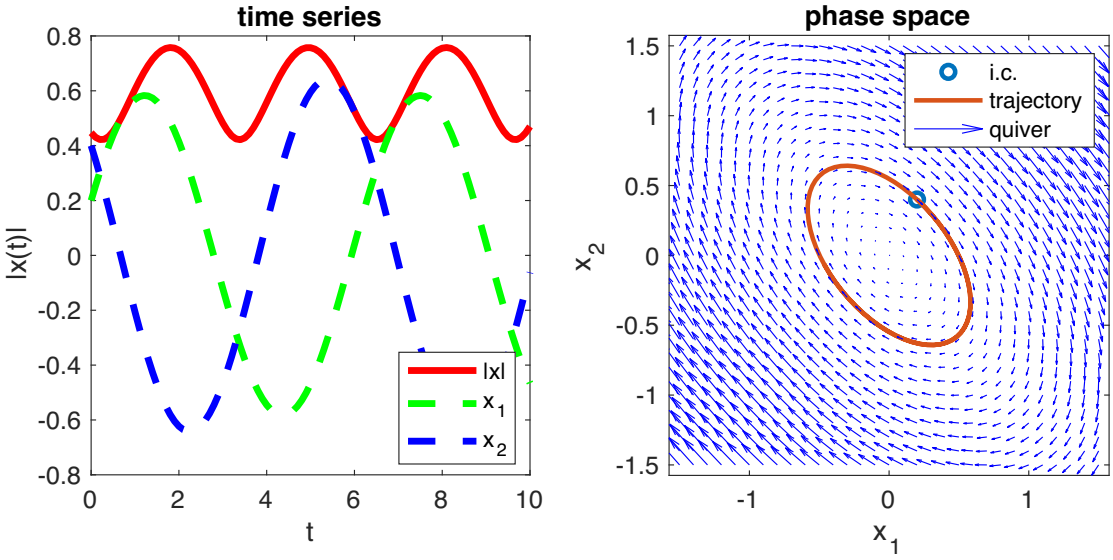
The phase portrait and time series solution of this system are shown below. Unlike the previous example, the vectors in this phase portrait are pointing outwards, showing the origin is *unstable*, and the solution is growing in time.



Imaginary eigenvalues: Next, let

$$A = \begin{bmatrix} 0.610 & 1.064 \\ -1.290 & -0.610 \end{bmatrix}; \quad \lambda_i = 0 \pm i; \quad S = \begin{bmatrix} -0.350 - 0.574i & -0.350 + 0.574i \\ 0.740 & 0.740 \end{bmatrix}.$$

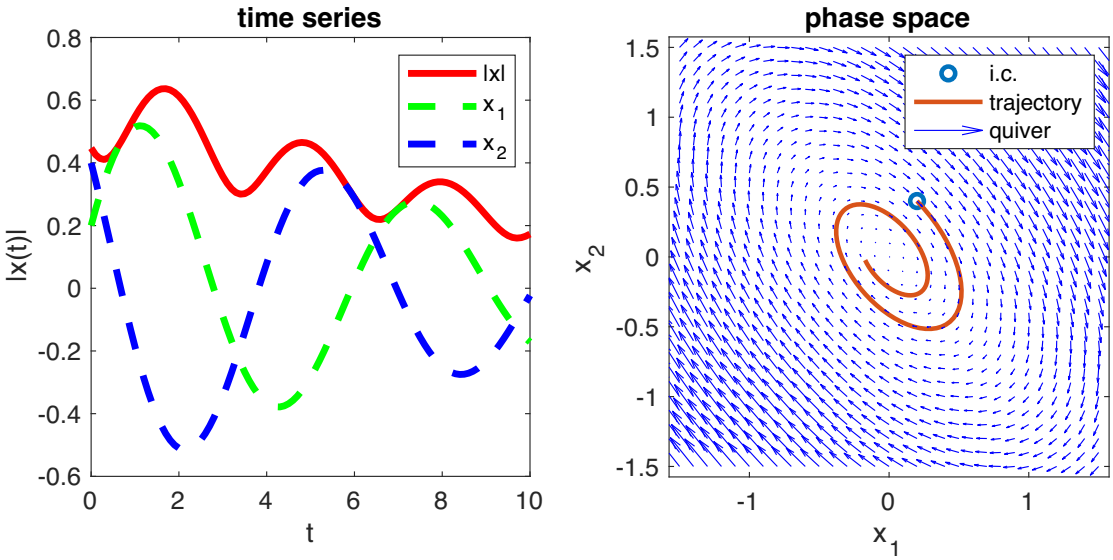
Because the real part of the eigenvalue is zero and the imaginary part non-zero, the phase portrait is an ellipse centered around the origin, and the time series shows oscillations that are not growing nor decaying. The origin is a *neutrally stable* equilibrium in this case.



Stable spiral: Now let

$$A = \begin{bmatrix} 0.510 & 1.064 \\ -1.290 & -0.710 \end{bmatrix}; \quad \lambda_i = -0.1 \pm i; \quad S = \begin{bmatrix} -0.350 - 0.574i & -0.350 + 0.574i \\ 0.740 & 0.740 \end{bmatrix}.$$

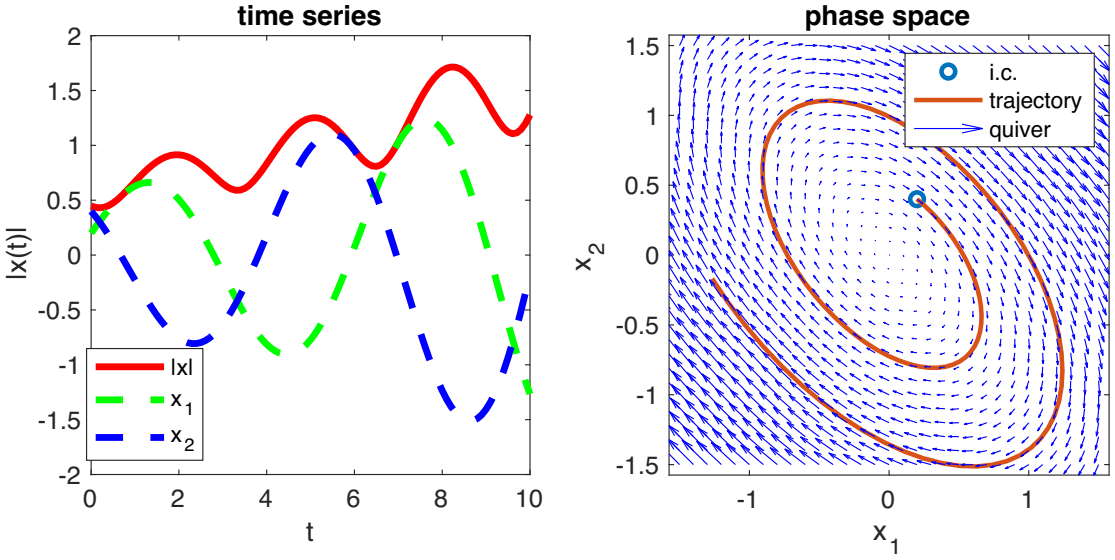
For such complex eigenvalues, with non-zero real part, the phase portrait is a spiral. Since the real part is negative, the origin is a stable point and the solution is a *stable spiral*.



Unstable spiral:

$$A = \begin{bmatrix} 0.710 & 1.064 \\ -1.290 & -0.510 \end{bmatrix}; \quad \lambda_i = 0.1 \pm i; \quad S = \begin{bmatrix} -0.350 - 0.574i & -0.350 + 0.574i \\ 0.740 & 0.740 \end{bmatrix}.$$

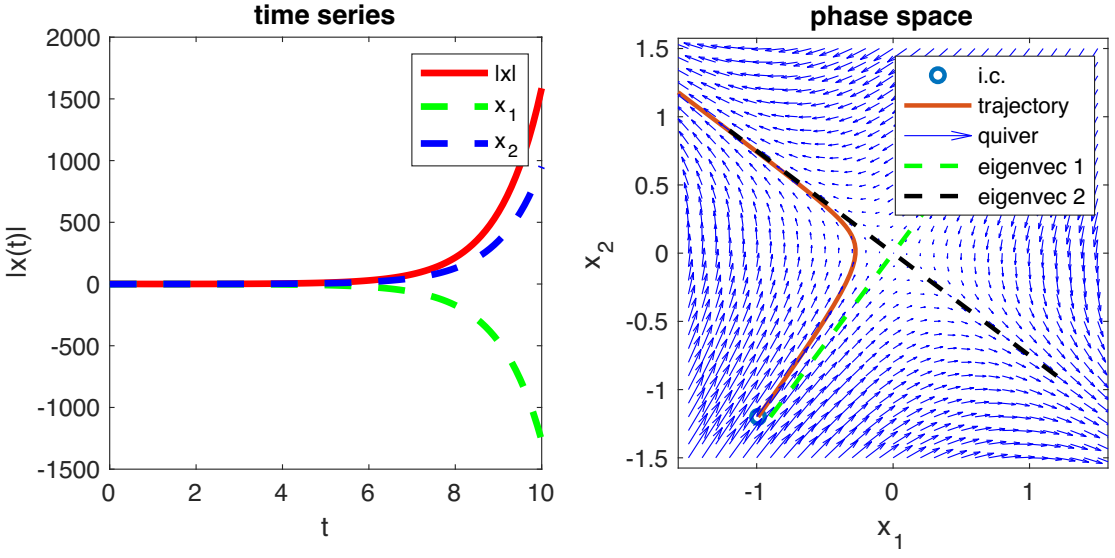
The eigenvectors are the same as in the previous example. However, because the real part of the eigenvalue is positive, the solution is an *unstable spiral*, spiraling away from the origin.



One stable, one unstable mode:

$$A = \begin{bmatrix} -0.08 & -1.44 \\ -1.44 & -0.92 \end{bmatrix}; \quad \lambda_i = -2, 1; \quad S = \begin{bmatrix} 0.6 & -0.8 \\ 0.8 & 0.6 \end{bmatrix}.$$

Because one eigenvalue is positive and one negative, the solution is nearly always going to infinity (unless starting exactly over the dash green line, in the direction of the eigenvector with a negative eigenvalue), but may go toward the origin first.



There are some additional possible behaviors even in two-dimensional systems. See Strogatz (1994).

3.7 Non-normal dynamics and transient growth

We now know that if the real part of all eigenvalues of a set of linear constant coefficient ODEs is negative, the solution must be decaying to zero for long times (as $t \to \infty$). It turns out, though, that before decaying the solution in some cases may first experience a very dramatic amplification. This phenomenon has been used to explain the explosive development of some weather systems, and can be used to predict similar explosive growth in economic models. To understand why and how this might happen, consider

$$\frac{d\mathbf{x}}{dt} = \mathsf{A}\mathbf{x}, \quad \mathbf{x}(t=0) = \mathbf{x}_0.$$

The solution, as shown by (3.11), is $\mathbf{x} = e^{\mathsf{A}t}\mathbf{x}_0 \equiv \mathsf{B}\mathbf{x}_0$. Given the eigenvectors/values of the matrix,

$$\mathsf{A}\mathbf{e}_i = \lambda_i \mathbf{e}_i,$$

where $\mathbf{e}_i$ are the eigenvectors of A. The solution may also be written as

$$\mathbf{x}(t) = \sum_i c_i \mathbf{e}_i e^{\lambda_i t}.$$

where the coefficients c_i are chosen to satisfy the initial conditions such that

$$\mathbf{x}_0 = \sum_i c_i \mathbf{e}_i.$$

The time dependent solution may also be written as,

$$\mathbf{x}(t) = e^{\mathsf{A}t}\mathbf{x}_0 \equiv \mathsf{B}\mathbf{x}_0.$$

If the eigenvalues all have negative real parts, we expect the solution to decay to zero, $\mathbf{x}(t \to \infty) \to 0$. However, it turns out that when A is non-normal, $\mathsf{A}\mathsf{A}^T \neq \mathsf{A}^T\mathsf{A}$ – that is, its eigenvectors are not orthogonal to each other (see appendix A.1) – the solution may undergo an arbitrarily large amplification before decaying to zero.

Consider first a geometric view of the amplification using a 2×2 example where $\mathbf{x} = (x, y)$ shown in Fig. 3.5. The solution may be written as $\mathbf{x}(t) = c_1\mathbf{e}_1 e^{\lambda_1 t} + c_2\mathbf{e}_2 e^{\lambda_2 t}$ and we assume for this example that $\lambda_2 \ll \lambda_1 < 0$. The initial conditions at $t = 0$ are therefore a superposition of two vectors parallel to the two eigenvectors, $\mathbf{x}_0 = c_1\mathbf{e}_1 + c_2\mathbf{e}_2$. As the figure shows, the initial conditions in this particular example are a combination of two nearly parallel large vectors that nearly cancel each other, leading to the smaller, unit norm, initial conditions. This occurs because c_1 is large and positive and c_2 is large and negative. By time $t = \tau$ the first vector decays just slightly, while the second decays significantly. As a result, the solution $\mathbf{x}(t)$, which is the sum of the two vector with the corresponding exponential time dependence, grows, leading to a "non-normal" amplification. At even later times, the first vector decays too, and so does the total solution.

Next, let's see how we find the initial conditions $\mathbf{x}_0$ that lead to a maximal squared norm at a time τ, $|\mathbf{x}|^2 = \mathbf{x}(\tau)^T\mathbf{x}(\tau)$, subject to the requirement that the initial conditions are normalized $|\mathbf{x}_0|^2 = \mathbf{x}_0^T\mathbf{x}_0 = 1$.

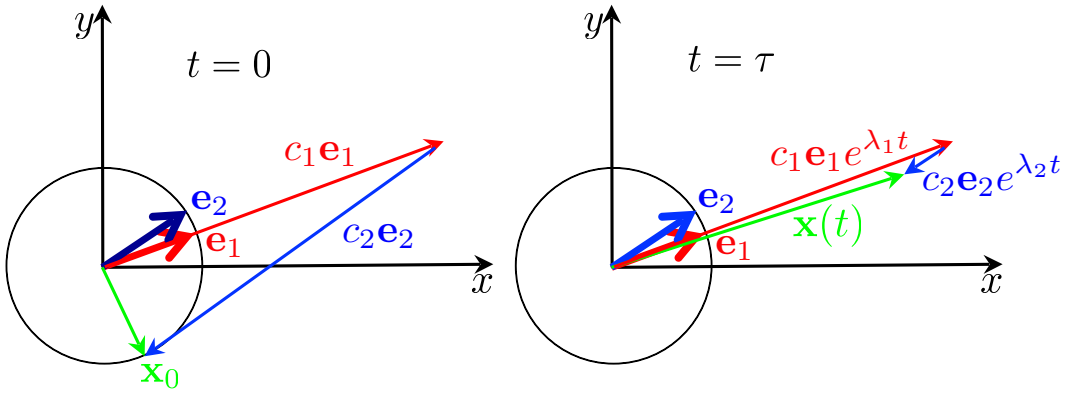


Figure 3.5: Initial conditions and later development in a 2×2 non-normal transient growth case with $\lambda_2 \ll \lambda_1 < 0$.

Use Lagrange multipliers to maximize

$$\begin{aligned} J(\mathbf{x}_0) &= \mathbf{x}(\tau)^T \mathbf{x}(\tau) + \lambda(1 - \mathbf{x}_0^T \mathbf{x}_0) \\ &= (\mathsf{B}\mathbf{x}_0)^T (\mathsf{B}\mathbf{x}_0) + \lambda(1 - \mathbf{x}_0^T \mathbf{x}_0) \\ &= \mathbf{x}_0^T (\mathsf{B}^T \mathsf{B}) \mathbf{x}_0 + \lambda(1 - \mathbf{x}_0^T \mathbf{x}_0). \end{aligned}$$

Denoting the components of the initial conditions as $x_{0,i}$, we require that at the maximum point $\partial J(\mathbf{x}_0)/\partial x_{0,i} = 0$. This, after some algebra, leads to

$$(\mathsf{B}^T \mathsf{B})\mathbf{x}_0 = \lambda \mathbf{x}_0. \tag{3.12}$$

The optimal initial conditions maximizing the state at a later time τ are therefore an eigenvector of $\mathsf{B}^T\mathsf{B}$, where B is the propagator $\mathsf{B} = \exp(\mathsf{A}\tau) = \mathsf{S}e^{\Lambda\tau}\mathsf{S}^{-1}$. We next show that the corresponding eigenvalue is the amplification factor from the initial conditions to the amplified state. Define the amplification factor to be the norm of the solution at τ divided by that at $t = 0$. It is convenient to calculate the square of the amplification factor, $\|\mathbf{x}(\tau)\|^2/\|\mathbf{x}_0\|^2$, and calculate it,

$$\begin{aligned} &\frac{\|\mathbf{x}(\tau)\|^2}{\|\mathbf{x}_0\|^2} = \frac{\mathbf{x}(\tau)^T \mathbf{x}(\tau)}{\mathbf{x}_0^T \mathbf{x}_0} = \frac{(\mathsf{B}\mathbf{x}_0)^T (\mathsf{B}\mathbf{x}_0)}{\mathbf{x}_0^T \mathbf{x}_0} \\ =&\frac{\mathbf{x}_0^T (\mathsf{B}^T \mathsf{B}) \mathbf{x}_0}{\mathbf{x}_0^T \mathbf{x}_0} = \frac{\mathbf{x}_0^T \lambda \mathbf{x}_0}{\mathbf{x}_0^T \mathbf{x}_0} \\ =&\lambda. \end{aligned}$$

This suggests that the optimal initial conditions are the eigenvector of $\mathsf{B}^T\mathsf{B}$ corresponding to the largest eigenvalue. The amplification factor is therefore is the square root of the largest eigenvalue.

An important point to note is that when we find the optimal initial conditions that maximize the solution norm at $t = \tau$, and then plot the norm of the solution as function of time, the maximum of this time series does not necessarily occur at $t = \tau$. That is, the above maximization only requires that $\|\mathbf{x}(\tau)\|^2$ is maximized subject to unit-norm initial conditions, it does not require the norm at time τ to be larger than at other times. This is demonstrated in Figure 3.6, where the solution was optimized for $\tau = 1$ (denoted by a dash vertical black line) yet the peak in the norm of the solution occurs around $t = 0.5$.

A numerical example based on,

$$\mathsf{A} = \begin{bmatrix} -9.7945 & 60.2566 \\ -0.7395 & 4.2945 \end{bmatrix}$$

which has eigenvalues -5 and -0.5 is shown in Fig. 3.6 where we are trying to maximize the solution noram at $t = \tau = 1$. First, we find the propagator, $\mathsf{B} = e^{A*1} = \mathsf{S}e^{\Lambda\tau}\mathsf{S}^{-1}$, and $\mathsf{B}^T\mathsf{B}$,

$$\begin{aligned} \mathsf{B} &= \mathsf{S}e^{\Lambda\tau}\mathsf{S}^{-1} \\ &= \begin{pmatrix} -0.9969 & -0.9883 \\ -0.0793 & -0.1525 \end{pmatrix} \begin{pmatrix} 0.0067 & 0 \\ 0 & 0.6069 \end{pmatrix} \begin{pmatrix} -2.0715 & 13.4291 \\ 1.0776 & -13.5451 \end{pmatrix} \\ \mathsf{B} &= \begin{pmatrix} -0.6325 & 8.0342 \\ -0.0986 & 1.2461 \end{pmatrix} \\ \mathsf{B}^T\mathsf{B} &= \begin{pmatrix} 0.4097 & -5.2042 \\ -5.2042 & 66.1019 \end{pmatrix} \end{aligned}$$

Now, we want to solve $(\mathsf{B}^T\mathsf{B})\mathbf{x}_0 = \lambda\mathbf{x}_0$ to find $\mathbf{x}_0$ - the eigenvector corresponding to the largest eigenvalue of $\mathsf{B}^T\mathsf{B}$. We find,

$$\mathbf{x}_0 = \begin{pmatrix} -0.0785 \\ 0.9969 \end{pmatrix} \quad \text{and the amplification factor, } \lambda = 66.5116.$$

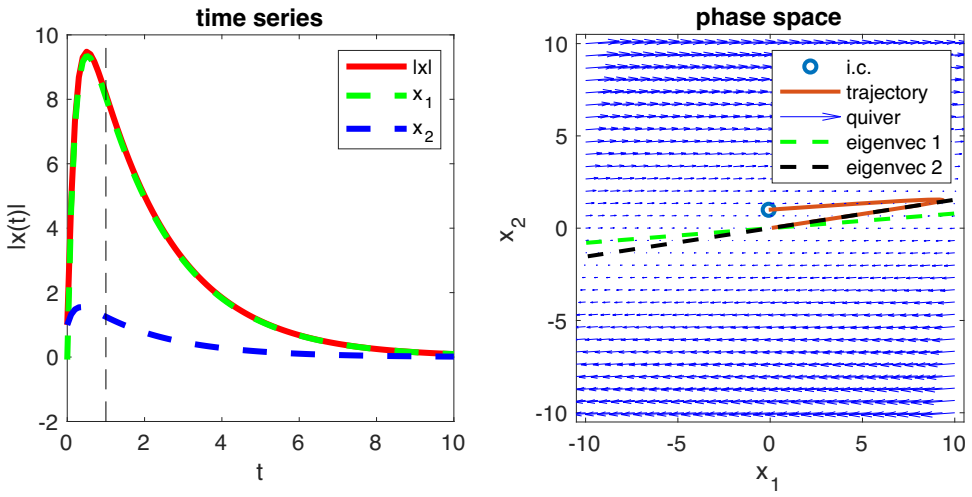


Figure 3.6: A numerical example of transient growth.

.

3.8 Jordan form

Much of the above discussion, including the solution of ODEs using matrix exponentiation, assumed that the matrix can be diagonalized. It turns out, though, that this is not always possible. To see this,

consider a matrix A,

$$\mathsf{A} = \begin{bmatrix} 2.5 & -0.5 & -1 \\ 0.5 & 1.5 & 1 \\ 0 & 0 & 2 \end{bmatrix} \tag{3.13}$$

which has a proper inverse and is therefore well-conditioned,

$$\mathsf{A}^{-1} = \begin{bmatrix} 0.375 & 0.125 & 0.125 \\ -0.125 & 0.625 & -0.375 \\ 0 & 0 & 0.5 \end{bmatrix}.$$

However, the eigenvalues and eigenvectors of A are all identical, and are calculated by Matlab to be,

$$\mathsf{S} = \begin{bmatrix} 0.7071 & 0.7071 & -0.7071 \\ 0.7071 & 0.7071 & -0.7071 \end{bmatrix}$$
$$\Lambda = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}.$$

The matrix of eigenvectors is therefore singular and cannot be inverted. As a result, we can't diagonalize the matrix using $\mathsf{SAS}^{-1} = \Lambda$, where Λ is the diagonal matrix of eigenvalues. The best we can do is to find an alternative similarity transformation that brings A to a "Jordan form", J. That is, we need find a matrix of M, such that $\mathsf{M}^{-1}\mathsf{AM} = \mathsf{J}$, where in the simple case considered here, the Jordan form has the eigenvalues on the diagonal and ones just above the diagonal

$$\mathsf{J} = \begin{bmatrix} \lambda & 1 & 0 \\ 0 & \lambda & 1 \\ 0 & 0 & \lambda \end{bmatrix}.$$

This form is called a "Jordan block", and in general a non-diagonalizable matrix has a Jordan form that is made of several Jordan blocks of varying sizes. We will consider first the simple case of a single Jordan form of a 3×3 matrix.

3.8.1 Calculating the transformation to Jordan form

Start with $\mathsf{M}^{-1}\mathsf{AM} = \mathsf{J}$, multiply by M on the left of both sides, to find $\mathsf{AM} = \mathsf{MJ}$. Writing $\mathsf{M} = [\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}]$, and given the special structure of J, we have,

$$\begin{aligned} \mathsf{A}\mathbf{v}_1 &= \lambda \mathbf{v}_1, \\ \mathsf{A}\mathbf{v}_2 &= \lambda \mathbf{v}_2 + \mathbf{v}_1, \\ \mathsf{A}\mathbf{v}_3 &= \lambda \mathbf{v}_3 + \mathbf{v}_2. \end{aligned}$$

This may also be written as,

$$(\mathsf{A} - \lambda \mathsf{I})\mathbf{v}_1 = 0 \tag{3.14}$$
$$(\mathsf{A} - \lambda \mathsf{I})\mathbf{v}_2 = \mathbf{v}_1, \tag{3.15}$$
$$(\mathsf{A} - \lambda \mathsf{I})\mathbf{v}_3 = \mathbf{v}_2. \tag{3.16}$$

Now multiply (3.15) by $(A-\lambda I)$ and use (3.14) to find $(A-\lambda I)^2\mathbf{v}_2 = 0$. Similarly, multiply (3.16) by $(A-\lambda I)^2$ and use $(A-\lambda I)^2\mathbf{v}_2 = 0$, to find,

$$\begin{aligned}(A-\lambda I)\mathbf{v_1} &= 0\\ (A-\lambda I)^2\mathbf{v}_2 &= 0\\ (A-\lambda I)^3\mathbf{v}_3 &= 0.\end{aligned} \tag{3.17}$$

This is why the $\mathbf{v}_i$ are called *generalized eigenvectors*, as opposed to regular eigenvectors which simply all satisfy $(A-\lambda I)\mathbf{e}_i = 0$.

To calculate the transformation matrix to Jordan form, M, start with $\mathbf{v}_3$. The equations above show that this vector is in the null space of the matrix $(A-\lambda I)^3$. However, note that $(A-\lambda I)^2\mathbf{v}_3 = \mathbf{v}_1 \neq 0$. Thus, $\mathbf{v}_3$ is found by looking for a vector that is in the null space of $(A-\lambda I)^3$ but not in the null space of $(A-\lambda I)^2$. Given $\mathbf{v}_3$ we then use (3.16) to calculate $\mathbf{v}_2$ and then use (3.15) to calculate $\mathbf{v}_1$. We can now form the transformation matrix to Jordan form as $M = [\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$.

3.8.2 Numerical example and sensitivity to noise

For the above example, solving for $V_{\text{candidates}}$ as the null space of $(A-2I)^3$, we find

$$V_{\text{candidates}} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}.$$

Next, multiplying the vectors in the null space by $(A-2I)^2$, we find that only the third columns does not yield zero,

$$(A-2I)^2 V_{\text{candidates}} = \begin{bmatrix}0 & 0 & -1\\ 0 & 0 & -1\\ 0 & 0 & 0\end{bmatrix}.$$

Therefore, the third column of $V_{\text{candidates}}$ is the desired vector $\mathbf{v}_3 = (0,0,1)^T$. Next, $\mathbf{v}_2 = (A-2I)\mathbf{v}_3 = (-1,1,0)^T$, and $\mathbf{v}_1 = (A-2I)\mathbf{v}_2 = (-1,-1,0)^T$. The needed transformation is found to be,

$$M = [\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] = \begin{bmatrix}-1 & -1 & 0\\ -1 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}.$$

Verify that M indeed provides the needed similarity transformation to Jordan form,

$$J = M^{-1}AM = \begin{bmatrix}-1 & -1 & 0\\ -1 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}2.5 & -0.5 & -1\\ 0.5 & 1.5 & 1\\ 0 & 0 & 2\end{bmatrix}\begin{bmatrix}-0.5 & -0.5 & 0\\ -0.5 & 0.5 & 0\\ 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}2 & 1 & 0\\ 0 & 2 & 1\\ 0 & 0 & 2\end{bmatrix}.$$

However, it turns out that the Jordan form is exceedingly sensitive to noise. Consider adding an extremely small perturbation to one of the elements of the matrix, creating

$$A_1 = \begin{bmatrix}2.5000001 & -0.5 & -1\\ 0.5 & 1.5 & 1\\ 0 & 0 & 2\end{bmatrix},$$

such that the difference between A_1 and A is merely 10^{-6} in a single element. The eigenvalues/eigenvectors of A_1 are now distinct and independent of each other,

$$S = \begin{bmatrix} 0.7076 & 0.7066 & 0.7071 \\ 0.7066 & 0.7076 & 0.7071 \\ 0 & 0 & 0.0000 \end{bmatrix}, \qquad \lambda_i = (2.0007, 1.9993, 2.0000),$$

such that A_1 does not require a Jordan form anymore to be diagonalized. We conclude from this that Jordan form is extremely sensitive to noise and therefore is not likely to come up in many data-driven applications. The next subsection provides some intuition for the reason for this extreme sensitivity to noise using an ODE perspective.

For some additional details and generalities and for proof by recursion that a Jordan form can always be found, see Strang Appendix B.

3.8.3 A fuller Jordan form example

So far we have dealt with a single Jordan block. The Jordan form of a more general matrix may be composed of multiple Jordan blocks, and while we do not deal with how to find the transformation matrix for this form (Strang Appendix B), here is an example calculated using Matlab/python.

■ **Example 3.1** Consider the matrix

$$A = \begin{bmatrix} 2 & 1 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4\frac{2}{3} & -\frac{1}{3} & 0 & 2\frac{2}{3} & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -16 & 2 & -8 & 0 & -8 & -2 \\ 0 & -\frac{2}{3} & \frac{1}{3} & 0 & 1\frac{1}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 8 & 0 & 4 & 0 & 6 & 1 \\ 0 & 2\frac{2}{3} & -1\frac{1}{3} & 0 & 18\frac{2}{3} & 0 & 0 & 6 \end{bmatrix}$$

whose Jordan form obtained using Matlab/python's `J=jordan(A)` is,

$$J = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}.$$

This Jordan form shows that A has one eigenvalue with the value of 2 with three identical eigenvectors, another eigenvalue 2 with a single, unique eigenvector, two eigenvalues of 4, each with a separate, unique eigenvectors, and two eigenvalues of 6 again sharing a single eigenvector. ■

3.8.4 Jordan form and ODEs

In order to provide a motivation for the Jordan form and to also understand its extreme sensitivity to noise, consider an example of a second order ODE equivalent to a set of first order ODEs based on a

Jordan form,

$$d\mathbf{x}/dt = \mathsf{J}\mathbf{x},$$
$$\mathsf{J} = \begin{pmatrix} -2 & 1 \\ 0 & -2 \end{pmatrix}$$

the two equations are

$$\dot{x} = -2x + y,$$
$$\dot{y} = -2y.$$

Take the derivative of the first $\ddot{x} = -2\dot{x} + \dot{y}$, add twice the first equation to find $\ddot{x} + 2\dot{x} = (-2\dot{x} + \dot{y}) + 2(-2x + y)$, and use the second equation to find,

$$\ddot{x} + 4\dot{x} + 4x = 0. \tag{3.18}$$

Note that this is equivalent to

$$\begin{aligned} 0 &= (d_t - (-2))^2 x \\ &= (d_t + 2)(\dot{x} + 2x) \\ &= (\ddot{x} + 2\dot{x}) + 2(\dot{x} + 2x) \\ &= \ddot{x} + 4\dot{x} + 4x. \end{aligned}$$

The general solution is

$$x = c_1 e^{-2t} + c_2 t e^{-2t}. \tag{3.19}$$

To find this, substitute $x = e^{at}$ in eqn. (3.18) to find $a^2 + 4a + 4 = (a+2)^2 = 0$, so that $a_{1,2} = -2$ and therefore the secular term te^{-2t} must be added as a second solution to be able to satisfy general initial conditions.

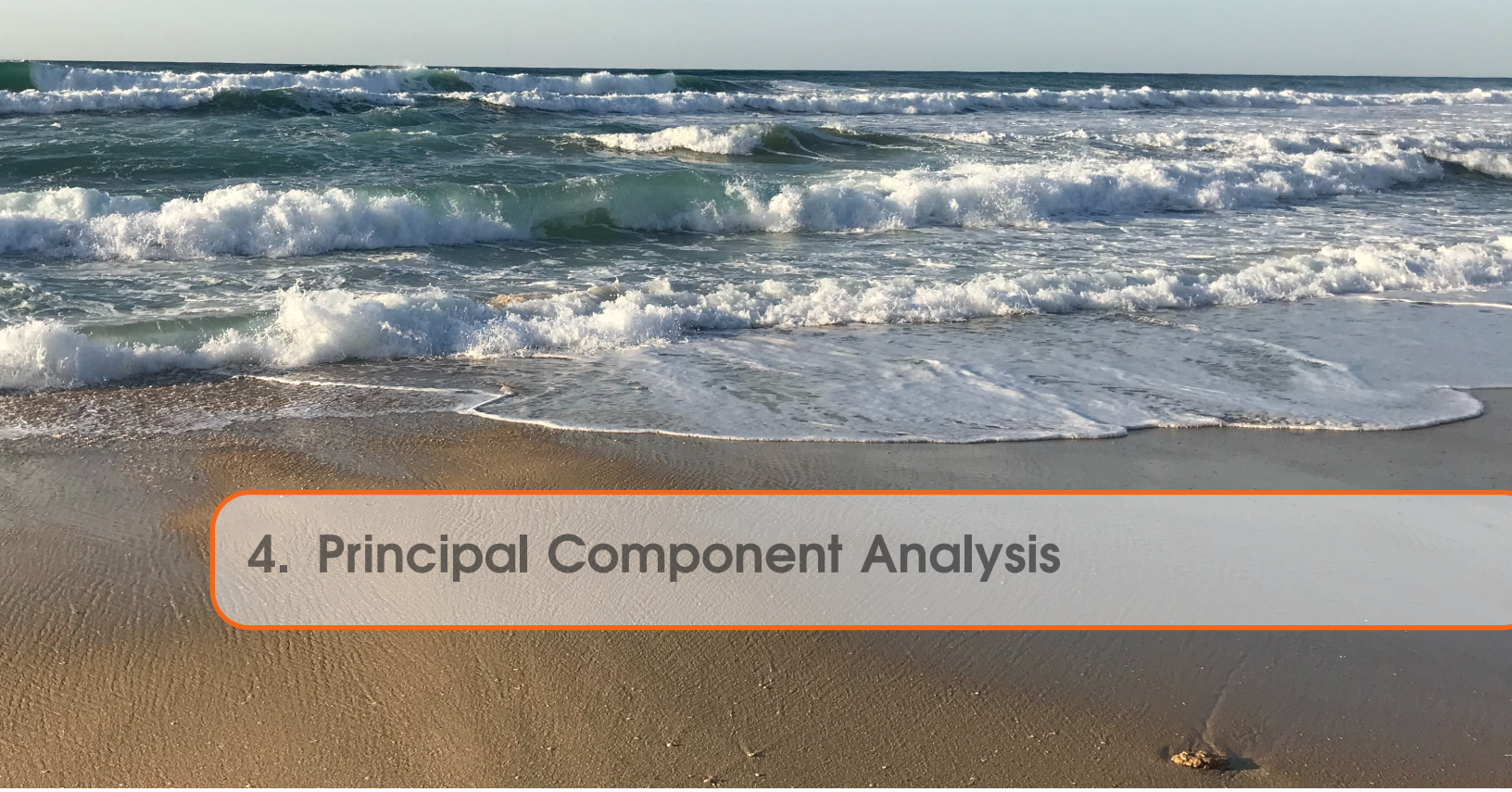
Physical interpretation: Eqn (3.18) is a "critically damped" oscillator equation. If the damping term coefficient were $4 + \varepsilon$, the solution would have exponentially approached zero. If it were $4 - \varepsilon$ (where ε is a small noise parameter) we would have obtained damped oscillations. On the boundary between these two regimes, one obtains the solution (3.19). This also helps explain why Jordan form is so sensitive to noise, and why it is not of practical interest: it's not likely that the friction would have exactly this critical value in a real-world problem.

More on connection to ODEs: Consider the exponential of a Jordan form. Let $\hat{\mathsf{I}}_{k\times k}$ be a matrix with 1 above the diagonal, so that one Jordan block is $\mathsf{J} = \lambda\mathsf{I} + \hat{\mathsf{I}}$. Now $e^{\mathsf{J}t} = (\mathsf{I}e^{\lambda t})e^{\hat{\mathsf{I}}t}$. Noting that $\hat{\mathsf{I}}^k = 0$, one finds that $e^{\hat{I}t}$ is given by a finite power series leading to a form (as in Strang, eqn 10 on p 331),

$$\mathsf{J}^k = \begin{pmatrix} \lambda & 1 & 0 \\ 0 & \lambda & 1 \\ 0 & 0 & \lambda \end{pmatrix}^k$$
$$e^{\mathsf{J}t} = (\mathsf{I}e^{\lambda t})e^{\hat{\mathsf{I}}t} = \begin{pmatrix} e^{\lambda t} & te^{\lambda t} & \frac{1}{2}t^2 e^{\lambda t} \\ 0 & e^{\lambda t} & te^{\lambda t} \\ 0 & 0 & e^{\lambda t} \end{pmatrix}$$

which contains resonant terms (proportional to powers of t multiplying the exponential solutions) as expected. The solution to a set of linear, constant coefficient ODEs can therefore be written in terms of the matrix exponential even in the case that the matrix cannot be diagonalized, using the matrix of generalized eigenvectors instead.

Part Two



4. Principal Component Analysis

4.1 Principal Component Analysis (PCA) from the covariance matrix

4.1.1 Motivation

Principle Component Analysis, also known as "Factor Analysis" or "Empirical Orthogonal Functions", is a powerful yet simple technique for uncovering relationships within data, with applications in finance, weather and climate, social sciences and more. Some interesting applications include,

- **Quantitative finance:** Consider a vector containing stock prices, where the vector is given daily, for several years, and can therefore be represented as a data matrix whose columns are the daily data. We wish to find out which stock prices vary together, and which stocks vary opposite to, or independently of, one another to create an optimal portfolio. PCA allows us to call upon the covariance matrix to more clearly understand these kinds of relationships.
- **Weather and climate systems:** See the animation of sea surface temperature (SST) during El Niño events in slides 1-4. We can use PCA to look for trends and correlations between different weather phenomena and observable climate metrics.
- **Neuroscience:** In neuroscience, spike-triggered covariance analysis (a variant of PCA) is used to analyze the generation of neuron action potential.

4.1.2 Derivation

Consider N vectors of size $M \times 1$, $\mathbf{f}_n = f_{mn}$. Each column vector could represent, for example, data at a given time, and the different components of $\mathbf{f}_n$ could correspond to, for example, either data – such as temperature – at different locations, or, say, prices of stocks of different companies at this time. Let $\mathsf{F} = (\mathbf{f}_1, \ldots, \mathbf{f}_N) = f_{mn}$ be the $M \times N$ data matrix containing the entire data set. While we are using a terminology that is based on the subscript representing time, this is not necessarily the case. For example, the elements of $\mathbf{f}_n$ could be the number of high school students taking courses in social sciences, biology, literature, math and physics (that is, $M = 5$), and the subscript n could represent

different schools.

The first step is to remove the mean from each row of the data matrix, defining a mean-less primed data matrix,

$$f'_{mn} = f_{mn} - \frac{1}{N}\sum_{i=1}^{N} f_{mi}$$

and from now on we are going to drop the prime and assume the mean is zero.

Our objective is to find M orthogonal vectors $\mathbf{u}_j$ of dimension $M \times 1$ (these are the principal components) that best describe the variability in the data. By this we mean that we look for $\mathbf{u}_1$ of magnitude one such that $\sum_{n=1}^{N}(\mathbf{u}_1 \cdot \mathbf{f}_n)^2$ is maximal. Maximizing this projection means that $\mathbf{u}_1$ is similar, up to a minus sign, to the typical patterns of the data vectors $\mathbf{f}_n$. Next, we look for $\mathbf{u}_2$ of magnitude one such that $\sum_{n=1}^{N}(\mathbf{u}_2 \cdot \mathbf{f}_n)^2$ is maximal again, and is also orthogonal to the first $\mathbf{u}_1$, i.e. $\mathbf{u}_1 \cdot \mathbf{u}_2 = 0$. Then, in general, we wish $\mathbf{u}_j$ to be of magnitude one and to maximize $\sum_{n=1}^{N}(\mathbf{u}_j \cdot \mathbf{f}_n)^2$ while being orthogonal to all previous vectors. Define a matrix U whose columns are the vectors $\mathbf{u}_j$, and use the data matrix notation to rewrite the sum to be maximized as,

$$\begin{aligned}
\frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_j \cdot \mathbf{f}_n)^2 = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_j^T \mathbf{f}_n)^2 &= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{m=1}^{M} U_{mj}F_{mn}\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{m=1}^{M} U_{mj}F_{mn}\right)\left(\sum_{k=1}^{M} U_{kj}F_{kn}\right) \\
&= \sum_{m=1}^{M}\sum_{k=1}^{M} U_{mj}\left(\frac{1}{N}\sum_{n=1}^{N} F_{mn}F_{kn}\right)U_{kj} \\
&= \mathbf{u}_j^T\left(\frac{1}{N}FF^T\right)\mathbf{u}_j.
\end{aligned}$$

The matrix that appears here,

$$\mathsf{C}_{M\times M} \equiv \frac{1}{N}\mathsf{F}_{M\times N}\mathsf{F}^T_{N\times M}$$

is the covariance matrix, whose element $c_{ij} = \frac{1}{N}\sum_{n=1}^{N} f_{in}f_{jn}$ is the time averaged product (covariance) of data elements i and j.

We next show that the covariance matrix is positive semi-definite,

$$\begin{aligned}
\mathbf{x}^T\mathsf{C}\,\mathbf{x} = \sum_{ij} x_i C_{ij} x_j &= \frac{1}{N}\sum_{ij} x_i\left(\sum_n F_{in}F_{jn}\right)x_j \\
&= \frac{1}{N}\sum_{ij} x_i\left(\sum_n F_{in}F_{jn}\right)x_j = \frac{1}{N}\sum_n\left(\sum_i x_i F_{in}\right)\left(\sum_j F_{jn}x_j\right) \\
&= \frac{1}{N}\sum_n\left(\sum_i x_i F_{in}\right)^2 \geq 0.
\end{aligned}$$

and this implies the eigenvalues are non-negative.

The constrained optimization problem of maximizing $\sum_{n=1}^{N}(\mathbf{u}_j \cdot \mathbf{f}_n)^2$ and requiring $\mathbf{u}_j$ to be of unit magnitude is solved using Lagrange multipliers by maximizing,

$$\mathbf{u}_j^T \mathsf{C}\, \mathbf{u}_j + \lambda(1 - \mathbf{u}_j^T \mathbf{u}_j).$$

By requiring the derivative with respect to the elements of $\mathbf{u}_j$ to vanish, we find that the optimal vectors are eigenvectors of the covariance matrix,

$$\mathsf{C}\mathbf{u}_j = \lambda \mathbf{u}_j.$$

Because the covariance matrix is symmetric, the eigenvectors $\mathbf{u}_j$ are orthogonal, as we required above. The projection of the eigenvector $\mathbf{u}_j$ on the data – the quantity being maximized – is equal to the corresponding eigenvalue,

$$\mathbf{u}_j^T \mathsf{C}\, \mathbf{u}_j = \mathbf{u}_j^T \lambda_j \mathbf{u}_j = \lambda_j,$$

and therefore the maximal projection occurs for the eigenvectors corresponding to the largest eigenvalues. This also provides some intuition for the discussion of the variance explained by each PC in section 4.1.4 below. We can project the data at a given time, $\mathbf{f}_n$, on a principal component $\mathbf{u}_j$, to obtain the amplitude for this principal component at that time,

$$t_{jn} = \mathbf{f}_n \cdot \mathbf{u}_j.$$

This corresponds to a time series $\{t_{j1}, \ldots, t_{jN}\}$ which represents the amplitude of the principal component $\mathbf{u}_j$ as function of time. Time series are more generally referred to as the principal component "expansion coefficients", in particular in applications where the different data vectors do not represent different times as mentioned above. There are M such time series, and we can put them as the rows of an $M \times N$ matrix as $\mathsf{T} = (t_{jn})$. Remembering that we defined the M principal components to be the columns of the $M \times M$ matrix $\mathsf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_M)$, we can write the equation used to derive the time series matrix as

$$\mathsf{T} = \mathsf{U}^T \mathsf{F}.$$

Multiplying by the orthogonal matrix of principal components U, which is also the inverse of U^T, we find,

$$\mathsf{F}_{M\times N} = \mathsf{U}_{M\times M}\, \mathsf{T}_{M\times N}.$$

This equation corresponds to an expansion of the data in terms of the eigenmodes (principal components). Writing it for one time,

$$\mathbf{f}_n = \sum_{j=1}^{M} \mathbf{u}_j t_{jn}.$$

If some of the PCs are judged not important, for example, if we decide that they represent only noise in the data, we may therefore wish to reconstruct the data using only $k < M$ PCs. Then,

$$\begin{aligned} \mathsf{F}_{\text{reconstructed}} &= \sum_{j=1}^{k} \mathbf{u}_j t_{jn} \\ &= \mathsf{U}_{M\times k} \mathsf{T}_{k\times N} \\ &= \mathtt{U(:,1:k)} * \mathtt{T(1:k,:)} \end{aligned}$$

4.1.3 Example of PCA

■ **Example 4.1** Consider the following data set that represents the deviations from the long-term mean of the CO_2 concentration in the atmosphere (first line), the global mean surface temperature (second) and the area of summer sea ice in the Arctic (third) over 9 consecutive years,

$$\mathsf{F} = \begin{pmatrix} 0 & -1.75 & 0.25 & -1.0 & 0.5 & -0.25 & 0.75 & 0.5 & 1.0 \\ -3.0 & -0.5 & -1.75 & 0.25 & -0.5 & 1.0 & 0.75 & 1.75 & 2.0 \\ 0 & 1.75 & -0.25 & 1.0 & -0.5 & 0.25 & -0.75 & -0.5 & -1.0 \end{pmatrix},$$

where each column represents a year.

The PCA steps are as follows,

1. Calculate the covariance matrix by calculating $\mathsf{F}\mathsf{F}^T/N$, where $N = 9$ is the number of years.
2. Calculate the eigenvectors of the covariance matrix and normalize them; these are the principle components. Place the eigenvectors as columns of a matrix U, where each column is a principal component of the data matrix.
3. Calculate the time series, $\mathsf{T} = \mathsf{U}^T\mathsf{F}$, where F is the original dataset and U is the matrix of eigenvectors. If the data matrix F has dimensions $M \times N$, the time series T should be the result of multiplying matrices that are $M \times M$ and $M \times N$ and so should also be of dimensions $M \times N$.

The covariance matrix is,

$$\mathsf{F}\mathsf{F}^T/N = \mathsf{C} = \begin{pmatrix} 0.6944 & 0.3472 & -0.6944 \\ 0.3472 & 2.361 & -0.3472 \\ -0.6944 & -0.3472 & 0.6944 \end{pmatrix}.$$

We see that the diagonal terms are the variance of the three variables, that is, the average of $(CO_2)^2$, of the temperature squared and of the ice area squared (because each variable (row) in the data matrix has zero mean). The off-diagonal terms are the covariance of the different variables. The values indicate that the CO_2 and global temperature vary in the same direction, hence positively correlated (entry c_{12}), while they are both negatively correlated with sea ice (c_{13}, c_{23}). That is, CO_2 and temperature tend to increase/decrease together, while sea ice vary in the opposite direction.

The eigenvalues and eigenvectors (PCs) of the covariance matrix are calculated, and sorted by the value of the eigenvalues from large to small,

$$\mathsf{D} = \begin{pmatrix} 2.57 & 0 & 0 \\ 0 & 1.18 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \mathsf{U} = \begin{pmatrix} 0.272 & 0.653 & 0.707 \\ 0.923 & -0.385 & 0 \\ -0.272 & -0.653 & 0.707 \end{pmatrix}.$$

The PCs $\mathbf{u}_i$ are orthogonal vectors whose structure best represents the nine data vectors. In other words, $\mathbf{u}_1$ is a vector that is as parallel as possible to the data vectors. $\mathbf{u}_2$ is as parallel as possible to the data vectors while also being perpendicular to $\mathbf{u}_1$. Because the data space is three-dimensional, the last eigenvector $\mathbf{u}_3$ is actually completely determined by the requirement that it is perpendicular to the first two PCs. The principal components corresponding to the largest eigenvalues represent the main modes in the data.

The eigenvector corresponding to the largest eigenvalue is the first PC $\mathbf{u}_1 = (0.272, 0.923, -0.272)^T$. From a climate point of view, this PC represents a variability mode in which when the CO_2 and global temperature both increase, the sea ice decreases, and vice versa.

Furthermore, we may be interested in calculating the time series, T, for this data, which we can find by multiplying the transpose of the matrix of principle components, U, by the original data set, F. We calculate the time series to be,

$$\mathsf{T} = \begin{pmatrix} -2.77 & -1.41 & -1.48 & -0.314 & -0.189 & 0.787 & 1.1 & 1.89 & 2.39 \\ 1.16 & -2.09 & 1.0 & -1.4 & 0.845 & -0.711 & 0.69 & -0.0212 & 0.535 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The time series of the first and most dominant PC, $\mathbf{u}_1$, shows a continuous increase over time. Given that, from our interpretation of the principle components, CO_2 concentration and temperature vary in the opposite direction of sea ice coverage, we can conclude from the time series that temperature and CO_2 concentration increase with time, while sea ice coverage decreases. The time series of the second mode, shows alternating signs, indicating short-term variability superimposed on the slower climate change represented by the largest mode. The smallest PC, is positive for CO_2 and sea ice coverage and negative for temperature. It explains zero percent of the variance (which will be discussed further in the next section), presumably because even during climate variability in this data set, the CO_2 and temperature never vary together while leaving the sea ice unchanged. Thus, this principal component does not represent a physical mode.

Two eigenvalues dominate since one is zero (yielding an empty bottom row in the time series), and as a result the data can effectively be reconstructed using only two PCs. Let the reconstructed data be $\mathsf{F}_{reconstructed}$, we can calculate $\mathsf{F}_{reconstructed} =$ `U(:,1:2)*T(1:2,:)`. Indeed, since one of the eigenvalues is zero (rather than, say, much smaller than the other two eigenvalues), the reconstructed data set is identical to the original. ∎

4.1.4 Fraction of variance explained by PC modes

We found so far that the eigenvectors of the covariance matrix are the principal components that best describe the variability in the data, and that the projection of these modes on the data is given by the time series. We consider further the role of the eigenvalues of the covariance matrix. First, let's introduce the concept of "total variance" using a simple example. If the data vectors are given by $\mathbf{f}_n = (x_n, y_n, z_n)^T$, using the expression for the variance of x_n, $var(x) = \sum_{n=1}^{N} x_n^2/N$, and similarly for y_n and z_n, we define the total variance of these data as $var(x_n) + var(y_n) + var(z_n)$. In terms of the data matrix, the variance of the ith variable in the data vectors (the ith row of F) is,

$$\frac{1}{N}\sum_{n=1}^{N} (f_{in})^2,$$

and the total variance is the sum over all of these,

$$\text{total variance} = \sum_{i=1}^{M}\left(\frac{1}{N}\sum_{n=1}^{N} (f_{in})^2\right). \tag{4.1}$$

Next, use the fact that this total variance is exactly the trace (sum of diagonal elements) of the covariance matrix, which is also equal to the sum of eigenvalues, $\text{trace}(\mathsf{C}) = \sum_{j=1}^{M} \lambda_j$. As a result, the fraction of the variance explained by the ith PC, $\mathbf{u}_i$ is,

$$\frac{\lambda_i}{\sum_{j=1}^{M} \lambda_j}. \tag{4.2}$$

We can consider another way to understand the issue of "fraction of total variance explained". Let $\mathbf{u}_1$ be the first PC, corresponding to the largest eigenvalue of C. Given also the corresponding time series, we can create a partial reconstruction of the data using this first PC only,

$$\mathsf{F}^{PC1} = \mathbf{u}_1 \mathbf{t}_1 = \mathtt{U}(:,1) * \mathtt{T}(1,:) \quad (= M \times N).$$

Now calculate the total variance of this reconstructed data using (4.1). The ratio of this total variance to that calculated using the full rather than reconstructed data, is the fraction of the total variance explained by the first PC, and should be equal to the first eigenvalue divided by the sum of all eigenvalues, as derived above.

4.1.5 PCA from covariance matrix in Matlab

```
%% calculate covariance matrix:
C=F*F'/N;
%% calculate PCs (matrix U) and eigenvalues:
[U,D]=eig(C);
%% calculate time expansion coefficients:
T=U'*F;
%% fraction of variance explained by each PC:
trace_C=trace(C);
fraction_variance=diag(D)/trace_C;
%% reconstruct data using only k PCs (assuming they are sorted!):
F_reconstructed=U(:,1:k)*T(1:k,:);
```

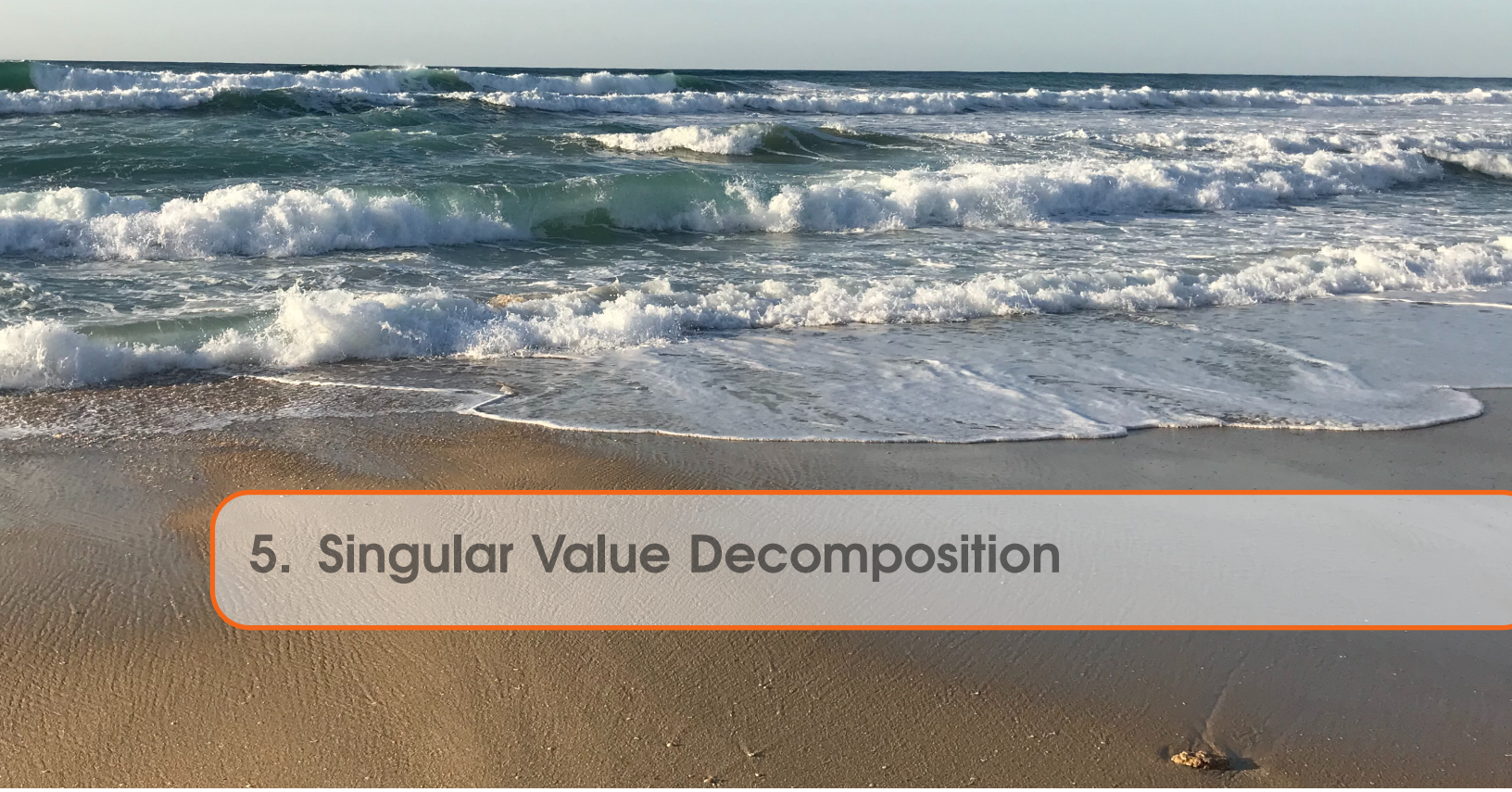
4.1.6 Examples and additional issues

Examples, activities and demos,

1. Complete the demo of PCA analysis of time series of four stocks PCA_small_data_example_using_covariance.m/py
2. Consider the idealized 1d example, demonstrating the meaning of the covariance matrix, and the calculation of PCs for both $\cos(kx)\cos(\omega t)$ and for random data: one_dim_covariance_matrix_and_PCA_tutorial.m/py
3. Idealized 2d example: which shows the calculated PC structure and time series: example_2_PCA.m/py

(Optional) Some additional issues to consider are that,

1. PCs do not represent physical modes, only data representation modes.
2. There may be distortion of calculated modes due to overlapping (non-orthogonal) modes of variability, finite length of time, commensurate periods of variability of different modes, etc.



5. Singular Value Decomposition

5.1 Singular Value Decomposition (SVD)

Singular Value Decomposition, or SVD, is one of the most useful linear algebra concepts, with applications in many different fields, from image compression to robotics to the solution of linear equations and more. We consider first its definition, how it is calculated, and then several of its many applications.

5.1.1 Statement, examples and calculation of SVD

Any $m \times n$ matrix can be written using its SVD as,

$$\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^T, \tag{5.1}$$

where U is an $m \times m$ orthonormal matrix ($\mathsf{U}^T\mathsf{U} = \mathsf{U}\mathsf{U}^T = I$), Σ is an $m \times n$ matrix with the "singular values" of A along its diagonal and zeros elsewhere, and V is another $n \times n$ orthonormal matrix. The singular values along the diagonal of Σ are the square roots of the non-zero eigenvalues of $\mathsf{A}\mathsf{A}^T$ or $\mathsf{A}^T\mathsf{A}$ (the non-zero eigenvalues of these two matrices are the same). The columns of U are the eigenvectors of $\mathsf{A}\mathsf{A}^T$, while the columns of V are the eigenvectors of $\mathsf{A}^T\mathsf{A}$.

To get some insight into the above statement of SVD, multiply the SVD decomposition (5.1) by U^T on the left and take transpose of both sides, to find

$$\mathsf{A}^T\mathsf{U} = \mathsf{V}\Sigma^T. \tag{5.2}$$

Similarly, multiply (5.1) by V on the right to find,

$$\mathsf{A}\mathsf{V} = \mathsf{U}\Sigma. \tag{5.3}$$

Writing these last two equations for each column of U and V separately,

$$\mathsf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i, \quad i = 1, \ldots, n \tag{5.4}$$
$$\mathsf{A}^T \mathbf{u}_i = \sigma_i \mathbf{v}_i, \quad i = 1, \ldots, m, \tag{5.5}$$

which has the form of two coupled eigenvalue problems. Multiply the first equation on the left by A^T and use the second, and then multiply the second by A^T and use the first, to find,

$$\mathsf{A}^T\mathsf{A}\mathbf{v}_i = \sigma_i \mathsf{A}^T \mathbf{u}_i = \sigma_i^2 \mathbf{v}_i,$$
$$\mathsf{A}\mathsf{A}^T \mathbf{u}_i = \sigma_i \mathsf{A}\mathbf{v}_i = \sigma_i^2 \mathbf{u}_i.$$

We have therefore shown that the columns of U are the eigenvectors of $\mathsf{A}\mathsf{A}^T$, while the columns of V are the eigenvectors of $\mathsf{A}^T\mathsf{A}$. This also revealed that the singular values σ_i are the square root of the eigenvalues of either $\mathsf{A}\mathsf{A}^T$ or $\mathsf{A}^T\mathsf{A}$.

Note that U and V are orthonormal because their columns are the eigenvectors of the symmetric matrices $\mathsf{A}\mathsf{A}^T$ or $\mathsf{A}^T\mathsf{A}$, which are orthogonal to each other. This is because a symmetric matrix is also normal and its eigenvectors are therefore orthogonal, see proof in section A.1. The eigenvalues of $\mathsf{A}\mathsf{A}^T$ or $\mathsf{A}^T\mathsf{A}$ are non-negative and therefore have real square roots, as these are both positive semi-definite matrices. The proof is essentially identical to the one used in section 4.1.2 to show that the covariance matrix derived from the data matrix is positive definite.

We will run through an example shortly, but first, the algorithm for the calculation of the singular value decomposition is as follows,

1. Consider the two square matrices $\mathsf{A}\mathsf{A}^T$ and $\mathsf{A}^T\mathsf{A}$, calculate the one with smaller dimensions.
2. Calculate the eigenvalues and eigenvectors of the resulting matrix.
3. Insert the square root of the calculated eigenvalues (i.e., the singular values) into Σ with the largest non-zero singular in the top left entry, decreasing down the diagonal.
4. If $\mathsf{A}\mathsf{A}^T$ was used, then its eigenvectors will be the first m columns of U, while if $\mathsf{A}^T\mathsf{A}$ was used, its eigenvectors will be the first n columns of V. Arrange the columns of the resulting matrix (U or V) from left to right with the eigenvector corresponding to the largest singular values to the left, decreasing to the right.
5. Calculate the other orthogonal matrix. If $\mathsf{A}\mathsf{A}^T$ was used to calculate U, then the first m columns of V are calculated using the matrix-form equations (5.2), or the vector form (5.5). Similarly, if $\mathsf{A}^T\mathsf{A}$ was used to calculate V, then the first n columns of U are calculated using (5.3) or (5.4).
6. Finally, complete the rest of the columns of V or U by starting with random vectors and using Gram-Schmidt orthogonalization to make sure the columns of the resulting matrix are orthogonal.

■ **Example 5.1** As a simple numerical example, consider the following matrix,

$$\mathsf{A} = \begin{pmatrix} 3 & 2 & 1 \\ 2 & 3 & 1 \end{pmatrix}$$

and calculate its SVD, $\mathsf{A}_{(n\times m)} = \mathsf{U}_{(n\times n)}\Sigma_{(n\times m)}\mathsf{V}^T_{(m\times m)}$. Because $\mathsf{A}^T\mathsf{A}$ is 3×3 while $\mathsf{A}\mathsf{A}^T$ is 2×2, start with the latter, and use the fact that the U vectors are the eigenvectors of $\mathsf{A}\mathsf{A}^T$.

$$\mathsf{A}\mathsf{A}^T = \begin{pmatrix} 14 & 13 \\ 13 & 14 \end{pmatrix}$$

The eigenvalues of AA^T are 1 and 27 (shown in the matrix D), and the corresponding matrix of eigenvectors, U, is

$$U = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & 0.7071 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 0 \\ 0 & 27 \end{pmatrix}$$

The singular values are square root of the diagonal of D. We rearrange the columns of U such that the first corresponds to the largest singular value, and create the singular values matrix by taking the square root of the above eigenvalues and appending a column of zeros to ensure the correct dimensions,

$$U = \begin{pmatrix} 0.7071 & -0.7071 \\ 0.7071 & 0.7071 \end{pmatrix} \qquad \Sigma = \begin{pmatrix} 5.1962 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Next, find V. We cannot use the eigenvectors of A^TA, because eigenvectors, even when normalized to have unit norm, are uniquely determined **only up to a minus sign**. However, the signs of U and V must be consistent to have $A = U\Sigma V^T$. Arbitrarily multiplying some of the columns of V by a minus sign would violate $A = U\Sigma V^T$. Using $\mathbf{v}_1 = A^T\mathbf{u}_1/\sigma_1$ and $\mathbf{v}_2 = A^T\mathbf{u}_2/\sigma_2$, we find

$$V = \begin{pmatrix} 0.6804 & -0.7071 & 0 \\ 0.6804 & 0.7071 & 0 \\ 0.2722 & 0 & 0 \end{pmatrix}.$$

the columns of U, V corresponding to the zero singular values (in this case, the third column of V) are not needed in most applications, but in case it is needed, start with a random vector and use Gram-Schmidt to get a vector perpendicular to the first two columns: set $\mathbf{v}_3 = (1,1,1)^T$ and perform G-S to find a $\mathbf{v}_3$ perpendicular to the $\mathbf{v}_1, \mathbf{v}_2$ in the matrix above, leading to,

$$V = \begin{pmatrix} 0.6804 & -0.7071 & -0.1925 \\ 0.6804 & 0.7071 & -0.1925 \\ 0.2722 & 0 & 0.9623 \end{pmatrix}.$$

Checking using Matlab we first recreate the original matrix:

$$U\Sigma V^T = \begin{pmatrix} 2.9999 & 2.0000 & 1.0001 \\ 2.0000 & 2.9999 & 1.0001 \end{pmatrix}$$

The result looks good! Now compare to Matlab's output using the `svd` function,

```
[U,S,V]=svd(A)
U =[-0.7071   -0.7071
    -0.7071    0.7071]
S =[5.1962         0         0
         0    1.0000         0]
V =[-0.6804   -0.7071   -0.1925
    -0.6804    0.7071   -0.1925
    -0.2722   -0.0000    0.9623]
```

Note that Matlab's response is different by a minus sign for the first U and first V vectors. This is fine as the two minus signs for $\mathbf{u}_1$ and $\mathbf{v}_1$ cancel out during the reconstruction of $A = U\Sigma V^T = \mathbf{u}_1\sigma_1\mathbf{v}_1 + \mathbf{u}_2\sigma_2\mathbf{v}_2$.

■

5.1.2 Geometric interpretation of SVD

Real square matrices can be used to represent transformations of geometric shapes. For example, consider a two-dimensional shape made of points (Figure 5.1, panel (a)) and consider each point as representing a vector from the origin. Place all vectors as columns in a matrix X. After multiplying each such vector by a matrix A, the shape (now given by the columns of AX) is transformed to a different one (Figure 5.1, panel b). Consider now the sequence of transformations given by the SVD decomposition of A.

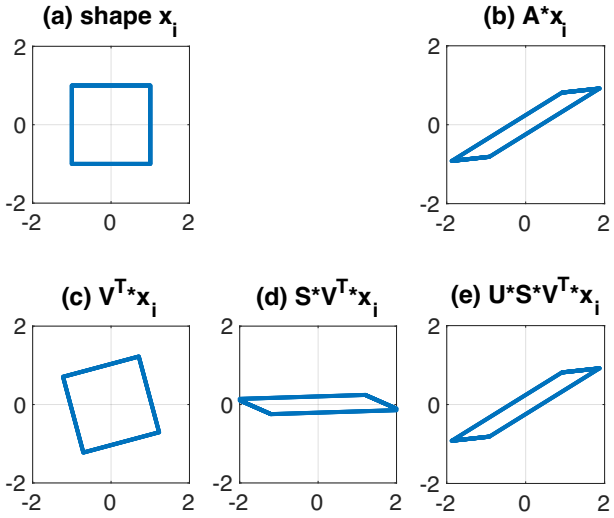


Figure 5.1: Geometric interpretation of SVD

When we carry out a singular value decomposition on a transformation matrix, A, we can think about each component, the resultant U, Σ and V^T matrices, as carrying out a part of the transformation.

■ **Example 5.2** Let us take the following matrix, A, as an example, using the diagram in Fig. 5.1.

$$A = \begin{pmatrix} 1.396 & -0.4776 \\ 0.8659 & -0.05269 \end{pmatrix}$$

Calculating the SVD,

$$U = \begin{pmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 1.7 & 0 \\ 0 & 0.2 \end{pmatrix}, \qquad V = \begin{pmatrix} 0.9659 & 0.2588 \\ -0.2588 & 0.9659 \end{pmatrix}$$

Recall that in transformations, $AX = (U\Sigma V^T)X$ implies the data X is transformed by V^T then by Σ and last by U. Here, V^T rotates the initial square spanned by unit vectors anti-clockwise by 30°. Σ applies a stretch to the square, creating a parallelogram, with the y-coordinates stretched by a factor of 0.2, and the x-coordinates stretched by a factor of 1.7. Finally, U applies a counter-clockwise rotation by 45°. Thus, we see the piecewise transformations applied to our original shape achieves the same transformation as the original A matrix. We will see this geometric interpretation of SVD in our discussion of polar decomposition. ■

See Matlab/python demos here; as well as an animation and caption from Wikipedia by Kieff, with some more details here.

5.2 SVD applications

5.2.1 Image compression, low-rank approximation

Sending or downloading images (e.g., from a satellite mission to Earth) often requires that we only store and send the essential components of the image. Say an image is $N \times N$ pixels, represented by a matrix, A, of N^2 numbers, each representing a gray scale value at a particular point in the image. SVD may be used to reduce the amount of data representing the image – that is, compress it. Given the SVD of A, the singular values determine the contribution of each of the U and V^T columns to the image, which becomes more apparent if we write the SVD as,

$$\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^T = \mathbf{u}_1\sigma_1\mathbf{v}_1^T + \mathbf{u}_2\sigma_2\mathbf{v}_2^T + \mathbf{u}_3\sigma_3\mathbf{v}_3^T + \cdots + \mathbf{u}_r\sigma_r\mathbf{v}_r^T$$

where r is the rank (defined for now as the number of non-zero singular values) of A, and the singular values are shown in order of descending magnitude.

If the matrix representing the image is dominated by a few, say k, large singular values, then the image matrix can be reconstructed using only the contributions from U and V vectors associated with these largest singular values. If we keep k singular values, and disregard the other $N-k$, we need to store k singular values plus (k terms in the expansion) times (k u,v eigenvectors) times (N values per vector) = $2kN+k$ numbers instead of the original N^2, which can be a significant saving when N is large and if the image can be reproduced well with a small k. Note that if we use all SVD modes for the reconstruction, then the required data includes both the U and V matrices plus N singular values, or $2N^2+N$, which amounts to approximately twice the number of entries in the original matrix, thus certainly not representing a compression...

An example of carrying out this SVD image compression on an image of a Baboon is shown in Fig. 5.2 and is given in the demo `SVD_applications_image_compression.m/py`.

The demo also shows the "explained variance" of the image, which is defined and calculated as follows. Let the image be $\mathsf{X}_{m\times n}$,

$$\mathsf{X}^T\mathsf{X} = (\mathsf{U}\Sigma\mathsf{V}^T)^T(\mathsf{U}\Sigma\mathsf{V}^T) = \mathsf{V}\Sigma^2\mathsf{V}^T$$

The total variance is sum of diagonal elements of $\mathsf{C} = \mathsf{X}^T\mathsf{X}/N$, that are given by,

$$C_{ii} = \sum_j X_{ij}X_{ij}/N = \sum_j V_{ij}V_{ji}\sigma_i^2/N = \sigma_i^2/N$$

The total variance is therefore sum of singular values squared divided by N, and the fraction of the variance explained by the first k modes is therefore,

$$\frac{\sum_{i=1}^{k}\sigma_i^2}{\sum_{i=1}^{r}\sigma_i^2}.$$

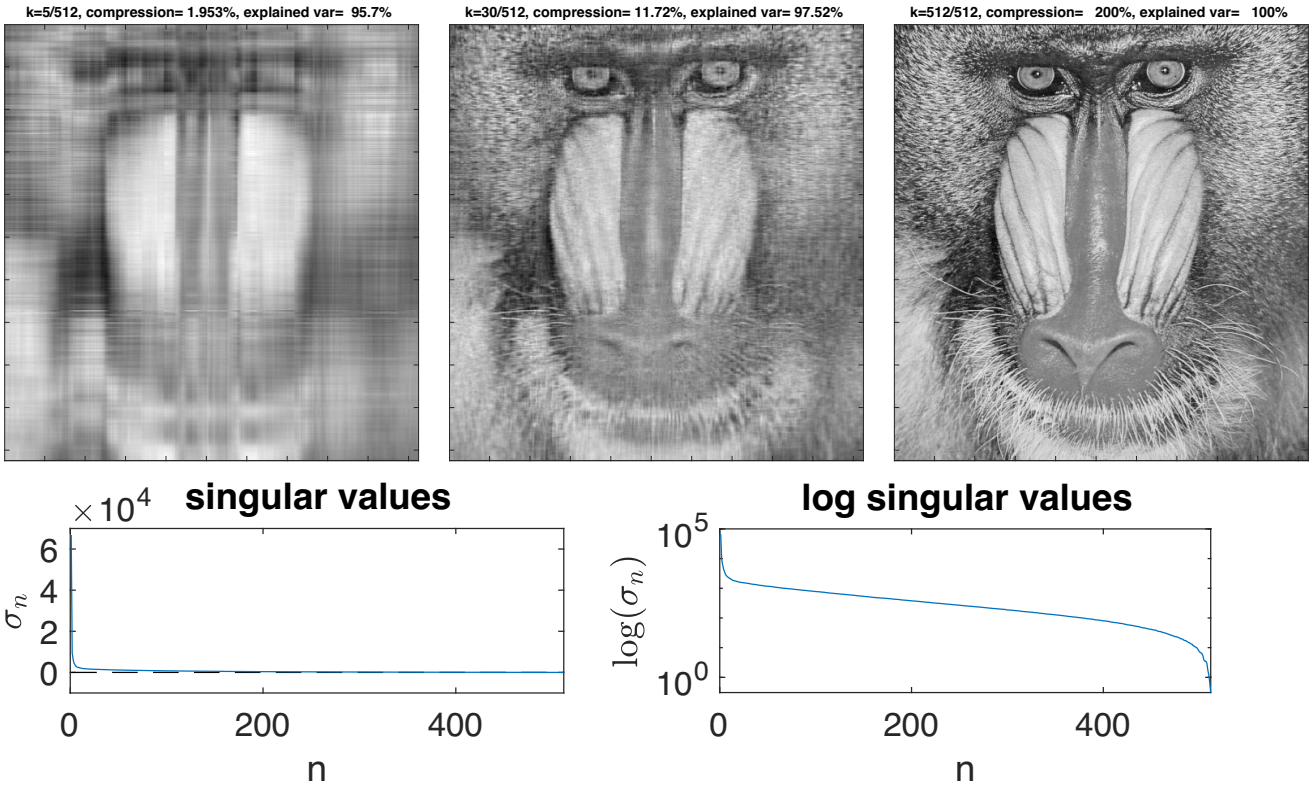


Figure 5.2: Image compression using SVD

5.2.2 Effective rank of a matrix

The "rank" of a matrix is defined as the number of independent rows or columns (smaller of the two, if the matrix is not square), and is equal to the number of non-zero eigenvalues. However, in practice, matrices may have columns that are *nearly* parallel, and treating them as independent can substantially ill-affect the solution of a linear system of equations based on the matrix. The "effective rank" of a matrix addresses this issue. Consider two examples matrices, where ε is assumed small,

$$\mathsf{A}_1 = \begin{pmatrix} 1 & 2 \\ 2 & 4+\varepsilon \end{pmatrix},$$
$$\mathsf{A}_2 = \mathsf{A}_1 \times 10^6.$$

For a finite value of ε, the two rows of A_1 are different and A_1's rank (number of non-zero eigenvalues) is 2. However, if ε is smaller than the error in the measurements that determined the values of A and $\mathbf{b}$, it would make sense to treat ε as very small noise, in which case the two equations represented by the matrix are effectively the same, and the rank of the matrix is effectively only 1. This implies that there are fewer equations than unknowns, and we will see in sections 5.2.5, 5.2.6 and 5.2.7 how to solve such systems where the numbers of equations and unknowns are not the same.

To determine the number of independent rows of a matrix, it is not wise to use the absolute value of the eigenvalues – that is, treat eigenvalues smaller than some threshold as zero – or similarly, assume that if the determinant is below some threshold it should be considered zero. In the above example, the

determinant and eigenvalues of A_2 are much larger than that of A_1, although it is equally ill-conditioned. Setting $\varepsilon = 10^{-6}$, the singular values of both matrices are,

$$\text{diag}(\Sigma_1) = (5, 0.0000002), \qquad \text{diag}(\Sigma_2) = 10^6(5, 0.0000002).$$

To determine the effective rank, let the specified relative error in the entries of the matrix and RHS be ε. Find the value of k for which $\sigma_k/\sigma_1 > \varepsilon$, while $\sigma_{k+1}/\sigma_1 < \varepsilon$. This k is the effective rank of the matrix. Assuming the relative error in the entries of A and $\mathbf{b}$ is, say, $\varepsilon = 0.001$, this criterion would suggest that both matrices above are of rank 1.

5.2.3 Matrix norm and condition number

We now revisit the conditioning of matrices, as discussed in earlier in the course (section 2.5). We are interested in understanding how sensitive the solution to a linear system is to noise in A or in the RHS $\mathbf{b}$, and how we can determine whether a matrix is well-conditioned or ill-conditioned.

Consider a linear equation $\mathsf{A}\mathbf{x} = \mathbf{b}$. Adding a small perturbation to the RHS, the solution will change to $\mathbf{x} + \delta\mathbf{x}$. The revised equation is,

$$\mathsf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$$

Subtracting the original equation leads to the "error equation" for the error in the solution due to a noise in the RHS,

$$\mathsf{A}\delta\mathbf{x} = \delta\mathbf{b}. \tag{5.6}$$

Intuitively, the error $\delta\mathbf{x} = \mathsf{A}^{-1}\delta\mathbf{b}$ in the solution would be large, when A^{-1} is large (in some yet-to-be-defined sense) or equivalently, when A is nearly singular. This motivates a discussion of the "condition number" of a matrix, which, in turn, depends on the definition of the norm of the matrix. Both are calculated using SVD.

Start by defining the "matrix norm", which can be thought of as the maximum stretching effect a matrix A can have on a vector,

$$\|\mathsf{A}\| = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathsf{A}\mathbf{x}\|}{\|\mathbf{x}\|} \tag{5.7}$$

We next show that the matrix norm is equal the largest singular value (or the square root of the largest eigenvalue). Start by rewriting the norm definition as,

$$\|A\|^2 = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathsf{A}\mathbf{x}\|^2}{\|\mathbf{x}\|^2} = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\mathbf{x}^T\mathsf{A}^T\mathsf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}.$$

Given that $\|\mathbf{x}\| \neq 0$, we can divide the numerator and denominator of the above definition by the norm of $\mathbf{x}$ and therefore assume that it is equal to 1. This leads to a revised definition,

$$\|\mathbf{x}\|^2 = \max_{\|\mathbf{x}\|=1} \mathbf{x}^T\mathsf{A}^T\mathsf{A}\mathbf{x}.$$

To find this maximum, use constrained optimization using a Lagrange multiplier,

$$J = \mathbf{x}^T\mathsf{A}^T\mathsf{A}\mathbf{x} + \lambda(1 - \mathbf{x}^T\mathbf{x})$$
$$J = \sum_{i,j,k} x_i A^T_{ij} A_{jk} x_k + \lambda(1 - \sum_k x_k^2).$$

The maximum occurs when,

$$\begin{aligned} 0 = \frac{dJ}{dx_m} &= \sum_{j,k} A^T_{mj} A_{jk} x_k + \sum_{i,j} x_i A i j^T A_{im} - 2\lambda x_m \\ &= \sum_{j,k} A^T_{mj} A_{jk} x_k + \sum_{j,i} A^T_{mj} A_{ji} x_i - 2\lambda x_m. \end{aligned}$$

Renaming index i to k in the second sum,

$$0 = \frac{dJ}{dx_m} = 2\sum_{j,k} A^T_{mj} A_{jk} x_k - 2\lambda x_m = 2(\mathsf{A}\mathsf{A}^T\mathbf{x} - \lambda\mathbf{x}).$$

Therefore, the maximum occurs when $\mathsf{A}^T\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$; that is, when $\mathbf{x}$, λ are eigenvector, eigenvalue of $\mathsf{A}^T\mathsf{A}$. At this point, we can calculate $\|\mathsf{A}\|^2$,

$$\begin{aligned} \|\mathsf{A}\|^2 &= \frac{\mathbf{x}^T\mathsf{A}^T\mathsf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \\ &= \frac{\mathbf{x}^T\lambda\mathbf{x}}{\mathbf{x}^T\mathbf{x}} = \lambda. \end{aligned}$$

The desired maximum of the squared norm therefore occurs when $\mathbf{x}$ is the eigenvector of $\mathsf{A}^T\mathsf{A}$ corresponding to its largest eigenvalue, and the value of the square of the matrix norm is the eigenvalue of $\mathsf{A}^T\mathsf{A}$, and thus equal to the square of the largest singular value, $\lambda = \sigma_1^2$. Thus, the norm of A is given by the largest singular value,

$$\|\mathsf{A}\| = \sigma_1.$$

Next, derive the condition number, c, of a matrix A, which provides a measure of the amplification of noise in the RHS. Let us compare the relative error in the solution, $\|\delta\mathbf{x}\|/\|\mathbf{x}\|$, with the relative change to the RHS, $\|\delta\mathbf{b}\|/|\mathbf{b}\|$. We write,

$$\|\mathsf{A}\mathbf{x}\| = \|\mathbf{b}\| \leq \|\mathsf{A}\| \cdot \|\mathbf{x}\|$$

so that,

$$\|\mathbf{x}\| \geq \frac{\|\mathbf{b}\|}{\|\mathsf{A}\|}.$$

Similarly, from the error equation (5.6) we find,

$$\|\delta\mathbf{x}\| = \|\mathsf{A}^{-1}\mathbf{b}\| \leq \|\mathsf{A}^{-1}\| \cdot \|\delta\mathbf{b}\|$$

so that,

$$\|\delta\mathbf{x}\| \leq \|\mathsf{A}^{-1}\| \cdot \|\delta\mathbf{b}\|.$$

Dividing the last equation by the norm $\|\mathbf{x}\|$ and using the above bound that $\|\mathbf{x}\| \geq \|\mathbf{b}\|/\|\mathsf{A}\|$ we find,

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|\mathsf{A}\| \cdot \|\mathsf{A}^{-1}\| \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

The number $c = \|\mathsf{A}\| \cdot \|\mathsf{A}^{-1}\|$ is termed the condition number. Using our above result that $\|\mathsf{A}\| = \sigma_{\max}$ and therefore also that $\|\mathsf{A}^{-1}\| = \sigma_{\min}$, where $\sigma_{\max,\min}$ are the largest and smallest singular values of A, we can equivalently write the condition number as,

$$\begin{aligned} c &= \|\mathsf{A}\| \cdot \|\mathsf{A}^{-1}\| = \sigma_{\max}/\sigma_{\min}, \\ \frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} &\leq c\,\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}. \end{aligned} \tag{5.8}$$

The *worst case* scenario in terms of the amplification of the noise $\delta\mathbf{b}$ into a large error in the solution, $\delta\mathbf{x}$, occurs when the relative error in the solution (LHS of 5.8) is equal to its upper bound (RHS of 5.8). That is, the relative error in the solution is as large as it can be, given added noise in the RHS. This occurs when $\mathbf{x}$ is proportional to the singular $\mathbf{v}$ vector corresponding to the largest singular value of A and $\delta\mathbf{x}$ is proportional to the smallest one. In that case, take the norm of the RHS and LHS of the equation and of the error equation and divide to find,

$$\frac{\|\mathsf{A}\delta\mathbf{x}\|^2}{\|\mathsf{A}\mathbf{x}\|^2} = \frac{\|\delta\mathbf{b}\|^2}{\|\mathbf{b}\|^2}.$$

Write the squared norm on the LHS explicitly,

$$\frac{(\mathsf{A}\delta\mathbf{x})^T(\mathsf{A}\delta\mathbf{x})}{(\mathsf{A}\mathbf{x})^T(\mathsf{A}\mathbf{x})} = \frac{\|\delta\mathbf{b}\|^2}{\|\mathbf{b}\|^2}$$

or

$$\frac{\delta\mathbf{x}^T(\mathsf{A}^T\mathsf{A})\delta\mathbf{x}}{\mathbf{x}^T(\mathsf{A}^T\mathsf{A})\mathbf{x}} = \frac{\|\delta\mathbf{b}\|^2}{\|\mathbf{b}\|^2}.$$

Using the fact that $\mathbf{x}$ and $\delta\mathbf{x}$ are chosen to be the largest and smallest singular vectors,

$$\frac{\sigma_{\min}^2\|\delta\mathbf{x}\|^2}{\sigma_{\max}^2\|\mathbf{x}\|^2} = \frac{\|\delta\mathbf{b}\|^2}{\|\mathbf{b}\|^2},$$

so that,

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} = \frac{\sigma_{\max}}{\sigma_{\min}}\,\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} = c\,\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

This implies that the worst case noise structure is proportional to $\delta\mathbf{b} = \mathsf{A}^{-1}\delta\mathbf{x}$.

See demo SVD_applications_matrix_rank_norm_condition_number.m/py).

5.2.4 Polar decomposition and the Kabsch algorithm

Polar decomposition is used in continuum mechanics and robotics, in applications requiring the ability to separate rotations from stretches. Below we show an application of polar decomposition in the comparison and identification of molecular structures.

Matrix polar decomposition is motivated by the representation of a complex number $z = x + iy$ in terms of an amplitude and a rotation angle, as $z = re^{i\theta}$. Matrix polar decomposition similarly writes a real, *square* matrix A as the product of a symmetric positive semi-definite matrix (the "stretching" or "amplitude"), S, and an orthogonal matrix, R, representing a rotation. If A is invertible, then R is positive definite.

SVD allows us to calculate this decomposition into $\mathsf{A} = \mathsf{R}_1\mathsf{S}_1$ or into $\mathsf{A} = \mathsf{S}_2\mathsf{R}_2$ by inserting $\mathsf{V}^T\mathsf{V} = \mathsf{I}$ into the SVD,

$$\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^T = (\mathsf{U}\mathsf{V}^T)(\mathsf{V}\Sigma\mathsf{V}^T) = \mathsf{R}_1\mathsf{S}_1$$
$$\mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^T = (\mathsf{U}\Sigma\mathsf{U}^T)(\mathsf{U}\mathsf{V}^T) = \mathsf{S}_2\mathsf{R}_2$$

The factor R_1 is orthogonal because $\mathsf{R}_1^T\mathsf{R}_1 = (\mathsf{U}\mathsf{V}^T)^T(\mathsf{U}\mathsf{V}^T) = \mathsf{V}\mathsf{U}^T\mathsf{U}\mathsf{V}^T = \mathsf{I}$, and S_1 is symmetric and semi-definite because Σ is. The same applies for R_2 and S_2.

With A, a transformation matrix (e.g., as in section 5.1.2), the decomposition $\mathsf{A} = \mathsf{R}\mathsf{S}$ breaks the transformation into a stretching followed by a rotation. Figure 5.3, based on the demo, here, shows the geometric interpretation of polar decomposition. Note the difference from the geometric interpretation of SVD itself. In that case, two rotations were used, before and after the stretching step. As a result, the stretching was done along the x and y axes. Here, only one rotation is involved, and the stretching and/or compression occur in some general directions, depending on the matrix S, that are not necessarily the directions of the two axes.

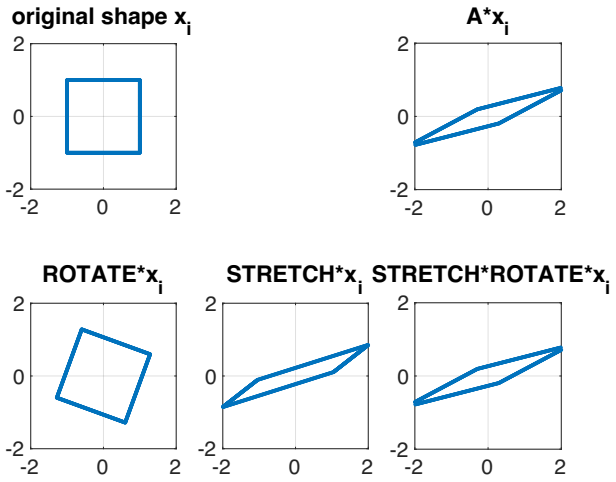


Figure 5.3: Geometric interpretation of polar decomposition.

Kabsch algorithm for comparing molecular structures

Start with the location of atoms in a molecule, given as a set of N 3×1 column vectors $\mathbf{x}_i$, and written as a $3 \times N$ matrix $\mathsf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N] = x_{ij}$. Consider another molecule given by a similar matrix $\mathsf{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_N] = y_{ij}$.

Next, shift both molecules to the origin by calculating and subtracting the mean location of atoms in each molecule,

$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i; \qquad \hat{\mathbf{x}}_i = \mathbf{x}_i - \bar{\mathbf{x}}, \qquad \hat{x}_{ij} = [\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_N]$$
$$\bar{\mathbf{y}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{y}_i; \qquad \hat{\mathbf{y}}_i = \mathbf{y}_i - \bar{\mathbf{y}}, \qquad \hat{y}_{ij} = [\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_N],$$

where hat represents the coordinates of the molecules shifted toward the origin, and $\hat{x}_{ij}$ is the ith coordinate (out of 3 space coordinates) of the jth shifted molecule.

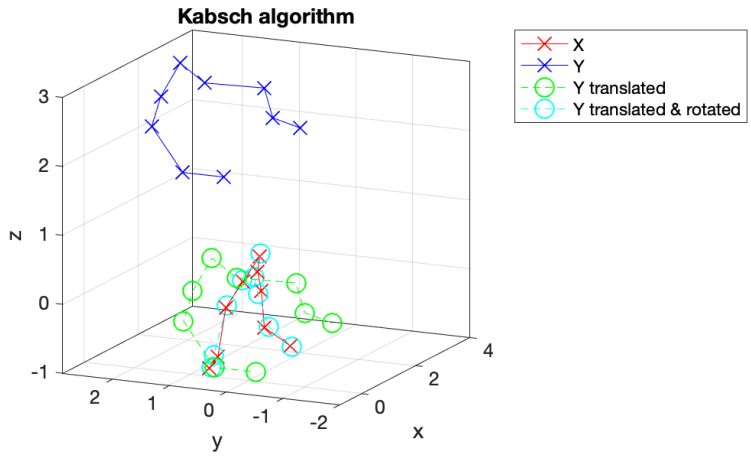


Figure 5.4: Demonstrating the Kabsch algorithm.

Next, calculate the covariance matrix of the two shifted molecules,

$$\mathsf{C} = \hat{\mathsf{X}}\hat{\mathsf{Y}}^T, \qquad c_{ij} = \sum_{k=1}^{N} \hat{x}_{ik}\hat{y}_{jk}.$$

Derive the SVD decomposition of the covariance matrix,

$$\mathsf{C} = \mathsf{U}\Sigma\mathsf{V}^T,$$

which can be used to derive two different polar decomposition forms, using $\mathsf{V}^T\mathsf{V} = \mathsf{U}^T\mathsf{U} = I$,

$$\mathsf{C} = \mathsf{U}\Sigma\mathsf{V}^T = \mathsf{U}(\mathsf{V}^T\mathsf{V})\Sigma\mathsf{V}^T = (\mathsf{U}\mathsf{V}^T)(\mathsf{V}\Sigma\mathsf{V}^T) = \mathsf{Q}\mathsf{S}_1$$
$$\mathsf{C} = \mathsf{U}\Sigma\mathsf{V}^T = \mathsf{U}\Sigma(\mathsf{U}^T\mathsf{U})\mathsf{V}^T = (\mathsf{U}\Sigma\mathsf{U}^T)(\mathsf{U}\mathsf{V}^T) = \mathsf{S}_2\mathsf{Q}$$

where Q is an orthogonal matrix $\mathsf{Q}^T\mathsf{Q} = \mathsf{I}$ and therefore represents a pure rotation, while S_1 and S_2 are positive definite symmetric matrices that represent stretching/compression. For the comparison of molecular structures, rotate the Y matrix using,

$$\hat{\hat{\mathsf{Y}}} = \mathsf{Q}\hat{\mathsf{Y}}$$

and now $\hat{\mathsf{X}}$ and $\hat{\hat{\mathsf{Y}}}$ should be as aligned as possible, and we can decide if they represent the same molecules or two different ones, as shown in Figure 5.4.

5.2.5 Least squares, over-determined systems

Consider M equations and N unknowns, in the form, $\mathsf{A}\mathbf{x} = \mathbf{b}$ where there may be more equations than unknowns (over-determined systems), and even inconsistent equations. That is, the $M \times N$ matrix A is not square, and $M > N$. An example with $M = 3$ and $N = 2$,

$$x = 1, \quad y = 2, \quad x + y = 4.$$

which may be written in the form $\mathsf{A}\mathbf{x} = \mathbf{b}$, where

$$\mathsf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}.$$

An exact solution does not exist, but we can look for a solution that minimizes the norm $\mathbf{e}^T\mathbf{e}$ of the error $\mathbf{e} = \mathsf{A}\mathbf{x} - \mathbf{b}$. For this, look for $\mathbf{x}$ that satisfies

$$\begin{aligned} 0 &= \frac{\partial}{\partial \mathbf{x}} \left(\mathbf{e}^T\mathbf{e}\right) = \frac{\partial}{\partial \mathbf{x}} \left((\mathsf{A}\mathbf{x} - \mathbf{b})^T (\mathsf{A}\mathbf{x} - \mathbf{b})\right) \\ &= \frac{\partial}{\partial \mathbf{x}} \left(\mathbf{x}^T \mathsf{A}^T \mathsf{A}\mathbf{x} - \mathbf{b}^T \mathsf{A}\mathbf{x} - \mathbf{x}^T \mathsf{A}^T \mathbf{b}\right) = 2\left(\mathsf{A}^T \mathsf{A}\mathbf{x} - \mathsf{A}^T \mathbf{b}\right) \end{aligned}$$

or, done more carefully,

$$\begin{aligned} 0 &= \frac{\partial}{\partial \mathbf{x}} \left(\mathbf{e}^T\mathbf{e}\right) = \frac{\partial}{\partial \mathbf{x}} \left((\mathsf{A}\mathbf{x} - \mathbf{b})^T (\mathsf{A}\mathbf{x} - \mathbf{b})\right) \\ &= \frac{\partial}{\partial \mathbf{x}} \sum_{i=1}^{M} (\mathsf{A}\mathbf{x} - \mathbf{b})_i (\mathsf{A}\mathbf{x} - \mathbf{b})_i = \frac{\partial}{\partial x_n} \sum_{i=1}^{M} \left(\sum_{j=1}^{N} A_{ij} x_j - b_i\right) \left(\sum_{k=1}^{N} A_{ik} x_k - b_i\right) \\ &= \frac{\partial}{\partial x_n} \sum_{i=1}^{M} \left(\sum_{(j,k)=1}^{N} A_{ij} A_{ik} x_j x_k - \sum_{k=1}^{N} A_{ik} x_k b_i - \sum_{j=1}^{N} A_{ij} x_j b_i + b_i b_i\right) \\ &= \sum_{i=1}^{M} \left(\sum_{k=1}^{N} A_{in} A_{ik} x_k + \sum_{j=1}^{N} A_{ij} A_{in} x_j - A_{in} b_i - A_{in} b_i\right) = 2\left(\mathsf{A}^T \mathsf{A}\mathbf{x} - \mathsf{A}^T \mathbf{b}\right). \end{aligned}$$

The optimal solution satisfies

$$\mathsf{A}^T \mathsf{A}\mathbf{x} = \mathsf{A}^T \mathbf{b}, \tag{5.9}$$

and the solution is, therefore,

$$\mathbf{x} = (\mathsf{A}^T \mathsf{A})^{-1} \mathsf{A}^T \mathbf{b}. \tag{5.10}$$

This is referred to as the least-squares solution because it minimizes the sum of squares of the elements of the error vector $\mathbf{e}$. In practice, for a large problem, it makes more sense to solve (5.9) using Gaussian elimination, rather than to invert the matrix $\mathsf{A}^T\mathsf{A}$ on the LHS and use the expression (5.10).

■ **Example 5.3** Consider the equations $2x = 3$, $3x = 4$. To solve, write $A = [2;3]$, $\mathbf{b} = [3;4]$, $\mathsf{A}^T\mathsf{A} = 13$, $\mathsf{A}^T\mathbf{b} = 18$. Optimal solution is therefore 18/13. The residual, while optimally small, is not zero if the equations are not consistent, and is equal to $\mathsf{A}\mathbf{x} - \mathbf{b}$, in this case,

$$\mathbf{r} = \mathsf{A}\mathbf{x} - \mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} (18/13) - \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} -0.23 \\ 0.15 \end{pmatrix}.$$

■

5.2.5.1 Over-determined problems and QR decomposition

Suppose we need to solve an over-determined problem $\mathsf{A}\mathbf{x} = \mathbf{b}$ (more equations than unknowns) for many right hand sides. Remember (equation 3.5) that we can write the Gram-Schmidt orthogonalization of the vectors represented by the columns of a matrix A as $\mathsf{A} = \mathsf{QR}$, with R upper-diagonal and Q orthogonal. Using this, write,

$$\begin{aligned}
\mathsf{A}^T\mathsf{A}\mathbf{x} &= A^T\mathbf{b}, \\
(\mathsf{QR})^T(\mathsf{QR})\mathbf{x} &= (\mathsf{QR})^T\mathbf{b}, \\
\mathsf{R}^T\mathsf{Q}^T\mathsf{QR}\mathbf{x} &= \mathsf{R}^T\mathsf{Q}^T\mathbf{b}, \\
\mathsf{R}^T\mathsf{R}\mathbf{x} &= \mathsf{R}^T\mathsf{Q}^T\mathbf{b}, \\
\mathsf{R}\mathbf{x} &= \mathsf{Q}^T\mathbf{b}.
\end{aligned}$$

The last equation is easily solved because R is upper diagonal, and thus we may use back substitution to solve for $\mathbf{x}$, at a cost much smaller than solving the equation (5.9). So if the over-determined problem needs to be solved many times for many different $\mathbf{b}$, QR provides an efficient algorithm.

5.2.6 Under-determined systems and the pseudo inverse

If $\mathsf{A}_{(M\times N)}\mathbf{x}_{(N\times 1)} = \mathbf{b}_{(M\times 1)}$ has fewer equations than unknowns, the problem is referred to as under-determined and there may be many possible solutions. SVD allows us to obtain a solution for such problems. A simple example is $x+y=2$, corresponding to $\mathsf{A} = (1,\ 1)$, $\mathbf{b} = 2$, which has an infinite number of solutions. A unique solution to this underdetermined problem may be found by adding a requirement that $\mathbf{x}$ has the smallest possible norm, and with this requirement $\mathbf{x}$ can then be found using SVD.

Our first step is to define the "pseudo inverse" of A, denoted $\mathsf{A}^\dagger$. For an $M \times N$ matrix Σ with non-zero elements only along the diagonal, the pseudo inverse is defined as the transpose of the original Σ, with the non-zero diagonal elements replaced by their inverse, and with the matrix transposed,

$$\Sigma = \begin{pmatrix} \sigma_1 & & & & & 0\ldots 0 \\ & \ddots & & & & 0\ldots 0 \\ & & \sigma_r & & & 0\ldots 0 \\ & & & 0 & & 0\ldots 0 \\ & & & & \ddots & 0\ldots 0 \\ & & & & 0 & 0\ldots 0 \end{pmatrix}_{(M\times N)} \qquad \Sigma^\dagger = \begin{pmatrix} \sigma_1^{-1} & & & & & \\ & \ddots & & & & \\ & & \sigma_r^{-1} & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \\ & & \vdots & & & \\ 0 & & & & & 0 \end{pmatrix}_{(N\times M)}$$

Given a general matrix with the SVD $\mathsf{A}_{(M\times N)} = \mathsf{U}_{(M\times M)}\Sigma_{(M\times N)}\mathsf{V}^T_{(N\times N)}$, we define its "pseudo inverse" as $\mathsf{A}^\dagger_{(N\times M)} = \mathsf{V}\Sigma^\dagger\mathsf{U}^T$. To see why it is called the pseudo inverse, consider $\mathsf{A}^\dagger\mathsf{A}$ and $\mathsf{A}\mathsf{A}^\dagger$. First,

note that,

$$
\begin{aligned}
&(\mathsf{A}^\dagger\mathsf{A})_{(N\times N)} = (\mathsf{V}_{(N\times N)}\Sigma^\dagger_{(N\times M)}\mathsf{U}^T_{(M\times M)})(\mathsf{U}_{(M\times M)}\Sigma_{(M\times N)}\mathsf{V}^T_{(N\times N)}) \\
&= \mathsf{V}\Sigma^\dagger\mathsf{U}^T\mathsf{U}\Sigma\mathsf{V}^T = \mathsf{V}\Sigma^\dagger\Sigma\mathsf{V}^T \\
&= \mathsf{V}_{(N\times N)}\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix}_{(N\times N)} \mathsf{V}^T_{(N\times N)}
\end{aligned} \tag{5.11}
$$

Similarly,

$$
\begin{aligned}
&(\mathsf{A}\mathsf{A}^\dagger)_{(M\times M)} = (\mathsf{U}_{(M\times M)}\Sigma_{(M\times N)}\mathsf{V}^T_{(N\times N)})(\mathsf{V}_{(N\times N)}\Sigma^\dagger_{(N\times M)}\mathsf{U}^T_{(M\times M)}) \\
&= \mathsf{U}\Sigma\mathsf{V}^T\mathsf{V}\Sigma^\dagger\mathsf{U}^T = \mathsf{U}\Sigma\Sigma^\dagger\mathsf{U}^T \\
&= \mathsf{U}_{(M\times M)}\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix}_{(M\times M)} \mathsf{U}^T_{(M\times M)}
\end{aligned} \tag{5.12}
$$

Note that the number of non-zero elements in the diagonal matrix is r, which may smaller or equal to $\min(M,N)$, in general, and is the rank of the matrix A. Consider now an under-determined system, with fewer equations than unknowns, $M < N$. If the rank of A is equal to the number of equations, $r = M < N$, then based on (5.11) and (5.12), we have $(\mathsf{A}\mathsf{A}^\dagger)_{(M\times M)} = \mathsf{I}_{(M\times M)}$ but $(\mathsf{A}^\dagger\mathsf{A})_{(N\times N)} \neq \mathsf{I}_{(N\times N)}$. This justifies the "pseudo inverse" terminology. Given this, it is easy to see that a solution in this particular case is given by

$$\mathbf{x}^\dagger = \mathsf{A}^\dagger\mathbf{b}, \tag{5.13}$$

because if we substitute $\mathbf{x}^\dagger = \mathsf{A}^\dagger\mathbf{b}$ in the matrix equation we find $\mathsf{A}\mathbf{x}^\dagger = \mathsf{A}\mathsf{A}^\dagger\mathbf{b} = \mathsf{I}\mathbf{b} = \mathbf{b}$.

Next, show that the solution $\mathbf{x}^\dagger = \mathsf{A}^\dagger\mathbf{b}$ is also of the smallest possible norm. Note that $\mathbf{x}$ has the dimension of the columns of V. Given that $\sigma_{r+1},\ldots,\sigma_M = 0$, we can write $\mathbf{x}^\dagger$ as an expansion in first r $\mathbf{v}_i$ vectors,

$$\mathbf{x} = \mathsf{A}^\dagger\mathbf{b} = \mathsf{V}\Sigma^\dagger\mathsf{U}^T\mathbf{b} = \sum_{i=1}^{r}\frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}\mathbf{v}_i.$$

The rest of the $\mathbf{v}_i$ vectors are in the null space of A,

$$\mathsf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i = 0, \quad i = r+1,\ldots,N$$

and the solution $\mathbf{x}^\dagger$ does not contain any of these null vectors. We can construct other solutions by adding any vector $\mathbf{x}_{\text{null}}$ that is a superposition of the null vectors of A,

$$\mathbf{x}_{\text{null}} = \sum_{i=r+1}^{N} c_i\mathbf{v}_i.$$

Because $\mathsf{A}\mathbf{x}_{\text{null}} = 0$, $\mathbf{x} = \mathbf{x}^\dagger + \mathbf{x}_{\text{null}}$ is also a solution to $\mathsf{A}\mathbf{x} = \mathbf{b}$. Any such solution, however, will have a larger magnitude than $\mathbf{x}^\dagger$, because the null and non-null parts are orthogonal (based on orthogonality of V vectors) and therefore satisfy Pythagoras' theorem,

$$\|\mathbf{x}\|^2 = \|\mathbf{x}^\dagger + \mathbf{x}_{\text{null}}\|^2 = \|\mathbf{x}^\dagger\|^2 + \|\mathbf{x}_{\text{null}}\|^2 \geq \|\mathbf{x}^\dagger\|^2.$$

We thus showed that $\mathbf{x}^\dagger$ is the smallest norm solution to the under-determined ($M < N$) full rank ($r = M$) problem.

For an example, see `SVD_application_underdetermined_linear_eqns.m/py`.

5.2.7 Solving linear systems when $r < \min(N, M)$

When solving systems that are *formally* over-determined, in the sense that there are fewer unknowns than equations, $\mathsf{A}_{(M\times N)}\mathbf{x}_{(N\times 1)} = \mathbf{b}_{(M\times 1)}$, $N < M$, we cannot use the methods discussed in section 5.2.5 if $(\mathsf{A}^T\mathsf{A})_{(N\times N)}$ is not full rank (non-invertible), that is, if $r < N$, because this matrix needs to be inverted to find the solution (5.10). Similarly, when solving an under-determined system with $M < N$, if the rank of A is less than the number of equations, $r < M$, we cannot use the solution (5.13) discussed in section 5.2.6 because as shown there $\mathsf{A}^\dagger\mathsf{A} \neq 1$ when $r < M$. Note that these issues occur in both cases when $r < \min(N, M)$. We first demonstrate these cases with simple examples.

Start with a rank-deficient under-determined problem,

$$\begin{aligned} x + y + z &= 2 \\ x + y + z &= 3 \end{aligned}$$

with $M = 2$ equations and $N = 3$ unknown. The matrix is,

$$\mathsf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

and its rank is clearly $r = 1 < M = 2$, and there is no solution because the equations are contradictory. Thus, we must resort to looking for a solution that minimizes the norm of the error, $\mathbf{e} = \mathsf{A}\mathbf{x} - \mathbf{b}$, rather than one that solves the above equations. A second example, of a rank-deficient over-determined problem is,

$$\begin{aligned} x + y &= 2 \\ x + y &= 3 \\ x + y &= 4, \end{aligned}$$

where the matrix is

$$\mathsf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}.$$

While there are more equations ($M = 3$) than unknowns ($N = 2$), the rank of the matrix is only $r = 1 < N = 2$, and we find,

$$\mathsf{A}^T\mathsf{A} = \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}$$

whose rank is also 1. $\mathsf{A}^T\mathsf{A}$ therefore cannot be inverted to solve using the least-squares solution (5.10).

The remedy to both pathologies demonstrated by these examples is to seek a solution that minimizes the error, such that the solution's norm is as small as possible – a combination of the approaches we used for the over-determined and under-determined cases. We already know that the solution that minimizes the error satisfies $\mathsf{A}^T\mathsf{A}\mathbf{x} = \mathsf{A}^T\mathbf{b}$. Given that $\mathsf{A}^T\mathsf{A}$ is not full-rank, there are an infinite number of solutions. We are therefore looking for the smallest norm solution for this equation, and as before, this is obtained by using the pseudo inverse,

$$\mathbf{x} = (\mathsf{A}^T\mathsf{A})^\dagger \mathsf{A}^T\mathbf{b}.$$

It turns out that this may be written simply as $\mathbf{x} = \mathsf{A}^\dagger\mathbf{b}$, exactly as in the case of under-determined problems, and this result is derived as follows,

$$\begin{aligned}
(\mathsf{A}^T\mathsf{A})^\dagger\mathsf{A}^T &= \left((\mathsf{U}\Sigma\mathsf{V}^T)^T(\mathsf{U}\Sigma\mathsf{V}^T)\right)^\dagger(\mathsf{U}\Sigma\mathsf{V}^T)^T \\
&= \left(\mathsf{V}\Sigma^T\mathsf{U}^T\mathsf{U}\Sigma\mathsf{V}^T\right)^\dagger(\mathsf{V}\Sigma^T\mathsf{U}^T) \\
&= \left(\mathsf{V}\Sigma^T\Sigma\mathsf{V}^T\right)^\dagger(\mathsf{V}\Sigma^T\mathsf{U}^T) \\
&= \mathsf{V}(\Sigma^T\Sigma)^\dagger\mathsf{V}^T\mathsf{V}\Sigma^T\mathsf{U}^T) \\
&= \mathsf{V}(\Sigma^T\Sigma)^\dagger\Sigma^T\mathsf{U}^T \\
&= \mathsf{V}(\Sigma^\dagger\Sigma^{T\dagger})\Sigma^T\mathsf{U}^T \\
&= \mathsf{V}\Sigma^\dagger(\Sigma^{T\dagger}\Sigma^T)\mathsf{U}^T \\
&= \mathsf{V}\Sigma^\dagger\mathsf{U}^T = \mathsf{A}^\dagger.
\end{aligned}$$

This proves that the optimal solution for the case of an over-determined problem with rank-deficient $\mathsf{A}^T\mathsf{A}$, or for an under-determined problem where the rank of A is less than the number of equations, is exactly the same as for the general under-determined case.

The pseudo inverse of an invertible (full-rank) matrix is equal to its standard inverse, $\mathsf{A}^{-1} = \mathsf{A}^\dagger$, which can be shown as follows,

$$A^{-1} = (U\Sigma V^T)^{-1} = (V^T)^{-1}\Sigma^{-1}U^{-1} = V\Sigma^{-1}U^T = V\Sigma^\dagger U^T = A^\dagger.$$

The above derivation thus shows that the solution to the full-rank over-determined case (5.10) can be written as,

$$\mathbf{x} = (\mathsf{A}^T\mathsf{A})^{-1}\mathbf{b} = (\mathsf{A}^T\mathsf{A})^\dagger\mathbf{b} = \mathsf{A}^\dagger\mathbf{b}.$$

Similarly, the solution to a problem with the same number of equations and unknowns, where the matrix may be inverted may be written as,

$$\mathbf{x} = \mathsf{A}^{-1}\mathbf{b} = \mathsf{A}^\dagger\mathbf{b}.$$

Thus, we conclude that the solution to *any* system of linear equations may be written as,

$$\mathbf{x} = \mathsf{A}^\dagger\mathbf{b}.$$

See demo of the different classes of linear equations using the code
Review_examples_linear_equations.m/py.

5.2.8 PCA using SVD

PCA may be derived directly from the data matrix using SVD, instead of deriving the covariance matrix as described above. Consider an $M \times N$ data matrix representing a set of $M \times 1$ column vectors $\mathbf{f}_i$, each representing, for example, a spatial structure at a time $i = 1, \ldots, N$, such that the data matrix is $\mathsf{F} = (\mathbf{f}_1, \ldots, \mathbf{f}_N)$. The covariance matrix $\mathsf{F}\mathsf{F}^T/N$ is therefore $M \times M$. The SVD decomposition is $\mathsf{F}_{M\times N} = \mathsf{U}_{M\times M}\Sigma_{M\times N}\mathsf{V}^T_{N\times N}$. Then, the columns of U are eigenvectors $\mathsf{F}\mathsf{F}^T$ – they are normalized, so this is the same as eigenvectors of $\mathsf{C} = \mathsf{F}\mathsf{F}^T/N$ – and correspond to the principal components describing the spatial structures, and the relationships between the variables in the data vectors $\mathbf{f}_n$.

When using the covariance matrix to calculate the PCs, the expansion coefficients (time series) are obtained by projecting the data on the PC vectors, using $\mathsf{T} = \mathsf{U}^T\mathsf{F}$. Given the SVD of F, this is equal to $\mathsf{T} = \mathsf{U}^T\mathsf{U}\Sigma\mathsf{V}^T = \Sigma\mathsf{V}^T$. The columns of V are therefore the *normalized*, or *non-dimensional*, PCA expansion coefficients (time series). The *dimensional* expansion coefficients of the PCs are the first M columns of V (each of size $N\times 1$) multiplied by the corresponding singular values ($\Sigma\mathsf{V}^T$).

The advantage of PCA via SVD is smaller round-off errors because the data matrix A is not squared. The advantage of using the covariance matrix $\mathsf{F}\mathsf{F}^T/N$ is that it can be much smaller in some cases when $N > M$, as there are often more time steps than variables. Thus, in these cases using the covariance matrix leads to a much smaller problem.

Here is a summary of the approach. F is the $M\times N$ data matrix, each column is data at one time, typically $N > M$. PCA analysis is simply done using,

```
[U,S,V]=svd(F);
```

PCs are the columns of U (M vectors of size $M\times 1$). The variance explained by each mode is given by the eigenvalues of the covariance matrix, which, up to a factor N, are the same as the squares of singular values of the data matrix. Because the fraction of variance explained is normalized by the sum of the eigenvalues of the covariance matrix, the factor N does not matter, and we have for the variance explained by the ith PC,

$$\frac{\sigma_i^2}{\sum_j \sigma_j^2},$$

which is equivalent to (4.2).

5.2.9 Multivariate PCA

In many applications, it is necessary to perform the PCA of several variables that are given in different units. Examples include analyzing the relation between several stocks in NY and several stocks in Tokyo given in different currencies, or between surface temperature over many locations in Pacific ocean and wind speed over these locations, again given in different units.

We proceed as follows. First, remove the mean (and linear trend, when relevant) from each variable. That is, remove the mean from each of the NY stocks and each of the Tokyo stocks. Next, calculate the standard deviation (std, square root of the total variance) of all of the NY stocks, and normalize each of them by this std, and do the same for the Tokyo stocks. Note that we do not normalize each individual stock by its own std, but normalize each group (all NY stocks) by the std calculated from the entire group. The normalization puts the two (or more) different data sets into the same normalized units. Finally, arrange all normalized variables in a single vector and proceed to produce the data matrix from these vectors and analyze it, as in the single variable case.

In mathematical terms, consider two groups of variables, in two data matrices, X_{mn} and Y_{ln}, corresponding to $n = 1,\ldots,N$ vectors of size M and L, respectively (i.e., M stocks in NY and L in Tokyo).

Remove the mean from each row of the data matrices,

$$X'_{mn} = X_{mn} - \frac{1}{N}\sum_{i=1}^{N} X_{mi}$$
$$Y'_{ln} = Y_{ln} - \frac{1}{N}\sum_{i=1}^{N} Y_{li}$$

Normalize by std of each group of variables,

$$\hat{X}_{mn} = \frac{X'_{mn}}{\sqrt{\frac{1}{MN}\sum_{j=1}^{N}\sum_{i=1}^{M}(X'_{ij})^2}}$$
$$\hat{Y}_{ln} = \frac{Y'_{ln}}{\sqrt{\frac{1}{LN}\sum_{j=1}^{N}\sum_{k=1}^{L}(Y'_{kj})^2}}$$

Next, create an $(L+M)\times N$ data matrix F corresponding to N vectors $\mathbf{f}_n$ of length $M+L$, as $\mathbf{f}_n = (\hat{\mathbf{x}}_n;\hat{\mathbf{y}}_n)$. SVD analysis of this combined matrix, $\mathsf{F} = \mathsf{U}\Sigma\mathsf{V}^T$, gives us the multivariate PCs (columns of U) and the corresponding time series (columns of $\Sigma\mathsf{V}^T$).

Multivariate PCA is a very powerful tool for analyzing the relation between two data sets, but it may fail to extract important information as demonstrated by the following example (see SVD_applications_MCA_vs_PCA.m/py).

■ **Example 5.4** Consider the following two data sets X and Y, and the composite data set F constructed as follows,

$$\mathsf{X}_{(4\times N)} = 3\begin{bmatrix}2\\2\\1\\1\end{bmatrix}\sin(\omega_1 t) + 0.2\begin{bmatrix}2\\0\\1\\0\end{bmatrix}\sin(\omega_3 t)$$
$$\mathsf{Y}_{(3\times N)} = 2\begin{bmatrix}1\\-1\\1\end{bmatrix}\sin(\omega_2 t) + 0.2\begin{bmatrix}-1\\0\\0\end{bmatrix}\sin(\omega_3 t)$$
$$t = 0.3(1,\ldots,N),\ N = 10000$$
$$\mathsf{F} = \begin{pmatrix}\mathsf{X}\\\mathsf{Y}\end{pmatrix}_{(7\times N)} \tag{5.14}$$

where $\omega_1 = 2\pi/11$, $\omega_2 = 2\pi/7$, and $\omega_3 = 2\pi/3$. Thus, X varies predominantly with a frequency ω_1, Y with a frequency ω_2, and they both have a small shared variability with a frequency ω_3. Using SVD to analyze F, we find that the first three singular values are 671, 245 and 25, while the other four are zero,

so that only the first three PCs contribute to the variance, and are given by,

$$\mathsf{U} = \begin{bmatrix} -0.63 & 0.00 & -0.56 \\ -0.63 & 0.00 & 0.56 \\ -0.31 & 0.00 & -0.28 \\ -0.31 & 0.00 & 0.28 \\ 0.00 & 0.57 & 0.37 \\ 0.00 & -0.57 & 0.18 \\ 0.00 & 0.57 & -0.18 \end{bmatrix}.$$

The first PC clearly represents the ω_1 variability of the first data set, while the second represents the ω_2 variability of the second data set. However, the co-variability of the two data sets, represented by the common ω_3 variability, is not well-represented by the third PC. We would have expected the third PC to have the structure of the ω_3 variability, $(2,0,1,0,-1,0,0)^T$. However, $(2,0,1,0,-1,0,0)^T$ is not orthogonal to the first two PCs and therefore cannot be the third PC. In general, the requirement that PCs be orthogonal to one another, in a system where the variability modes are not necessarily orthogonal, may sometimes render the PCs less useful, as demonstrated by this particular example. The analysis suggested in the next section addresses this shortcoming of multivariate PCA in the context of analyzing the relation between two data sets. ∎

5.2.10 Maximum covariance analysis (MCA)

Multivariate PCA attempts to diagnose the variability of two or more data sets, in addition to their possible covariance, and for this reason sometimes fails to correctly diagnose the *covariance* as seen above. MCA is an alternative SVD-based analysis method that can be used to cleanly analyze the covariance of the two data sets by not attempting to also analyze the variance of each data set.

Consider again two data matrices, $\mathsf{X}_{(M\times N)}$ and $\mathsf{Y}_{(L\times N)}$, corresponding to $n = 1,\ldots,N$ vectors of size M and L, respectively (i.e., M stocks in NY and L in Tokyo, given over N days). In this case we need to remove the mean from each variable, but do not need to normalize by the standard deviation. Calculate the covariance matrix (of dimensions M by L) as

$$\mathsf{C} = \mathsf{X}\mathsf{Y}^T/N = \sum_{n=1}^{N} \mathbf{x}_n \mathbf{y}_n^T/N.$$

The "total *covariance*" is defined as the sum over all of the squared elements of C (also known as the Frobenius norm of this matrix),

$$\text{total covariance} \equiv \sum_{i=1}^{M}\sum_{j=1}^{L} c_{ij}^2.$$

To proceed, note that this total covariance is also equal to the sum over the squared singular values of C (section A.2). This suggests an SVD analysis of the covariance matrix, $\mathsf{C} = \mathsf{U}\Sigma\mathsf{V}^T$. Given this role of the singular values, we deduce that the fraction of the total *covariance* explained by SVD mode k is therefore given by,

$$\frac{\sigma_k^2}{\sum_i \sigma_i^2}.$$

Note that the columns of U have the same dimension as the columns of the X data vectors, while columns of V have the same dimension as those of the Y data matrix. If, for example, there is only one non-zero singular value, σ_1, then we deduce that the first two singular vectors, $\mathbf{u}_1$ and $\mathbf{v}_1$, completely explain the co-variability of the two data sets. Specifically, these vectors are the structures in X and Y, correspondingly, that vary together. Normally, additional singular values would be non-zero, and in this case the first mode explains the strongest co-variability, corresponding to the largest fraction of the total covariance. The next SVD vectors, $\mathbf{u}_2$ and $\mathbf{v}_2$, again represent X and Y structures that vary together, each of them is orthogonal to the previous corresponding vectors, and they explain (are parallel to) the next strongest co-variability signal. Unlike multivariate PCA, MCA does not attempt to explain the separate variability (variance) of X and Y, but only to extract the parts of the two data sets that vary together.

For the data set from the above example (5.14) (letting $N = 10,000$ to eliminate excess noise in the resulting PCs), we find,

$$\mathsf{C} = \mathsf{X}\mathsf{Y}^T/N = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{ccc} y_1 & y_2 & y_3 \\ -0.04 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ -0.02 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \end{array}$$

$$\sigma_i = 0.045, 0, 0$$

$$\mathbf{u}_1 = (-0.89, 0, -0.45, 0)^T$$

$$\mathbf{v}_1 = (1, 0, 0)^T$$

The covariance matrix shows that x_1 and x_3 are correlated with y_1, and that there are no other correlations between the two data sets. The first U and V SVD vectors nicely capture the structure of the vectors used to create the co-variability with a frequency ω_3, as desired. In this case, there is only one co-variability mode that explains all of the total *covariance* because there is only one non-zero singular value. The uncovered relationship between x_1, x_3 and y_1 is seen in the initial structure of the data governed by the combined ω_3 vector for X and Y, $(2, 0, 1, 0, -1, 0, 0)^T$.

Next, one may want to examine the contribution of the co-variability to the original two data sets. For this purpose, project the original X, Y data on the MCA vectors to find the corresponding time series,

$$\mathsf{T}_\mathsf{X} = \mathsf{U}^T\mathsf{X}$$
$$\mathsf{T}_\mathsf{Y} = \mathsf{V}^T\mathsf{Y}.$$

In the above example, this extracts only the ω_3 variability of each of the two data sets.

Finally, the co-variability of the two data sets may be a small part of the variability of each of the two, as in the above example, or may be more dominant. To quantify this, it is possible to calculate the *variance* of each of the data sets explained by the SVD vectors of the covariance matrix, and this is discussed in section 5.2.10.1. In conclusion, we use multivariate PCA to analyze the variance of multiple data sets and MCA to analyze the covariance of two data sets. Both analyses together provide a fuller picture than either method alone.

5.2.10.1 Variance explained by MCA (SVD) modes

(Optional) In MCA analysis of two co-varying data sets, we normally ask what part of the *covariance* is explained by each SVD mode, but we can also ask what part of the *variance* of each data set is

explained by the SVD modes. For example, if a certain mode explains all of the covariance between the two fields, but does not explain much of the variance, it means that the two data sets are barely related to each other.

Consider the MCA/SVD analysis of two data sets $\mathsf{X}_{M\times N}$ and $\mathsf{Y}_{L\times N}$, and for convenience assume that different columns correspond to different times. Write these as time-dependent (zero-mean, detrended) vectors $\mathbf{x}_n$ and $\mathbf{y}_n$ where n is a discrete index running over all times in the data. Let the SVD modes of the covariance matrix $\mathsf{C} = \mathsf{X}\mathsf{Y}^T/N$, corresponding to X, be $\mathbf{e}_m$, $m = 1,\ldots,M$. We can expand $\mathbf{x}_n$ in these modes as,

$$\mathbf{x}_n = \sum_{m=1}^{M} a_{mn}\mathbf{e}_m,$$

where the expansion coefficients are obtained from this first equation using,

$$\mathbf{e}_j^T\mathbf{x}_n = \sum_{m=1}^{M} a_{mn}\mathbf{e}_j^T\mathbf{e}_m = a_{jn}.$$

The norm square of $\mathbf{x}_n$ at a time n is,

$$\mathbf{x}_n^T\mathbf{x}_n = \left(\sum_{m=1}^{M} a_{mn}\mathbf{e}_m^T\right)\left(\sum_{j=1}^{M} a_{jn}\mathbf{e}_j\right).$$

Given that the SVD modes are orthonormal, $\mathbf{e}_m^T\mathbf{e}_j = \delta_{mj}$,

$$\mathbf{x}_n^T\mathbf{x}_n = \sum_{m=1}^{M} a_{mn}^2.$$

The total variance at all times is therefore

$$var(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n^T\mathbf{x}_n = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} a_{mn}^2.$$

So the total variance explained by, say, SVD mode k is,

$$\frac{1}{N}\sum_{n=1}^{N} a_{kn}^2,$$

and the percent total variance of X explained by mode k is, therefore,

$$100\times\frac{\sum_{n=1}^{N} a_{kn}^2}{\sum_{n=1}^{N}\sum_{m=1}^{M} a_{nm}^2}.$$

In Matlab, this would be calculated as follows,

```
C=XY'/N;
% e_j are the columns of U:
[U,S,V]=SVD(C);
% calculate expansion coefficients of X data in terms of its SVD modes:
a=U'*X;
% calculate total variance
tmp=a.*a/N;
total_variance=sum(tmp(:))
% calculate variance of X explained by modes k=1:3
for k=1:3;
  mode_k_variance(k)=a(k,:)*a(k,:)'/N;
end
% desired fraction:
var_k_percents=100*mode_k_variance(:)/total_variance;
% sum of the variance of X explained by these first few modes:
sum(var_k_percents)
```

5.2.11 The Netflix Prize

Consumers reveal their preferences – whether for their music taste, movie preference, favorite TV shows, or products they like to buy on-line – via their ratings of products, and companies therefore have large amounts of rating data that can be used to more effectively recommend new content to their customers. To invite improved algorithm ideas, Netflix hosted a competition for a $1,000,000 prize. The winning entry was based on SVD, among other factors Koren et al. (2009).

Two possible strategies for recommendation systems are based on *content* and on *collaborative filtering*. In the context of music recommendation, for example, content-based filtering is based on a list of properties acquired from the information provided by the users about their preferences, as well as on properties of the music or artists they listened to, such as genre. Collaborative filtering calls upon the user's past behavior – for example, previous transactions or movie ratings – without explicitly using user-profile information. Collaborative filtering suffers the "cold start" problem of difficulty in making appropriate recommendation to new users.

Collaborative filtering can work by comparing either items (e.g. movies) or users. Comparing items: suppose movies x, y, z are similar in the sense that they were liked by the same users. Then if a given user liked movies x, y she is also likely to like z. Comparing users: suppose users a, b, c are similar in the sense that they liked the same movies. Then, if users a, b liked a given movie, we can guess that user c will also like this movie.

The ratings of movies by users can be put into a large sparse matrix whose elements r_{um} contain the rating by user u of movie m. It turns out to be useful to use SVD to process and analyze this matrix in order to obtain optimal recommendations. The following is a simplified version that ignores the dimension-reduction element of the algorithm.

1. Define the element r_{ij} of the rating matrix R as the rating by user i of movie j. There are m users and n movies so m rows and n columns in the user-movie matrix, R.
2. Fill missing values: calculate average rating for each movie over users, and fill missing values with the appropriate average for that movie.
3. Remove from each row (that is, from each user) the mean over movies, $\bar{r}_i = \frac{1}{n}\sum_{j=1}^{n} R_{ij}$.
4. Perform SVD, decide on a rank k, and calculate a reduced rating matrix to remove noise, replacing

R with $\mathsf{U}_{m\times k}\Sigma_{k\times k}\mathsf{V}^T_{k\times n}$.

5. The columns of U,V tell us something useful about the user groups and movie groups.
6. Calculate similarity of movies j,f based on the reduced rating matrix (the similarity is the correlation of movies j,f based on average over users),

$$sim_{jf} = \frac{\sum_{i=1}^{m} R_{ij}R_{if}}{\sqrt{(\sum_{i=1}^{m} R_{ij}^2)(\sum_{i=1}^{m} R_{if}^2)}}$$

7. Calculate a prediction of the rating by user i of movie j, by averaging over the ratings of all movies by the same user, each weighted according to its similarity to movie j,

$$pr_{ij} = \frac{\sum_{f=1}^{n} sim_{jf}R_{if}}{\sum_{f=1}^{n} |sim_{jf}|} + \bar{r}_i$$

Example code is given by, `SVD_applications_Netflix.m/py`.

(Optional) Information on the fuller procedure in highlighted parts of section 6.1 of Vozalis and Margaritis (2006) is available here. Note that instead of eqn (4) in Vozalis, the predicted rating of movie a by user j is $pr_{aj} = \sum_{i=1}^{n} sim_{ji}(rr_{ai} + \bar{r}_a)/(\sum_{i=1}^{n} |sim_{ji}|)$, where sim_{ji} is the similarity between the ratings of movies i,j by all users, and the sum is over movies.

6. Similar items and frequent patterns

6.1 Similar items

6.1.1 Motivation

In analyzing large data sets, we may be interested in the similarity between different items. Are two faces similar? Does the similarity between two texts suggests plagiarism? Are two shoppers buying similar items? Are two consumers watching similar movies?

We start by considering a few measures of similarity, and then discuss "hash functions" which are used to convert text to numerical data for more efficient similarity analysis. Then, we look at the matrix representation of sets, which is useful in comparing documents, shoppers, and so on.

6.1.2 Jaccard similarity

The Jaccard similarity of two sets A and B is defined as the ratio of the size of the intersection of A and B to the size of their union,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}, \tag{6.1}$$

where $|A|$ is the number of elements in a set A. Consider a few examples:

1. **Sets of logicals:**

   ■ **Example 6.1** Calculate the Jaccard similarity, $J(a,b)$, of the following sets:

   ```
   a=[0 0 1 1 0 1 0 1 1]
   b=[0 1 0 1 0 0 1 0 1]
   ```

   There are $n = 9$ entries in each of the sets, but the similarity is defined only based on the columns of the sets that have at least one 1 entry. We thus exclude columns 1 and 5 from our calculation

of the similarity, and have a union of 7. We then calculate the number of columns in the data with a 1 in both set a and b - there are two such columns. Thus, we conclude that $\boxed{J(a,b) = 2/7}$. ■

2. **Sets of integers:**

   ■ **Example 6.2** Calculate the Jaccard similarity, $J(p,q)$, of the following sets:

   ```
   p=[1,2,3,4,5,6]
   q=[3,5,8]
   ```

   To find the union, $p \cup q$, we count the number of distinct entries: there are six unique entries in set p, and one additional unique entry in set q (3 and 5 are repeated from the first set), for a total of seven unique entries. The intersections of the set are the numbers 3 and 5 — so two overlaps. Thus, $\boxed{J(p,q) = 2/7}$. ■

3. **Sets of words:**

   ■ **Example 6.3** Calculate the Jaccard similarity $J(x,y)$, of the following two sets of words:

   ```
   x={Chase after money and security and your heart will never
   unclench Care about peoples approval}
   y={and you will be their prisoners Do your work then step
   back The only path to serenity};
   ```

   To calculate the Jaccard similarity of these strings we calculate the number of distinct words in these two strings: we find that this union is 28 words. The words that appear in both strings, x and y, are "and", "your" and "will", or a total of three words. Thus, $\boxed{J(x,y) = 3/28}$. ■

For further examples and for programming tips, please see the demo `Jaccard_examples.m/py`. See also Leskovec et al. (2014)§3.1.1.

6.1.3 Shingling of documents

6.1.3.1 k-Shingling

To conduct similarity analysis on a large set of possibly large documents, two steps are needed. First, it turns out that individual words from a document do not convey the document's contents as well as small sets of adjacent words from the document, known as "shingles". Second, it is more efficient to turn these shingles into numbers using hash functions and analyze those numbers, than to analyze the words themselves.

Consider first the shingling of a string of characters. The "k-shingling" involves taking k continuous characters at a time. For example, the set of 3-shingles of the string `x={abcdeffedcba}`, are `{abc,bcd,cde,def,eff,ffe,fed,edc,dcb,cba}`. We can apply this concept of k-shingling on a much larger scale for different k's and compare the emerging sets from two documents to determine their similarity. The concept of k-shingles is especially useful when forcing shingles of words rather than characters: in most cases of plagiarism, for example, one rarely finds a verbatim copy of text, but rather similar meaning, rearrangement of word order, and portions of shared text that can be revealed by comparing appropriate word-shingles.

The general rule of thumb is to pick a k such that the probability of any given shingle appearing in any given document is low but not too low. For example, if we pick a k that is too small, like $k = 1$, we would find that the sets emerging from a document would be common among many other documents which may in fact be dissimilar. If we choose k to be too large, then no two strings/documents will be similar.

6.1.3.2 Shingling using stop words

Often, a useful way to construct shingles of words from longer text, is to make sure they always start with what are known as "stop words". Examples of stop words, though they can be designed according to specific context, are "and", "but", "you", and so on. Stop words are common words which are not specific or revealing of context. Experience shows they are helpful in delimiting sets of words into chunks that can be compared for analyzing the similarity of documents.

■ **Example 6.4** As an example, here is how we could calculate the first eight 3-shingles starting with stop words for the following text:

```
Chase after money and security and your heart will never unclench Care about
peoples approval and you will be their prisoners Do your work then step back
                        The only path to serenity
```

The first eight "3-shingles" are then,

```
{"and security and","and your heart", "your heart will","will never unclench",
   "about peoples approval","and you will","you will be", "will be their"}
```

and so on. ■

6.1.4 Hash functions

The purpose of a hash function is to produce a number that represents a given string or k-shingle of words, such that two different strings are **not likely** to result in the same number. A simple yet oversimplified (and thus poor) example in Matlab code is,

```
% specify input string:
input_string='adr1'
% create an array of numbers with an element corresponding to each
% character in the string:
array=double(input_string)
% this gives:
% array =    97   100   114    49
% calculate the product of all numbers:
n=prod(array);
% yielding :
% n=54184200
% Calculate the hash as the reminder of division by a prime number,
% say 17, so that the hash value is always between 0 and 16:
hash_value=mod(n,17);
% which gives the final hash value of hash_value=15.
```

The function maps any string to a number between 0 and 16, so now instead of comparing an infinite set of strings, we need to compare numbers in that range, a much more manageable task.

However, this hash function fails as it returns the same number for two different strings. For example, the hash values of both 'asdf1' and 'asdf2' are both zero, so the comparison of these two hash values will give us the wrong impression that these are equal strings. The problem here is the letter 'f': Because `double(f)=102` is divided by 17 with no remainder, the multiplication by 102 in both strings

results in a zero hash value. More sophisticated hash functions can be constructed such that they fail this way very rarely, so that if the hash values are similar, we can assume with a high level of certainty that the original strings or shingles are indeed identical.

See the Matlab demos of the above oversimplified hash function `simple_hash_function.m/py` as well as a more sophisticated "Cyclic Redundancy Check" hash function known as crc32, `crc32_demo.m/py`.

6.1.5 Matrix representation of sets

It is often helpful in calculating Jaccard similarities between shopping baskets of customers, or documents represented by shingles of words, to represent these sets (of documents, shopping baskets, and so on) using matrices. We construct a matrix of logicals (1 or 0) such that the row corresponds to a particular item in the store (or k-shingle in a document), and each column corresponds to a given set (one basket or one document). Consider, for example, the sets in Table 6.1. In this example, each set contains items from the larger set $U = \{a,b,c,d,e,f\}$. Entry (1,1) is 1 because item a is in Set 1, and entry (2,1) is 0 because item b is not in Set 1. We can see that $S_1 = \{a,e\}$ and $S_4 = \{b,c,e,f\}$.

item	S1	S2	S3	S4
a	1	1	0	0
b	0	1	1	1
c	0	1	1	1
d	0	1	0	0
e	1	1	0	1
f	0	0	0	1

Table 6.1: Matrix representation of 4 sets

Companies often want to be able to recommend an item to a potential customer: which book to buy, or which movie to watch, based on an individual's previous choices. One way to do this is to recommend items that have been bought/watched by other people with a similar taste. This is referred to as *collaborative filtering* – as discussed in the Netflix challenge example, section 5.2.11. For example, in the context of online shopping, similar tastes would be expressed by a significant Jaccard similarity. The definition of "significant" similarity needs to be determined empirically for a given problem such that it is small enough to indicate that tastes are indeed similar, but not too small such that no two tastes are similar. This empirically determined value may be very small (a few to less than one %), but in practice this may still suggest useful recommendations.

6.2 Frequent patterns and association rules

We now move onto a discussion of frequent patterns and association. We are interested in evaluating, in large datasets, the frequency with which we find items together. For example, in the context of a grocery store, what items are *frequently* purchased together (are often together in a customer's "basket" – e.g., the obvious combination of hot dogs and mustard, or, more surprisingly, beer and diapers!). With this information, stores can organize sale items to encourage bundled purchasing. In the context of the internet, words frequently searched together are used to give recommendations to user searches. For example, for a user who searches "Harvard" may also be interested in "Ivy League". Similarly with book recommendations on online book stores, etc. There are applications of frequent patterns in medical diagnosis as well, in for example, biomarkers in diseased plants. Each "basket" here contains

the biomarkers seen in an affected plant, and the fact that they appear together helps us understand the role of previously unidentified biomarkers.

We next examine the efficient "A-priori" algorithm for finding frequent patterns in a large data sets, and see how such patterns help us identify "association rules" that teach us about interesting connections in the data. A helpful reference for this section is Leskovec et al. (2014).

6.2.1 Mining frequent patterns

In order to aid the discovery of frequent patterns and association rules between items we use a *market-basket* model of data, where each *basket* has a subset of *items*, known as an *itemset*. A set of items that appears in many different baskets is a "frequent" itemset. Specifically a set is frequent if it appears in a fraction of the baskets larger than a specified *support threshold*, s. The "support", k, of an itemset, I, is the number of baskets which contain I.

■ **Example 6.5** As an example, consider a few Google searches from what appears to be a pet-lover living in Cambridge.

```
S1: {dog, cat, good}
S2: {rain, in, Cambridge}
S3: {rain, dog, good}
S4: {Cambridge, bad, weather, rain}
S5: {Pets, Harvard, Cambridge}
S6: {Harvard, dog, policy, good}
S7: {Harvard, Square, Cambridge}
```

Setting the *support threshold* to $s = 3/7$, we look through the above baskets to find the frequent "singletons" – i.e., single words which occur in three or more baskets. We find that,

```
dog, rain, good, Harvard, Cambridge
```

are the frequent singletons that appear in 3 or more sets. Now, using these singletons we form all the candidate frequent pairs (of which there are $\binom{5}{2}$ - order does not matter),

```
{dog, rain}
{dog, good}
{dog, Harvard}
{dog, Cambridge}
{rain, good}
{rain, Harvard}
{rain, Cambridge}
{good, Harvard}
{good, Cambridge}
{Harvard, Cambridge}
```

Looking at the sets which contain one or more of these frequent pair candidates,

```
S1: {dog, cat, good}
S2: {rain, in, Cambridge}
S3: {rain, dog, good}
S4: {Cambridge, bad, weather, rain}
```

```
S5: {Pets, Harvard, Cambridge}
S6: {Harvard, dog, policy, good}
S7: {Harvard, Square, Cambridge}
```

we see that the number of occurrences of these candidates are,

```
{dog, rain}=1
{dog, good}=3
{dog, Harvard}=1
{dog, Cambridge}=0
{rain, good}=1
{rain, Harvard}=0
{rain, Cambridge}=2
{good, Harvard}=1
{good, Cambridge}=0
{Harvard, Cambridge}=2
```

and thus the only frequent pair is `{dog, good}` because it occurs in at least a fraction s (3/7) of the itemsets. Thus, the Googler seems to associate "good" and "dog". This is an example of a two iteration A-priori algorithm, covered in more detail next. ■

6.2.2 A-priori algorithm and association rules

The A-priori algorithm allows us to efficiently identify frequent patterns in a large set of baskets. First consider the following definitions. A "frequent set" I is one that appears in a larger fraction of the baskets than the specified threshold, s. An "association rule" $I \rightarrow j$ tells us that if a given set I appears in a basket, a given item j is likely to be in that basket as well. The "confidence" of a rule is the fraction of baskets with I that also contain j, and the "interest" in a rule is the difference between confidence in $I \rightarrow j$ and fraction of baskets that contain j. These are all demonstrated next.

The A-priori algorithm proceeds as follows:

1. Select a support threshold, s, as a fraction of all sets.
2. For every item (word, book, etc.), count the number of sets the item occurs in.
3. Identify the frequent singletons – the items which are in at least a fraction s of the sets.
4. List the candidates for frequent pairs – all combinations of frequent singletons. For n frequent singletons there should be $\binom{n}{2}$ candidates for frequent pairs.
5. List the sets which contain one or more frequent pair candidate.
6. Identify the frequent pairs – those with a fraction equal to or greater than s.
7. Identify the candidate triple frequent sets: the triple sets that are composed of pairs of items that themselves are frequent pairs.
8. Identify the frequent triple sets by searching the data.
9. Repeat to find frequent 4-sets or higher, if desired.

■ **Example 6.6** As an example, consider the following sets,

```
S1: 3,4,5,6,7,8,9,10,13,
S2: 1,2,3,4,5,7,11,13,
S3: 1,2,3,5,6,12,
S4: 2,3,4,7,9,11,12,13,15,
S5: 1,3,4,7,9,12,13,14,
```

```
S6: 2,4,5,6,8,14,
S7: 1,2,3,6,7,8,9,10,
S8: 1,2,3,4,6,8,9,10,13,
S9: 1,2,4,5,6,9,11,13,15,
S10: 2,4,7,8,9,10,13,14,
```

Assume a threshold $s = 0.55$, and calculate the fraction occurrence of each single items across the sets,

Item	% Occurence of Item
1	0.6
2	0.8
3	0.7
4	0.8
5	0.5
6	0.6
7	0.6
8	0.6
9	0.7
10	0.4
11	0.3
12	0.3
13	0.7
14	0.3
15	0.2

Thus, candidate pairs are pairs that involve items (1,2,3,4,6,7,9,13). There are $\binom{8}{2} = 28$ candidates. Of these, we find that there are only 4 frequent pairs, composed of the frequent items (2,4,9,13),

Item 1	Item 2	% Occurrence of Pair
2	4	0.6
4	9	0.6
4	13	0.7
9	13	0.6

The possible items in a frequent triple are therefore, (2,4,9,13), yielding 4 possible triplets ($\binom{4}{3} = 4$ candidates). For the triples,

$$(4,9,13)$$
$$(2,9,13)$$
$$(2,4,13)$$
$$(2,4,9)$$

to be considered as candidate frequent triplets, all possible sub-pairs of a triplet need to be frequent. So, of these four possible triplets, we are left only with (4,9,13) as a candidate frequent triple set. Indeed, counting the number of occurrences, we find that this triple is frequent,

Item 1	Item 2	Item 3	% Occurrence of Triplet
4	9	13	0.6

The possible association rules that can be deduced from the frequent triplet are,

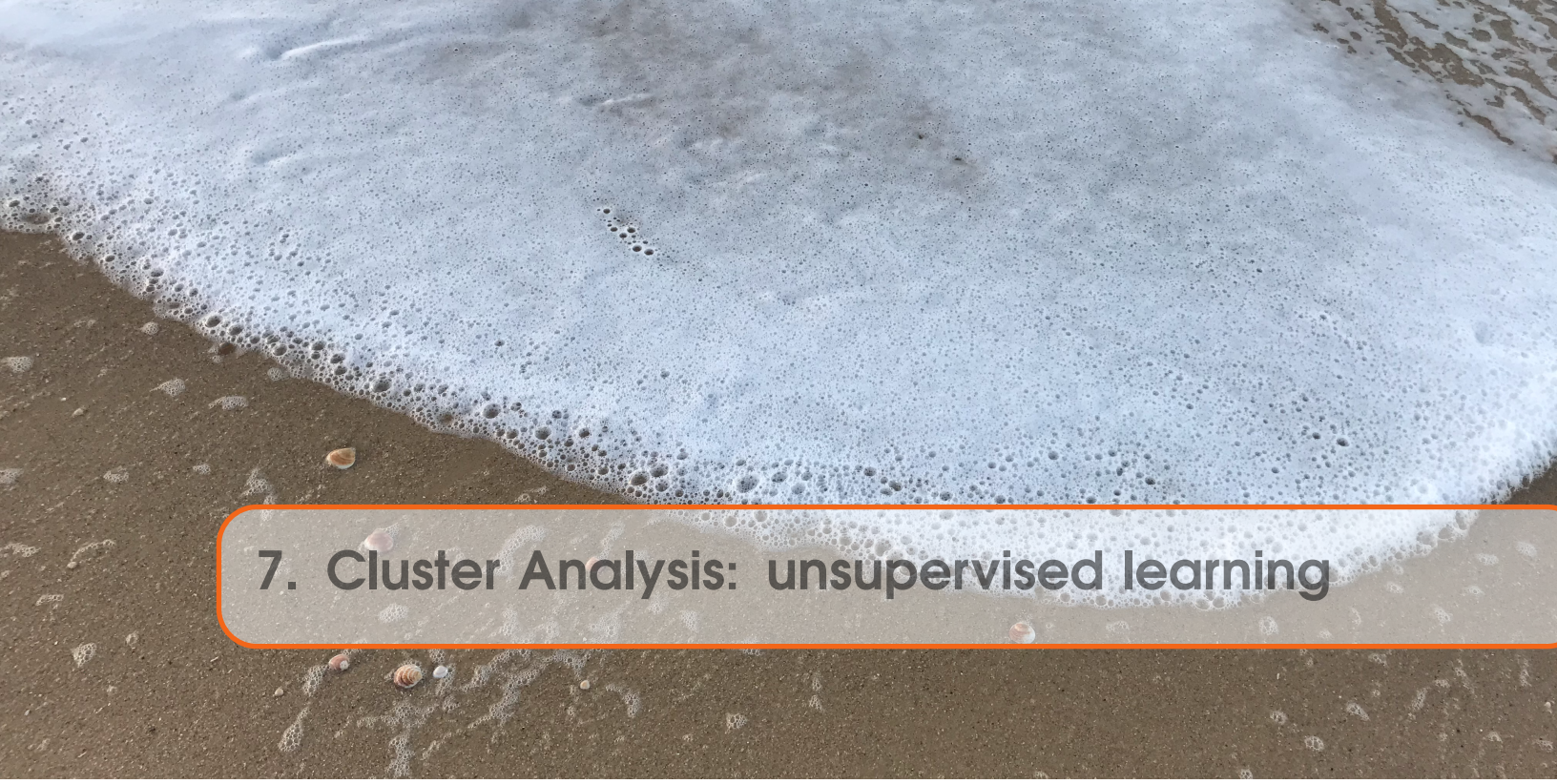
Association Rule
$(4, 9) \to 13$
$(4, 13) \to 9$
$(9, 13) \to 4$

The confidence (fraction of baskets with I that also contain j) and interest (difference between confidence in $I \to j$ and fraction of baskets that contain j) are,

Association Rule	I Support	(I, j) Support	j Fraction	Confidence	Interest
$(4, 9) \to 13$	6	6	0.7	1	0.3
$(4, 13) \to 9$	7	6	0.7	$\frac{6}{7}$	0.157
$(9, 13) \to 4$	6	6	0.8	1	0.2

Note also that the interest in a rule can be negative, and rules with negative interest can be... interesting! For example, if out of one hundred customers, 10 buy Coke and 10 buy Pepsi, with no overlap, then the *confidence* in the Association Rule Coke $\to$ Pepsi is 0, yet the *interest* is $0 - \frac{1}{10} < 0$, clearly telling us that the items may in fact be "substitute goods". ∎

Part Three



7. Cluster Analysis: unsupervised learning

7.1 Motivation

In this chapter we explore a variety of "unsupervised" learning techniques for clustering large data sets. We begin by discussing the various ways of measuring "distance" between data points – the key tool in clustering that allows us to cluster together similar items, whose distance from each other is small. Quite often, data may not be in Euclidean space, and thus other distance measures may be required. We then survey two main approaches to clustering: hierarchical (each point is an initial cluster, then clusters are merged to form larger ones) and point-assignment (starting with points that are cluster representatives, clusteroids, and then adding other points one by one). We will also consider methods that are appropriate for very large problem when computer memory becomes an issue. This process of cluster discovery and analysis has broad ranging applications, from archaeology to weather and climate.

Some example interesting applications are,

- *Archaeology/Anthropology*: Clustering allows us to group pottery samples from multiple sites according to color, shape, material and discovery location, to hopefully reveal their original cultural origin.
- *Genetics*: Gene expression data can be clustered together to identify genes of similar function; grouping known genes with novel ones may reveal the function of the novel genes.
- *TV Marketing*: Grouping TV shows into groups likely to be watched by people with similar purchasing habits;
- *Criminology*: Clustering Italian provinces shows that crime is not necessarily linked to geographic location (North vs. South Italy), as is sometimes assumed.
- *Medical Imaging*: Identifying volumes of cerebrospinal fluid, white matter, and gray matter from magnetic resonance images (MRI) of the brain using clustering methods.
- *Social networks*: Cluster individuals based on behavior patterns to identify communities.

- *Internet Search Results*: Show relevant related results to a given search beyond using keywords and link analysis.
- *Weather and climate*: Identify consistently re-occurring weather regimes to increase predictability.

7.2 Distances/metrics

Before diving into some different measures of distance, there are three conditions that must be satisfied by a distance function, $d(x,y)$:

1. $d(x,y) \geq 0$ – No negative distance.
2. $d(x,y) = d(y,x)$ – Distance from x to y is the same as from y to x, i.e., distance is a symmetric function with respect to its two arguments.
3. $d(x,y) \leq d(x,z) + d(z,y)$ – This is known as the "triangle inequality". It ensures that the distance between two points is the shortest path from one to the other.

7.2.1 Euclidean distances

The typical measure of distance in an n-dimensional space is known as the L_2-norm, and is what we typically think of as "distance". In an n-dimensional Euclidean space (where points are defined by vectors of n numbers),

$$d([x_1,x_2,...,x_n],[y_1,y_2,...,y_n]) = \sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

■ **Example 7.1** Consider this 6-dimensional example,

$$\begin{aligned}\mathbf{x} &= \begin{bmatrix}2 & 4 & 5 & 3 & 8 & 6\end{bmatrix}^T, \quad \mathbf{y} = \begin{bmatrix}1 & 3 & 10 & 2 & 5 & 7\end{bmatrix}^T \\ d(\mathbf{x},\mathbf{y}) &= \sqrt{(2-1)^2+(4-3)^2+(5-10)^2+(3-2)^2+(8-5)^2+(6-7)^2} \\ &= \sqrt{38}.\end{aligned}$$

■

Another notable Euclidean distance is the L_1-norm, also known as the *Manhattan Distance*, which defines the distance between two points as the sum of the magnitudes of the differences in each dimension. In two dimensions, the Manhattan distance is the travel distance between two points in a city with a grid street system, which is where the distance measure derives its name from.

More generally, the L_r-norm is defined as,

$$d([x_1,x_2,...,x_n],[y_1,y_2,...,y_n]) = \left(\sum_{i=1}^{n}|x_i-y_i|^r\right)^{1/r}$$

Finally, another interesting measure is the L_∞-norm, which is the limit as $r \to \infty$ of the L_r norm. As r grows, only the dimension with the largest difference matters, so formally, the L_∞-norm is the maximum of $|x_i - y_i|$ over all dimensions i. For example, consider the 3d Euclidean space and the points (1,2,3) and (4,8,6). The L_∞-norm is,

$$\max(|1-4|,|2-8|,|3-6|) = \max(3,6,3) = 6.$$

7.2.2 Hamming distance

The Hamming distance between two ordered sets of elements, $d_H(\mathbf{x},\mathbf{y})$, is the number of components in which $\mathbf{x}$ and $\mathbf{y}$ differ.

■ **Example 7.2** For example, for the vectors,

$$\mathbf{x} = \begin{bmatrix} 4 & 6 & 8 & 6 \end{bmatrix}^T, \quad \mathbf{y} = \begin{bmatrix} 4 & 7 & 5 & 6 \end{bmatrix}^T,$$

the Hamming Distance is $d_H(\mathbf{x},\mathbf{y}) = 2$, because these vectors differ in two components, the second and third. ■

7.2.3 Cosine distance

The cosine distance, $d_{\cos}$, considers points of a similar angle from the origin as "close". For example, a scalar multiple of a vector has the same angle from the origin as the original vector, and thus these two vectors have a cosine distance of 0. The cosine distance is calculated as,

$$d_{\cos}(\mathbf{x},\mathbf{y}) = \cos^{-1}\left(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{(\sum_{i=1}^{n} x_i^2)(\sum_{i=1}^{n} y_i^2)}}\right)$$

7.2.4 Jaccard distance

In the previous chapter, we discussed the Jaccard similarity $J(\mathbf{x},\mathbf{y}) \in [0,1]$ as a measure of the likeness of items in two sets. The Jaccard distance, $d_J(\mathbf{x},\mathbf{y})$, is simply defined as $1 - J(\mathbf{x},\mathbf{y})$. This is because if two sets are similar, they have a high Jaccard similarity, but in the context of clustering should have a *small* distance.

7.2.5 Edit distance

Finally, the Edit distance, $d_E(\mathbf{x},\mathbf{y})$, between two strings, $\mathbf{x}$ and $\mathbf{y}$, is defined as the smallest number of insertions or deletions of single characters required to convert $\mathbf{x}$ to $\mathbf{y}$ (or vice versa). Consider, for example, the strings $\mathbf{x} =$ `fcvbnn` and $\mathbf{y} =$ `fvgnm`. The edit distance $d_E(\mathbf{x},\mathbf{y}) = 5$, because 5 operations are required to convert $\mathbf{x}$ to $\mathbf{y}$, which are,

1. Delete `c`
2. Delete `b`
3. Insert `g` after `v`
4. Delete the second `n`
5. Insert `m` after `n`

7.3 The curse of dimensionality

Euclidean distance measures in a high-dimensional space can often introduce non-intuitive issues which are referred to as the "curse of dimensionality", whose two manifestations are: (1) distances between almost all pairs of points are nearly equal, and (2) almost all vector pairs are orthogonal to each other. This renders Euclidean and cosine distance measures meaningless, and thus can make clustering of points in high dimensional spaces a challenge.

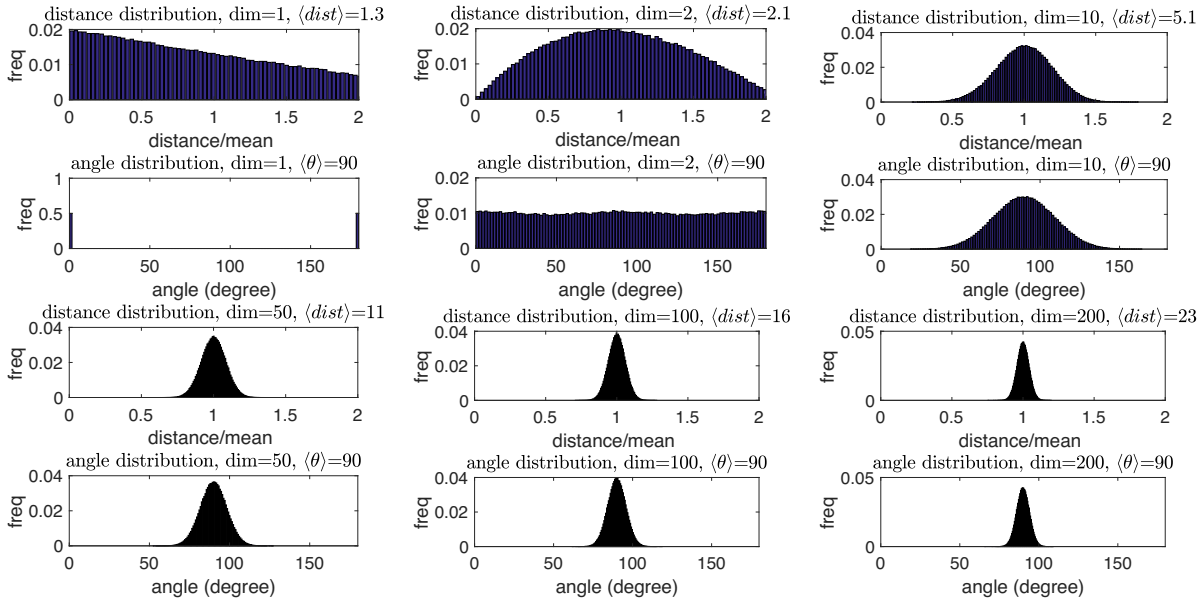


Figure 7.1: The curse of dimensionality, showing the distribution of distances and angles between pairs of random points as function of data space dimension.

7.3.1 Euclidean distances in high-dimensional spaces

Consider *random points* in the unit cube in a d-dimensional Euclidean space. Each point is written as $\mathbf{a} = [a_1, a_2, ..., a_d]'$, with $a_i \in [0,1]$. The distance squared between two points $\mathbf{a} = [a_1, a_2, ..., a_d]$ and $\mathbf{b} = [b_1, b_2, ..., b_d]$ is,

$$\sum_{i=1}^{d} (a_i - b_i)^2.$$

Given that a_i and b_i are random numbers in the range of 0 to 1, then for large d, the squared distance between the two is proportional to the average of random squared differences $(a_i - b_i)^2$. As long as the dimension is large, the distance squared is always close to this average, and thus we conclude that the distance between any two random vectors is always nearly the same. This is problematic, for example, if we try to cluster together nearby points, since the points will essentially be inseparable using Euclidean distance measures. Be wary that this analysis assumes the data points are random, which is not the case in applications, but it still points to a surprising property of high-dimensional spaces.

7.3.2 Angles between random vectors in high-dimensional spaces

A similarly unexpected result occurs with regards to the angles between random vectors in high-dimensional space. Consider two points, $\mathbf{x}$ and $\mathbf{y}$. The cosine of angle between these two vectors is,

$$\cos(\angle x, y) = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \sqrt{\sum_{i=1}^{d} y_i^2}}.$$

This is also the formula for the correlation of two sets of numbers. Given that x_i and y_i are assumed random, and thus uncorrelated, this correlation is expected to be very small for a large value of d. Thus, in a high dimensional space, the cosine of the angle between vectors is zero, and so the angle between vectors is 90^o. Thus, random vectors in high dimensions tend to be orthogonal! This implies that the cosine distance is not expected to be a useful measure of distance for the purpose of clustering random data either, because the angles between most data points will be near 90°.

Fig. 7.1 shows the tendency for Euclidean distances to be uniform and for angles between random points to be 90°, respectively.

7.4 Hierarchical clustering

Hierarchical clustering algorithms start with each point as its own cluster. Pairs of clusters are then merged based on their "proximity" (which can be defined in several ways). Merging stops once further clustering would produce clusters that are less useful. The optimal number of clusters and the stopping criteria are discussed below.

■ **Example 7.3** Consider a simple example using Euclidean distance between *clusteroids* (cluster centers, defined as the average location of all points in the cluster) as the measure for deciding which clusters to merge. Fig. 7.2 shows the five points. The clustering proceeds as follows.

1. Initially each point is its own cluster, and the clusteroids are just the data points themselves

Cluster:	1	2	3	4	5
x	1.2068	1.2462	0.7871	−0.8844	−1.0588
y	0.4	1.0714	1.0268	0.9298	1.0137

2. Calculate the Euclidean distances between all points, constructing a 5×5 symmetric distances matrix, D, where d_{ij} is the distance between points i and j.

$$\mathsf{D} = \begin{pmatrix} 0 & 0.6726 & 0.7543 & 2.157 & 2.347 \\ 0.6726 & 0 & 0.4613 & 2.135 & 2.306 \\ 0.7543 & 0.4613 & 0 & 1.674 & 1.846 \\ 2.157 & 2.135 & 1.674 & 0 & 0.1935 \\ 2.347 & 2.306 & 1.846 & 0.1935 & 0 \end{pmatrix}$$

3. The smallest distance is between points 4 and 5, so we merge them and calculate their centroid to be `[-0.9716,0.9718]`. The number of clusters is now $k = 4$, given by {1}, {2}, {3}, {4,5}. The new centroids of the 4 clusters are now,

Cluster:	1	2	3	(4 and 5)
x	1.207	1.246	0.7871	−0.9716
y	0.4	1.071	1.027	0.9718

4. Next, calculate the distances between the clusters again (a 4×4 distance matrix, not shown) and merge the clusters with the smallest distance between them (2 and 3). The number of clusters is now $k = 3$, given by {1}, {2,3} and {4,5}.
5. Calculate the clusteroids of the remaining 3 clusters, and the revised 3×3 distance matrix, revealing that we should merge 1 with {2,3}. The number of clusters is now $k = 2$, given by {1,2,3} and {4,5}.

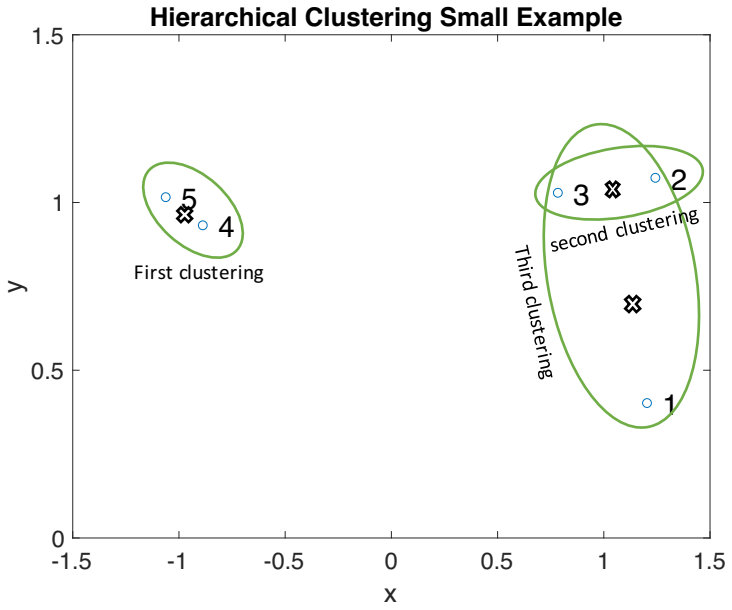


Figure 7.2: First three steps in a simple 5 point hierarchical clustering example.

6. Merge the remaining 2 clusters into a single cluster, if desired.

■

We will see below how the optimal number of clusters can be determined from this analysis.

7.4.1 Efficiency of hierarchical clustering

This hierarchical clustering algorithm is quite inefficient. At each step we calculate the distance between all pairs of clusters to determine the best clusters to merge. The first distance calculation takes n^2 steps, and subsequent steps require $(n-1)^2, (n-2)^2, \ldots$. This sums up to is $O(n^3)$. As noted in Leskovec et al. (2014), there are more efficient versions of hierarchical clustering that have efficiency $O(n^2 \log(n))$.

7.4.2 Merging and stopping criteria

In order to decide which clusters to merge and when to stop merging, we need to define some measures of cluster quality. The clusteroid, or cluster center, is the average of the locations of all N points in the cluster,

$$\bar{\mathbf{x}} = \mathbf{x}_c = \frac{1}{N}\sum_i \mathbf{x}_i.$$

The cluster radius, r, is the largest distance of a point in the cluster to the clusteroid, $r = \max_i(|\mathbf{x}_i - \mathbf{x}_c|)$. The cluster diameter is the largest distance between any two members of the cluster, $d = \max_{i,j}(|\mathbf{x}_i - \mathbf{x}_j|)$. The cluster density is the number of points in the cluster, divided by the radius, $\rho = N/r$, or divided by the diameter, or by the radius raised to the power of the data dimension, and so on. The variance of cluster A is,

$$\text{var}_A = \sum_{j=1}^{N_A}\sum_{i=1}^{N_d}(x_{ij} - \bar{x}_i)^2$$

where N_d is the dimension of the data vectors $\mathbf{x}_j$, N_A is the number of points in cluster A, x_{ij} is the ith element of data vector $\mathbf{x}_j$, and $\bar{x}_i$ is the location of the ith coordinate of the clusteroid. A good cluster should be characterized by a small radius, diameter or variance and a large density.

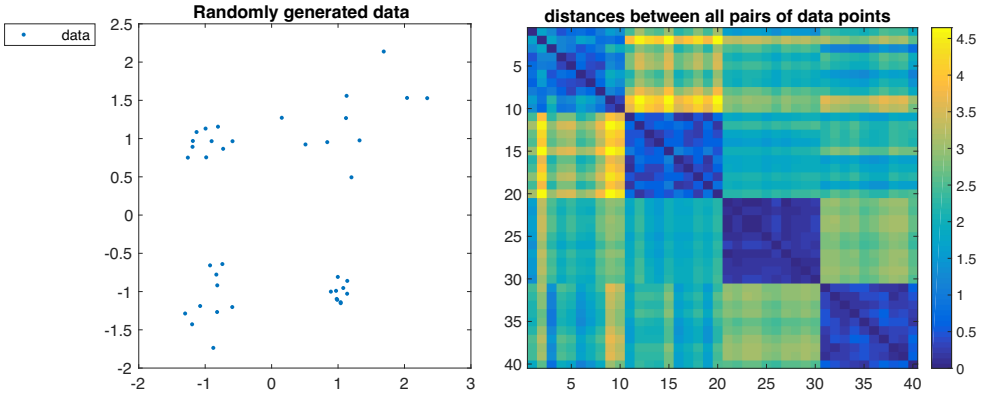


Figure 7.3: 40 random data points and their distance matrix, represeting their respective distances from one another.

To demonstrate how the optimal number of clusters is chosen, consider a hierarchical clustering example of 40 points in a 2d space (Figure 7.3). Our merging criteria uses the popular Ward method, described in section 7.4.4 below, which merges clusters such that the merge leads to the smallest increase in variance.

The entries along the diagonal of the distance matrix are zero, and the points seem to form natural clusters: the points indexed 1–10 appear to have small distance from one anther, as do 11–20, 21–30 and 31–40. The clusters at different values of the choice for the clusters number k are shown in Fig. 7.6.

In determining the optimal number of clusters and the order of the clustering process we consider a "dendrogram plot" (Fig. 7.4) and an "elbow plot" (Fig. 7.5). The dendrogram shows the progress of the clustering, with the horizontal axis representing all data points, and the vertical axis representing the merging criterion or a related measure. The vertical axis could show, for example, the value of the variance in the Ward method at each merging, or the smallest cluster diameter at merging, or the average cluster diameter, or the distance between clusteroids at merging, and so on. The figure shows a large increase in the vertical measure when going from four clusters to two clusters, indicating that four may be a good choice. Note also the somewhat large distance when going from five clusters to four, suggesting that perhaps five clusters is also a reasonable choice.

To identify the optimal number of clusters we can also look for a "break point" in the elbow plot (Fig. 7.5). This figure shows a measure of the cluster quality (say maximum diameter in this case), as function of the number of clusters. Going from small number of clusters k to a larger one, the maximum diameter goes down. When increasing the number of clusters does not lead to a reduction in the maximum diameter (or other merging criterion being used), we conclude that the optimal number has been reached. Again, in this case the elbow plot suggests four or five clusters is optimal. The final choice has to be made based on what makes sense for the application being considered.

There are several possible methods (referred to as "linkages") for deciding which clusters to merge at each step. We can choose to merge two clusters that,

- have the smallest distance between clusteroids.
- lead to the smallest increase in the variance of the merged cluster (Ward method).

- result in the smallest radius or diameter of the resulting cluster.
- result in the highest density of the resulting cluster.
- are characterized by the smallest "single" distance: this distance is the smallest distance between any two points in the two clusters. This approach is especially helpful in analyzing complexly-shaped clusters (section 7.10).

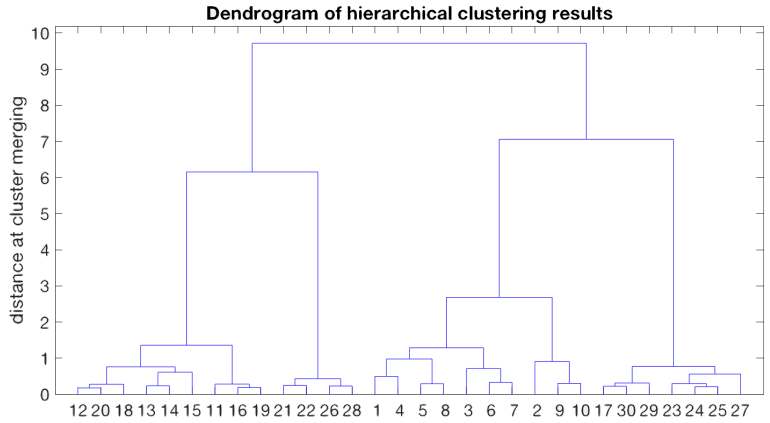


Figure 7.4: A "dendrogram" plot, showing the progression of cluster merging in hierarchical clustering.

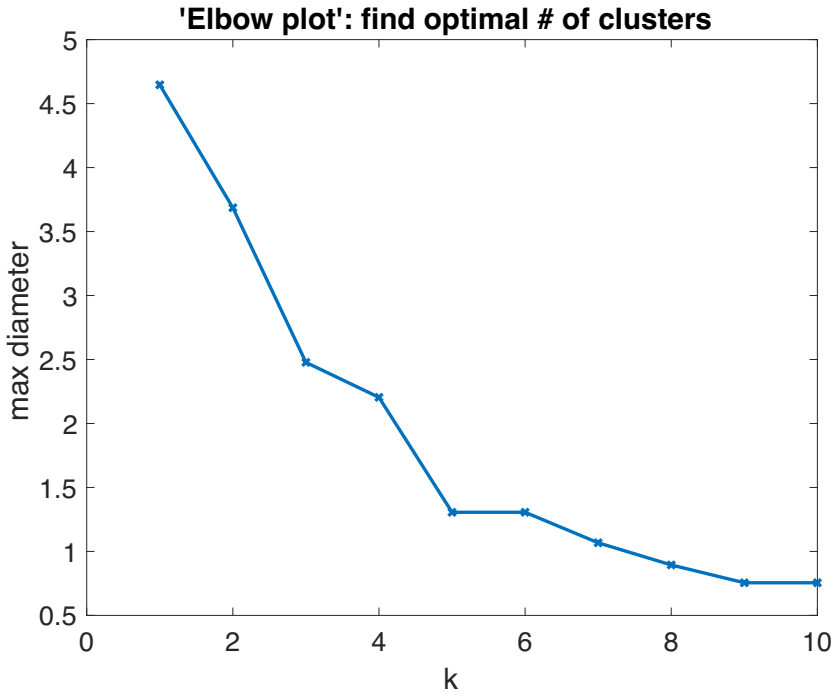


Figure 7.5: Elbow plot revealing that $k = 5$, or perhaps 4, is a sensible optimal number of clusters.

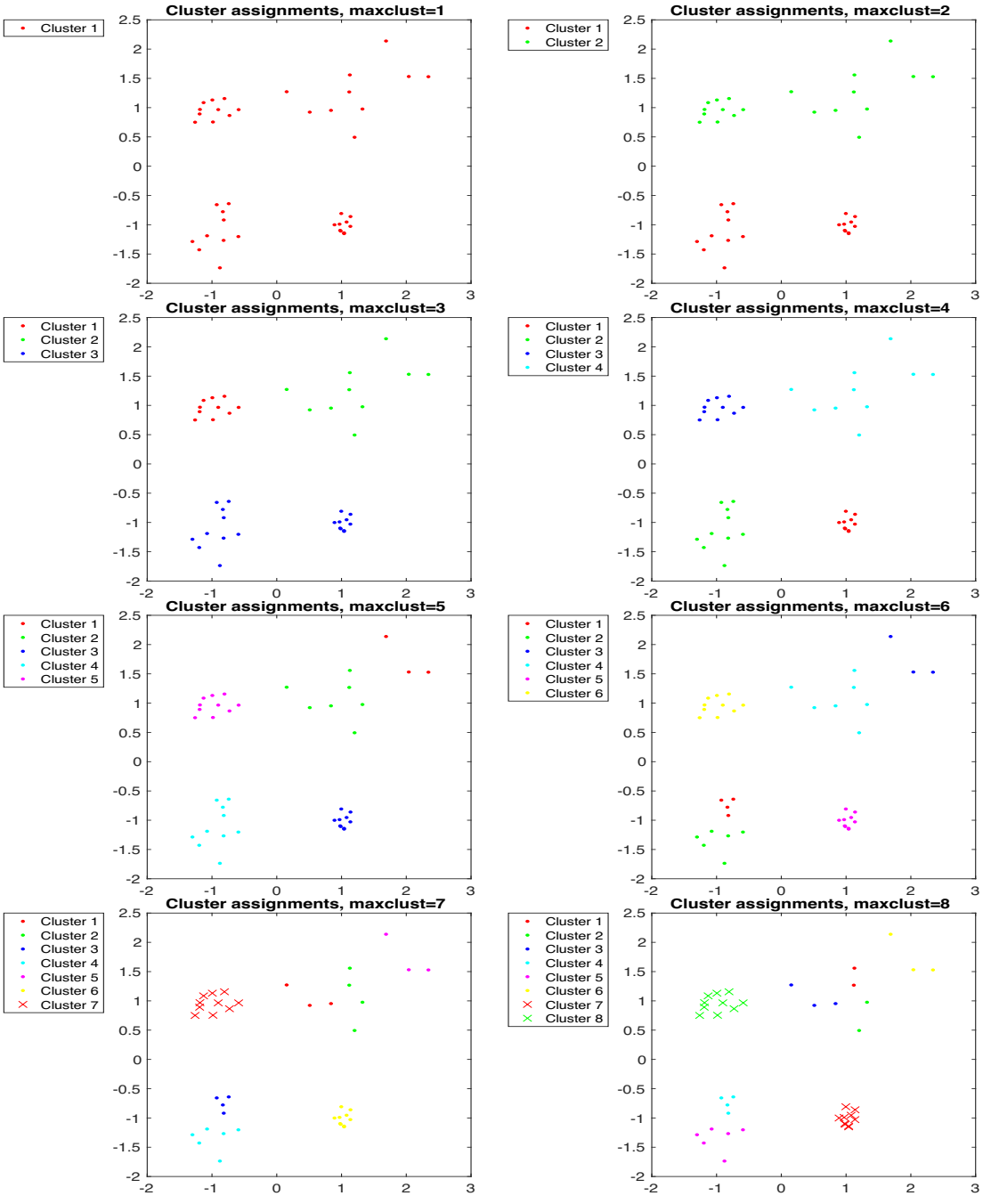


Figure 7.6: Resulting clustering of 40 points using the Ward method for merging, depending on desired number of clusters, k.

7.4.3 Hierarchical clustering in non-Euclidean spaces

Non-Euclidean spaces do not support the concept of a centroid, and thus we need to use some other measure of distance as discussed in section 7.2: Jaccard, cosine, edit or Hamming distances. We can still define the diameter, density, etc. of a cluster, however. Instead of using a clusteroid, we can find a data point that is "representative" of the cluster. An optimal representative generally minimizes one or more of three measures,

1. The sum of the square of the distances to the other points in the cluster
2. The sum of the distances to the other points in the cluster
3. The maximum distance to another point in the cluster

■ **Example 7.4** For example, consider the clustering of the strings `fghj`, `ftgh`, `tghj` and `gjhk`. We calculate the edit distances between these strings to be,

	`fghj`	`ftgh`	`tghj`	`gjhk`
`fghj`	0	2	2	4
`ftgh`	2	0	2	4
`tghj`	2	2	0	4
`gjhk`	4	4	4	0

Applying the three criteria for a good representative we can see that `fghj`, `ftgh`, and `tghj` are all equally good representatives, but `gjhk` is not. ■

7.4.4 Ward method

In the Ward method, a common merging strategy in hierarchical clustering, clusters are merged when the resulting change in the variance within a newly merged cluster is minimal. Here we develop the formula for calculating this change in variance. Consider the clustering of N-dimensional data vectors, where X_{ijA} is the i component of the jth data vector, which is a part of cluster A. The clusteroid and total variance for cluster A are,

$$\bar{X}_{iA} = \frac{1}{N_A}\sum_{j=1}^{N_A} X_{ijA}$$

$$var_A = \sum_{j=1}^{N_A}\sum_{i=1}^{N}(X_{ijA} - \bar{X}_{iA})^2$$

with a similar formula for cluster B, where N_A and N_B are the number of points in clusters A and B being merged into a cluster C and defining $N_C = N_A + N_B$. Using

$$\bar{X}_{iC} = \frac{1}{N_A + N_B}(N_A\bar{X}_{iA} + N_B\bar{X}_{iB}),$$

the total variance of the combined cluster is given by,

$$\begin{aligned} var_C &= \sum_{i=1}^{N}\sum_{j=1}^{N_C}(X_{ijC}-\bar{X}_{iC})^2 \\ &= \sum_{i=1}^{N}\left[\sum_{j=1}^{N_A}(X_{ijA}-\bar{X}_{iC})^2+\sum_{j=1}^{N_B}(X_{ijB}-\bar{X}_{iC})^2\right] \\ &= \sum_{i=1}^{N}\left[\sum_{j=1}^{N_A}\left((X_{ijA}-\bar{X}_{iA})-\frac{1}{N_C}(N_B\bar{X}_{iB}-N_B\bar{X}_{iA})\right)^2\right. \\ &\qquad \left.+\sum_{j=1}^{N_B}\left((X_{ijB}-\bar{X}_{iB})-\frac{1}{N_C}(N_A\bar{X}_{iA}-N_A\bar{X}_{iB})\right)^2\right]. \end{aligned}$$

Rearranging,

$$\begin{aligned} &= \sum_{i=1}^{N}\left[\sum_{j=1}^{N_A}(X_{ijA}-\bar{X}_{iA})^2+\sum_{j=1}^{N_B}(X_{ijB}-\bar{X}_{iB})^2\right. \\ &-\sum_{j=1}^{N_A}2(X_{ijA}-\bar{X}_{iA})\frac{1}{N_C}(N_B\bar{X}_{iB}+N_B\bar{X}_{iA})+\sum_{j=1}^{N_A}\frac{1}{N_C^2}(N_B\bar{X}_{iB}-N_B\bar{X}_{iA})^2 \\ &\left.-\sum_{j=1}^{N_B}2(X_{ijB}-\bar{X}_{iB})\frac{1}{N_C}(N_A\bar{X}_{iA}+N_A\bar{X}_{iB})+\sum_{j=1}^{N_B}\frac{1}{N_C^2}(N_A\bar{X}_{iA}-N_A\bar{X}_{iB})^2\right]. \end{aligned}$$

Using $\sum_{j=1}^{N_A}(X_{ijA}-\bar{X}_{iA})=0$, we have,

$$\begin{aligned} &= var_A+var_B+\sum_{i=1}^{N}\frac{N_AN_B^2}{N_C^2}(\bar{X}_{iB}-\bar{X}_{iA})^2+\frac{N_BN_A^2}{N_C^2}(\bar{X}_{iA}-\bar{X}_{iB})^2 \\ &= var_A+var_B+\sum_{i=1}^{N}\frac{N_AN_B(N_B+N_A)}{(N_A+N_B)^2}(\bar{X}_{iB}-\bar{X}_{iA})^2 \\ &= var_A+var_B+\sum_{i=1}^{N}\frac{N_AN_B}{N_A+N_B}(\bar{X}_{iB}-\bar{X}_{iA})^2. \end{aligned}$$

The change in variance is therefore given by,

$$\Delta var = var_C-(var_A+var_B)=\frac{N_AN_B}{N_A+N_B}\sum_{i=1}^{N}(\bar{X}_{iB}-\bar{X}_{iA})^2,$$

or, in vector form, the final expression for the change in variance due to the merging of clusters A and B is,

$$\Delta var = var_C-(var_A+var_B)=\frac{N_AN_B}{N_A+N_B}\|\bar{X}_B-\bar{X}_A\|^2.$$

Note that in some software packages, including Matlab (see near bottom of the following link), the above expression is changed in two ways. First, using the square root of Δvar and second, including a factor of 2 as follows,

$$\sqrt{\Delta var}=\sqrt{var_C-(var_A+var_B)}=\sqrt{\frac{2N_AN_B}{N_A+N_B}}\|\bar{X}_B-\bar{X}_A\|.$$

With this factor of 2, the distance between two singleton clusters is the same as the Euclidean distance.

7.5 K-means

The k-means algorithm is an example of a *point-assignment* clustering method. We assume for simplicity a Euclidean space and specify the number of clusters, k, in advance. The algorithm is as follows for clustering N data points,

1. Choose k data points that are as far as possible from one another, such that are likely to be in different clusters. These are the initial centroids:
   (a) Choose the first point randomly.
   (b) Choose the $m+1$ representative point such that its distance from the previous m representatives is as large as possible. Specifically, go over all data points and for each point calculate the smallest distance to one of the m previously selected representatives. Then choose representative point $m+1$ such that it maximizes the smallest distance to the previous m points.
2. For each of the remaining $N-k$ data points:
   (a) Find the centroid closest to a given point
   (b) Add this point to the cluster of that centroid
   (c) Adjust the centroid of that cluster to include the added point
3. Optional "iterations": Use the final centroids as initial representative points and go through steps 2(a),(b),(c) again with all data points; repeat as many of these iterations as needed to converge.
4. Optional additional "replicates": Repeat all above steps with different choices for the randomly selected initial point and then choose the replicate that leads to the best clusters.

Given the k-means results for different values of k, it is then necessary to choose the optimal number of clusters using an elbow plot or a similar criterion.

This *point-assignment* clustering method is different from the hierarchical clustering method discussed earlier because we specify k to begin with and choose a set of initial clusteroids or representative points. It is not necessary to calculate all distances between all pairs of the N points at every step – we only need to calculate the distance from each data point to the k clusteroids. This is much more efficient, as typically $k \ll N$.

Demos: `run_kmeans_clustering_demos.m/py`.

7.6 Self-organizing maps

The Self-Organizing Maps (SOM) clustering method, also known as Kohonen Maps, calculates a specified number of clusteroids using a machine learning approach: going through the data points one at a time and improving the estimate for the k clusteroids gradually. We start by defining "representative points" that will eventually become the clusteroids in two separate spaces,

1. The data space, where the representative points are marked by the vectors $\mathbf{m}_i$, of the same dimension as the data points.
2. A grid space, in which the representatives are arranged in an equi-distant line or 2d grid. The locations of the representatives in this space are denoted $\mathbf{r}_i$. The grid space also defines the "neighbors" of each representative.

First, we choose the number of clusters to be calculated and set the initial position of the appropriate number of clusteroids in data space. At each iteration (t), the algorithm proceeds as follows:

1. Select one data point, $\mathbf{z}(t)$
2. Find the representative nearest to $\mathbf{z}(t)$, mark its location as $\mathbf{m}_c(t)$
3. Move this nearest representative slightly toward the data point

4. Identify the neighboring representative points in grid space of the nearest representative (numbers 2 and 4 in Fig. 7.7). Move these representatives toward $\mathbf{z}(t)$ as well, but less, as specified below.

This is repeated for all points in the data, where a full pass over all points is referred to as an *epoch*, and the process is repeated for several epochs until convergence.

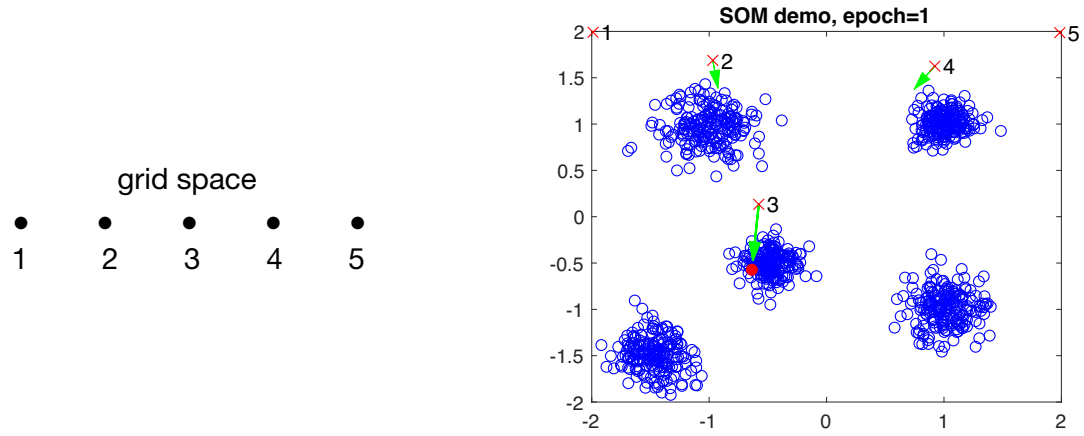


Figure 7.7: (left) An example 1d grid space. (right) An example SOM iteration. Representative point number 3 is the nearest to the data point being considered (red circle) thus it is pushed toward the nearest data point. Representative number 3's nearest neighbors in grid space are representative points 2 and 4, which are also pushed toward the data point, although to a lesser degree.

Consider at this point the first demo self_organizing_maps_hand_calculation_example.m/py which uses a 1d arrangement of the three representative points. This demo highlights two problems that need to be resolved with appropriate refinements to the above algorithm. First, the clusteroids $\mathbf{m}_i(t)$ continue moving until the end of the calculation, never converging. Second, the clusteroids do not end up at the center of their respective cluster. The reason for these two problems is that each clusteroid is "attracted" to the points in other clusters, even if it is not the closest representative to those points. Consider, therefore, the following modifications to the algorithm:

1. Make the "learning rate", $\alpha(t)$, at which representatives move toward the data points, decrease as the iterations proceed. This leads to a convergence of $\mathbf{m}_i(t)$ and prevents the representative points from moving forever.
2. At later stages of the learning, where each representative point has found its cluster, ensure that its location is not affected by data points to which it is not the closest representative point. This way each representative point is eventually influenced by its own cluster only (that is, affected only by points to which it is the closest representative), so that it finds itself at the center of that cluster.

In addition, as a useful generalization, one may allow a 2d arrangement in grid space of the representative points.

The movement of a representative point $\mathbf{m}_i(t)$ in data space, when representative $\mathbf{m}_c(t)$ is the closest to data point $\mathbf{z}(t)$, is given by,

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + h_{ci}(t)[\mathbf{z}(t) - \mathbf{m}_i(t)] \tag{7.1}$$

$$h_{ci}(t) = \alpha(t) e^{-\|r_i - r_c\|^2/\sigma(t)}. \tag{7.2}$$

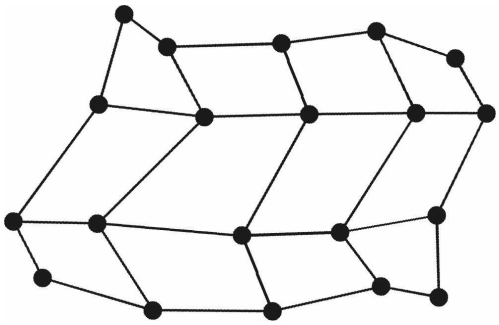


Figure 7.8: A Sammon mapping of grid space from Johnson et al. (2008): the black circles describe the positions of the SOM patterns.

Both $\alpha(t)$ and $\sigma(t)$ should be functions that decrease with iteration number (e.g., $1/t$). Note that the nearest representative, for which $i = c$ moves toward the data point at a rate $\alpha(t)$ as the Gaussian is equal to one when $i = c$. Other points move less, at a rate also determined by the Gaussian term. As the neighborhood size in grid space represented by $\sigma(t)$ becomes smaller (that is, the "neighborhood kernel" becomes more restricted in grid space), points other than $i = c$ are affected less and less and now each representative is influenced only by its own cluster, allowing it to move to the center of that cluster. A simpler example of a neighborhood kernel that is used in some of our class demos sets the kernel to one for nearest neighbors and zero otherwise, instead of letting $h_{ci}(t)$ decay away from the representative being considered.

Finally, consider the "Sammon map" which summarizes the results by showing which clusters are similar: the grid space is re-plotted as points in 2d such that the distance between neighboring representatives is proportional to the (say, Euclidean) distance between the corresponding clusteroids. See Fig. 2 of Johnson et al. (2008), reproduced in Fig. 7.8.

7.7 Mahalanobis distance

In clustering applications, we must be careful to choose a distance measure that correctly takes into account the structure of data. Consider, for example, the 2d data in Fig. 7.9. There is an elliptical cluster of points, with the clusteroid marked by a cross and some point A highlighted in red, and another point B marked in blue. Point B appears to lie outside of the elliptical cluster, even though point A is further from the clusteroid than point B, based on a Euclidean distance. This reveals that a Euclidean distance is inappropriate here as it does not take into account the structure of the data and would have led to the inclusion of point B in the cluster, even though we can see it does not belong there. We therefore would like to derive an alternative distance measure that takes into account the structure of the data.

Let $X_{2\times N} = \{\mathbf{x}_n\}$, $n = 1 \ldots N$ represent N random 2d column vectors, Gaussian distributed with standard deviation of one and zero mean, shown in Fig. 7.10a. Its Euclidean distance from the origin,

$$d_E(\mathbf{x}, \mathbf{0})^2 = (\mathbf{x} - \mathbf{0})^T (\mathbf{x} - \mathbf{0}) = \mathbf{x}^T \mathbf{x}$$

is shown in Fig. 7.10a.

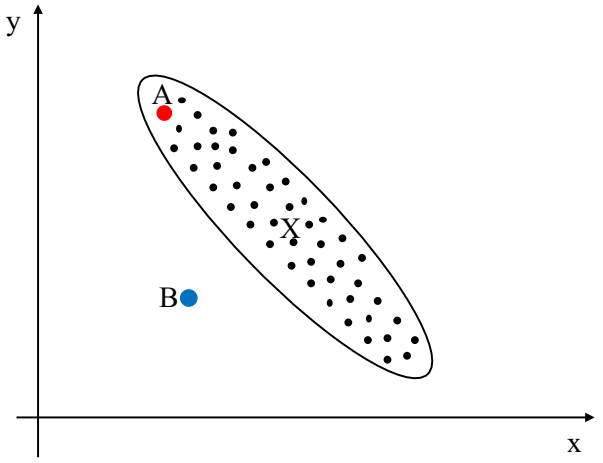


Figure 7.9: Data showing an elliptical cluster, where point A is intuitively expected to be in the cluster while point B is outside. However, point B has a smaller Euclidean distance to the clusteroid than A, demonstrating the shortcoming of Euclidean distance measures in clustering unusually shaped data.

Next, consider a set of data points that is not symmetric around zero. For this, consider the transformation matrix

$$\mathsf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix}$$

which stretches the data in the vertical direction by a factor of 5 and leaves the horizontal direction unchanged, and let $\mathbf{y}_n = A\mathbf{x}_n$, with a corresponding data matrix $Y_{2\times N} = \{\mathbf{y}_n\}$, $n = 1 \ldots N$. The stretched data and their Euclidean distances from the origin are shown in Fig. 7.10c. This distance distribution is clearly not satisfactory from the point of view of cluster analysis, as there is a clear cluster, but one would expect its edge to be characterized by a constant distance from the origin. This is the objective of the Mahalanobis distance.

To calculate the Mahalanobis distance, define the covariance matrix $\mathsf{C} = \mathsf{Y}\mathsf{Y}^T/N$, and the inverse covariance C^{-1} which turns out numerically to be,

$$\mathsf{C} \approx \begin{pmatrix} 1 & 0 \\ 0 & 25 \end{pmatrix}.$$

Let us calculate the covariance matrix analytically as well,

$$\begin{aligned} \mathsf{C} &= \sum_n \mathbf{y}_n \mathbf{y}_n^T / N = \sum_n (\mathsf{A}\mathbf{x}_n)(\mathsf{A}\mathbf{x}_n)^T / N \\ &= \sum_n \mathsf{A}\mathbf{x}_n \mathbf{x}_n^T \mathsf{A}^T / N = \mathsf{A}\left(\sum_n \mathbf{x}_n \mathbf{x}_n^T / N\right)\mathsf{A}^T \\ &= \mathsf{A}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\mathsf{A}^T = \mathsf{A}\mathsf{A}^T = \begin{pmatrix} 1 & 0 \\ 0 & 25 \end{pmatrix}. \end{aligned}$$

We have used the fact that the non-transformed data vectors are random (uncorrelated) so that their covariance matrix is the unit matrix, $\langle x_i(n) x_j(n) \rangle = \delta_{ij}$, where we denoted $\mathbf{x}_n = (x_1(n), x_2(n))^T$. The covariance matrix of the transformed data is diagonal and corresponds to the stretching in the vertical direction, as discussed above.

The inverse of the covariance matrix is,

$$\mathsf{C}^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 0.04 \end{pmatrix}.$$

The Mahalanobis distance from the origin, defined as,

$$\begin{aligned} d_M(\mathbf{y}, \mathbf{0}) &\equiv \sqrt{\mathbf{y}^T \mathsf{C}^{-1} \mathbf{y}} \\ &= \sqrt{y_1^2 + \frac{1}{25} y_2^2} \end{aligned}$$

is shown in Fig. 7.10d. Note how the distance in the second dimension is reduced by the same factor in which it was stretched to create the data. This distance has the desired properties because C^{-1} undoes the data stretching, and the Mahalanobis distance between the transformed data points is equal to the regular distance between the non-transformed data.

$$\begin{aligned} d_M(\mathbf{y}, \mathbf{0})^2 &= \mathbf{y}^T \mathsf{C}^{-1} \mathbf{y} = (\mathsf{A}\mathbf{x})^T \mathsf{C}^{-1} (\mathsf{A}\mathbf{x}) = \mathbf{x}^T \mathsf{A}^T \mathsf{C}^{-1} \mathsf{A}\mathbf{x} \\ &= \mathbf{x}^T \mathsf{A}^T (\mathsf{A}\mathsf{A}^T)^{-1} \mathsf{A}\mathbf{x} = \mathbf{x}^T \mathsf{A}^T ((\mathsf{A}^T)^{-1} \mathsf{A}^{-1}) \mathsf{A}\mathbf{x} = \mathbf{x}^T \mathbf{x}. \end{aligned}$$

Given that the covariance matrix is diagonal, let its diagonal elements be σ_i^2. Its inverse is then also diagonal, composed of $1/\sigma_i^2$, and the distance may therefore be simply written as,

$$d_M(\mathbf{y}, \mathbf{x}) = \sqrt{\sum_i \frac{(y_i - x_i)^2}{\sigma_i^2}}.$$

This shows that, in this example, the Mahalanobis distance simply undoes the stretching by normalizing the data about its standard deviation in each direction.

Consider next both a stretching and a rotation of the data $\mathbf{y} = (\mathsf{B}\mathsf{A})\mathbf{x}$, where,

$$\begin{aligned} \mathsf{A} &= \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix} \\ \mathsf{B} &= \begin{pmatrix} 0.7071 & 0.7071 \\ -0.7071 & 0.7071 \end{pmatrix} \\ \mathsf{R}(\theta) &= \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}, \end{aligned}$$

where $R(\theta)$ represents the general form of a 2d rotation matrix by θ degrees, showing B represents an anti-clockwise rotation of 45°. The Euclidean distance from the origin is shown by the color shading in Fig. 7.10e. The numerically calculated covariance and inverse covariance are then,

$$\begin{aligned} \mathsf{C} &= \begin{pmatrix} 0.7071 & -3.5355 \\ 0.7071 & 3.5355 \end{pmatrix} \\ \mathsf{C}^{-1} &= \begin{pmatrix} 0.7071 & 0.7071 \\ -0.1414 & 0.1414 \end{pmatrix} \end{aligned}$$

and the corresponding Mahalanobis distance is shown in Fig. 7.10f. As a check, calculate the covariance matrix analytically again,

$$\begin{aligned}\mathsf{C} &= \sum_n \mathbf{y}_n \mathbf{y}_n^T / N = \sum_n (BA\mathbf{x}_n)(BA\mathbf{x}_n)^T / N \\ &= \sum_n (\mathsf{BA})\mathbf{x}_n \mathbf{x}_n^T (\mathsf{BA})^T / N = (BA) \sum_n \mathbf{x}_n \mathbf{x}_n^T / N (BA)^T \\ &= (\mathsf{BA}) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} (\mathsf{BA})^T = \mathsf{BAA}^T\mathsf{B}.\end{aligned}$$

The Mahalanobis distance based on this C is shown in Fig. 7.10f and again has the desired property of equal distance to cluster edge.

In general, the Mahalanobis distance is helpful because it takes advantage of existing structure (covariance) in the data, and thus allows us to cluster data with unique shape.

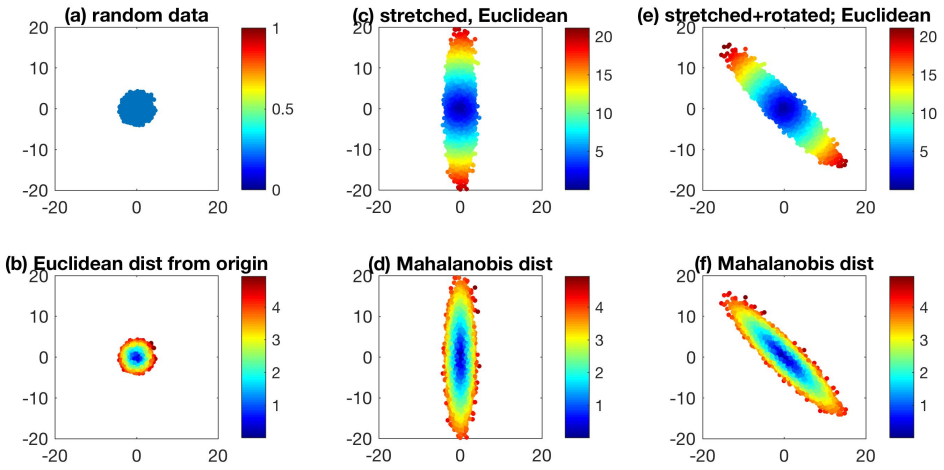


Figure 7.10: Euclidean vs. Mahalanobis distances

7.8 Spectral clustering

Earlier in the course we explored how to partition networks into two distinct subgroups using the second eigenvector of the Laplacian matrix of a network. We now take this a step further and explore how to cluster networks or data points into two or more clusters using a similar spectral clustering algorithm. The spectral clustering algorithm is simple, efficient, and in some cases may even outperform much simpler algorithms such as the k-means (Von-Luxburg, 2007). Before diving into spectral clustering, you may find it useful to revisit our discussion of partitioning of networks in Chapter 3.

7.8.1 Similarity, degree and Laplacian matrices

When discussing network clustering, we constructed an adjacency matrix A such that $a_{ij} = 1$ if nodes i, j are connected, and 0 otherwise. Instead, we now construct a similarity matrix W for data points based on their distances from one another. Given a set of data points, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ the objective is to cluster together nearby points. Given the matrix with distances between each two nodes, $\text{dist}_{ij} = |\mathbf{x}_i - \mathbf{x}_j|$, we

construct a *similarity* matrix W, for example by using $w_{ij} = \exp(-\text{dist}_{ij}^2/\sigma^2)$. Thus, instead of binary entries we have a range of values from 0 to 1. The diagonal degree matrix, D, is then defined as before,

$$d_{ii} = \sum_{j=1}^{n} w_{ij}.$$

The Laplacian matrix is also defined as before using $\mathsf{L} = \mathsf{D} - \mathsf{W}$.

Theorem 7.8.1 — Laplacian Matrix. The Laplacian matrix, L, satisfies the following properties:

1. For every vector $\mathbf{x}$ of a dimension n equal to the number of data points,

$$\mathbf{x}^T \mathsf{L}\mathbf{x} = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}(\mathbf{x}_i - \mathbf{x}_j)^2.$$

   Proof: Using the definition of d_{ii},

$$\mathbf{x}^T \mathsf{L}\mathbf{x} = \mathbf{x}^T \mathsf{D}\mathbf{x} - \mathbf{x}^T \mathsf{W}\mathbf{x} = \sum_{i=1}^{n} d_i \mathbf{x_i}^2 - \sum_{i,j=1}^{n} \mathbf{x_i}\mathbf{x_j} w_{ij}$$
$$= \frac{1}{2}\Big(\sum_{i=1}^{n} d_i \mathbf{x_i}^2 - 2\sum_{i,j=1}^{n} \mathbf{x_i}\mathbf{x_j} w_{ij} + \sum_{j=1}^{n} d_j \mathbf{x_j}^2\Big) = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}(\mathbf{x_i} - \mathbf{x_j})^2$$

2. L is symmetric and positive semi-definite.
   Proof: The symmetry of L is a direct result of the symmetry of D and A, and it is positive semi-definite because, as shown above, $\mathbf{x}^T \mathsf{L}\mathbf{x} \geq 0$.

3. The smallest eigenvalue of L is zero and the corresponding eigenvector is a vector of ones, $\mathbf{1}$.

4. Because it is symmetric and non-negative, L has n non-negative, real-valued eigenvalues,

$$0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$$

The eigenvector of L corresponding to λ_2 can be used to cluster the data into two parts, as explained and proved in the context of network clustering in Chapter 3.

7.8.2 Spectral clustering algorithm

Consider the clustering of n data vectors of dimension m into k clusters. The spectral clustering algorithm proceeds as follows,

1. Calculate the L_2 distances dist_{ij} between each pair of the m-dimensional n data vectors to create the distance matrix.
2. Calculate the $n \times n$ "similarity" matrix, W, such that close points are assigned a value close to 1 and far points are represented by a value close to zero. E.g., $w_{ij} = \exp(-\text{dist}_{ij}^2/\text{dist}_{mean}^2)$, where dist_{mean} is the mean of all non-zero values in the distances matrix, dist.
3. Calculate the diagonal degree matrix, D.
4. Calculate the $n \times n$ Laplacian matrix, $\mathsf{L} = \mathsf{D} - \mathsf{W}$.
5. Calculate the n-dimensional eigenvectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{k-1}$, corresponding to the $k-1$ smallest eigenvalues.

6. In order to cluster into k clusters, place the calculated eigenvectors $\mathbf{u}_2,\ldots,\mathbf{u}_{k-1}$ into a new $(k-1)\times n$ data matrix $\mathsf{F}=(\mathbf{u}_2,\ldots,\mathbf{u}_{k-1})$.
7. Cluster the data matrix F into k clusters, treating it as a set of n data vectors of dimension $(k-1)$. For 2 clusters use the signs of $\mathbf{u}_2$ to cluster the data. For 3 or more clusters, use k-means or similar methods. The resulting k clusters of the n vectors in F is the desired clustering of the original n vectors.

Note that because the number of required clusters is normally much smaller than the dimension of the original data vectors, the dimension of the new data matrix, $(k-1)\times n$, is much smaller than the dimension of the original data set, $m\times n$, and thus its clustering is easily obtained. To calculate the smallest $k-1$ eigenvectors of L, we can use the efficient inverse block power method. The use of the eigenvectors of the Laplacian matrix corresponding to the $k-1$ smallest non-zero eigenvalues is justified as was the use of the second eigenvectors in the case of network clustering: these vectors minimize $\mathbf{x}^T\mathsf{L}\mathbf{x}$ and thus minimize the distance between pairs of points in the same cluster.

7.8.3 Example

■ **Example 7.5** Consider a data matrix of 2d vectors, X, shown in Figure 7.11,

$$\mathsf{X}=\begin{bmatrix}5 & 7 & 23 & 70 & 6 & 5\\ 9 & 5 & 92 & 62 & 60 & 70\end{bmatrix}$$

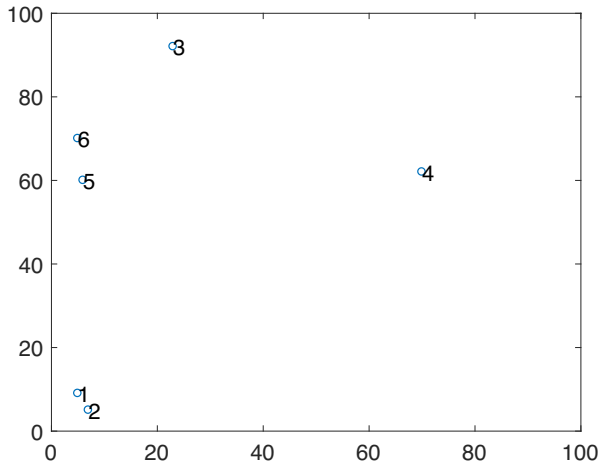


Figure 7.11: 2d Data for Spectral Clustering

Define the similarity matrix, W, such that

$$w_{ij}=\exp(-\text{dist}_{ij}^2/\text{dist}_{mean}{}^2).$$

We calculate the distances between the points as:

$$distances=\begin{bmatrix}0 & 4.47 & 84.9294 & 83.87 & 51.01 & 61\\ 4.47 & 0 & 88.459 & 84.96 & 55.01 & 65.03\\ 84.92 & 88.45 & 0 & 55.76 & 36.24 & 28.43\\ 83.86 & 84.96 & 55.76 & 0 & 64.03 & 65.49\\ 51.01 & 55.01 & 36.24 & 64.03 & 0 & 10.05\\ 61 & 65.03 & 28.43 & 65.49 & 10.05 & 0\end{bmatrix}.$$

We calculate that $\text{dist}_{mean} = 55.9152$, and use this to calculate the similarity matrix, whose elements $w_{ij} = \exp(-\text{dist}_{ij}^2/\sigma^2)$:

$$\mathsf{W} = \begin{bmatrix} 1 & 0.99 & 0.10 & 0.10 & 0.43 & 0.30 \\ 0.99 & 1 & 0.08 & 0.09 & 0.37 & 0.25 \\ 0.09 & 0.08 & 1 & 0.36 & 0.65 & 0.77 \\ 0.10 & 0.09 & 0.36 & 1 & 0.26 & 0.25 \\ 0.43 & 0.37 & 0.65 & 0.26 & 1 & 0.96 \\ 0.30 & 0.25 & 0.77 & 0.25 & 0.96 & 1 \end{bmatrix}.$$

Using this similarity matrix, sum the columns to calculate the elements of the diagonal degree matrix, D,

$$\mathsf{D} = \begin{bmatrix} 2.93785 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.81334 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.98069 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.09786 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3.70971 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3.55686 \end{bmatrix}.$$

Finally, we calculate the Laplacian matrix, $\mathsf{L} = \mathsf{D} - \mathsf{W}$,

$$\mathsf{L} = \begin{bmatrix} 1.937 & -0.993 & -0.099 & -0.105 & -0.435 & -0.304 \\ -0.993 & 1.813 & -0.081 & -0.099 & -0.379 & -0.258 \\ -0.099 & -0.081 & 1.980 & -0.369 & -0.657 & -0.772 \\ -0.105 & -0.099 & -0.369 & 1.097 & -0.269 & -0.253 \\ -0.435 & -0.379 & -0.657 & -0.269 & 2.709 & -0.968 \\ -0.304 & -0.258 & -0.772 & -0.253 & -0.968 & 2.556 \end{bmatrix}.$$

The second and third eigenvectors and eigenvalues of the Laplacian matrix are,

$$\mathsf{F}^T = \begin{bmatrix} 0.4657 & -0.2514 \\ 0.5058 & -0.3086 \\ -0.2338 & 0.5134 \\ -0.6853 & -0.6024 \\ 0.002229 & 0.2774 \\ -0.05465 & 0.3716 \end{bmatrix}$$

$$\text{Eigenvalues} = (0, \mathbf{1.097}, \mathbf{1.599}, 2.851, 2.934, 3.615)$$

Given the second and third smallest eigenvectors of the Laplacian matrix in F, we cluster the six 2d data points into three clusters as shown in Fig. 7.12.

We could use hierarchical or *k*-means to cluster these points, but the three clusters based on the plot are clearly (1,2), (3,5,6), 4 – consistent with the drawing of the original data above. In this case, the original data as well as the Laplacian eigenvectors we ended up clustering were in 2d (since we only needed two eigenvectors), and so clearly there was no efficiency advantage to be gained here. ∎

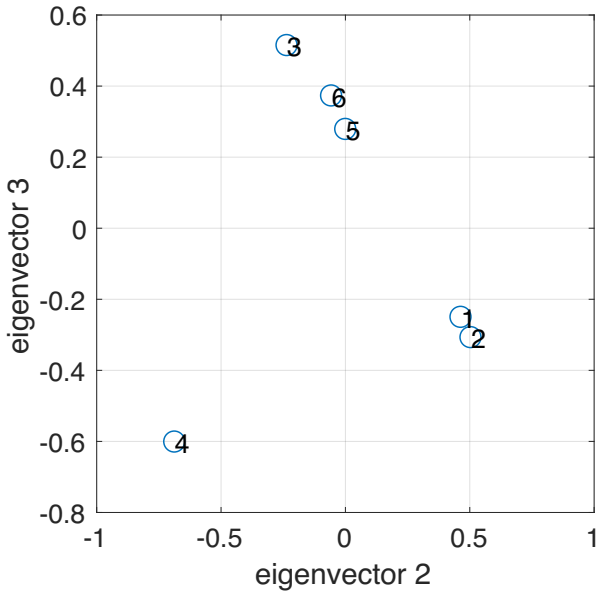


Figure 7.12: Results of spectral clustering.

7.9 BFR algorithm

The Bradley, Fayyad and Reina (BFR) algorithm (Bradley et al., 1998), is a variation on the k-means algorithm, designed to cluster extremely large data sets in a very high dimensional space, and is therefore designed to minimize storage requirements. The algorithm assumes the data to be normally distributed about a centroid. The mean and standard deviation may vary from dimension to dimension and are updated during the processing of the data, but the data in different dimensions must be independent of each other for the algorithm to hold. In 2d, for example, this means that a cluster can take the shape of an ellipse, but not of a rotated ellipse.

The BFR algorithm begins by selecting k initial points, using one of the methods described in Section 7.5. The data points are then read in small groups (chunks) into memory, while summaries of the k clusters are also stored in main memory, including the mean (clusteroid) for each cluster and its standard deviation in each direction. As the data are read into memory, they are divided into three sets:

1. *The discard set*: These are summaries of the clusters themselves as the algorithm progresses. They are called discard sets because, once clustered, the data points represented by the summaries are discarded.
2. *The compressed set*: The compressed sets are summaries for sets of points not close to any of the k clusters initialized in the algorithm, but have been found close to one another. Summaries of these data points are stored as "mini-clusters", and the original data are discarded.
3. *The retained set*: These are the data points that are neither assigned to one of the k clusters, nor to a "mini-cluster". As a result, no summary is kept for these points, and the original data points are kept in memory.

Processing data using BFR

1. For a given chunk of input data, all points that are "sufficiently close" (see below) to the centroid of a cluster are assigned to that cluster and are used to update its clusteroid and standard deviation

in each direction.

2. Points that are not sufficiently close to any of the centroids but are close to other compressed or retained points are clustered with these points. This forms "mini-clusters" – their summaries are kept and the data points eliminated. Remaining single-points that are not close to other retained points or compressed sets are added to the retained set in memory.
3. When possible, updated "mini-clusters" are merged with each other or with the existing k clusters.
4. The above steps are repeated for all chunks of data.
5. Once all data have been processed, data points in the retained set are either noted as outliers, or are assigned to the nearest clusters, if possible.

Definition of "sufficiently close"

Two approaches are discussed in Leskovec et al. (2014),

a. Add a point p to an existing cluster if the centroid is not only the closest but if it is also unlikely that after the remaining points have been processed other cluster centroids will form that are closer to p. This second criterion is strongly linked to the original assumption that the data were normally distributed in each dimension, and to the estimated standard deviations of each cluster in all data directions.
b. Use the Mahalanobis Distance (section 7.7) to estimate the likelihood that p belongs to one of the k clusters.

7.10 CURE (Clustering Using REpresentatives)

The CURE (Clustering Using REpresentatives) algorithm is used for clustering large amounts of data where clusters are expected to take complex shapes, such as two concentric rings. These types of data rings cannot be accurately clustered based on distance from their centroids because the data rings share a centroid at their respective centers. An example is shown in the right panel of Fig. 7.13, where the blue and red clusters share the same clusteroid, yet are clearly distinct. CURE is a point-assignment clustering algorithm that, like k-means, relies on a series of pre-selected representative points that span the entire, complexly-shaped cluster, and uses these points to cluster the rest of the data appropriately. The difference is that CURE uses more than a single representative point for each cluster in order to be able to accommodate the complex cluster shapes.

To initialize CURE,

1. First, a small random sample of data points are clustered using hierarchical clustering based on the "single" linkage (section 7.4.2).
2. The resulting set of points from each cluster (or a subset of these) are designated as its *representatives*, and are ideally as far apart from one another as possible.

Next, we carry out a point assignment on the rest of the data points and assign each new point p to the cluster containing a representative closest to p. An example of the CURE algorithm is shown in Fig. 7.13.

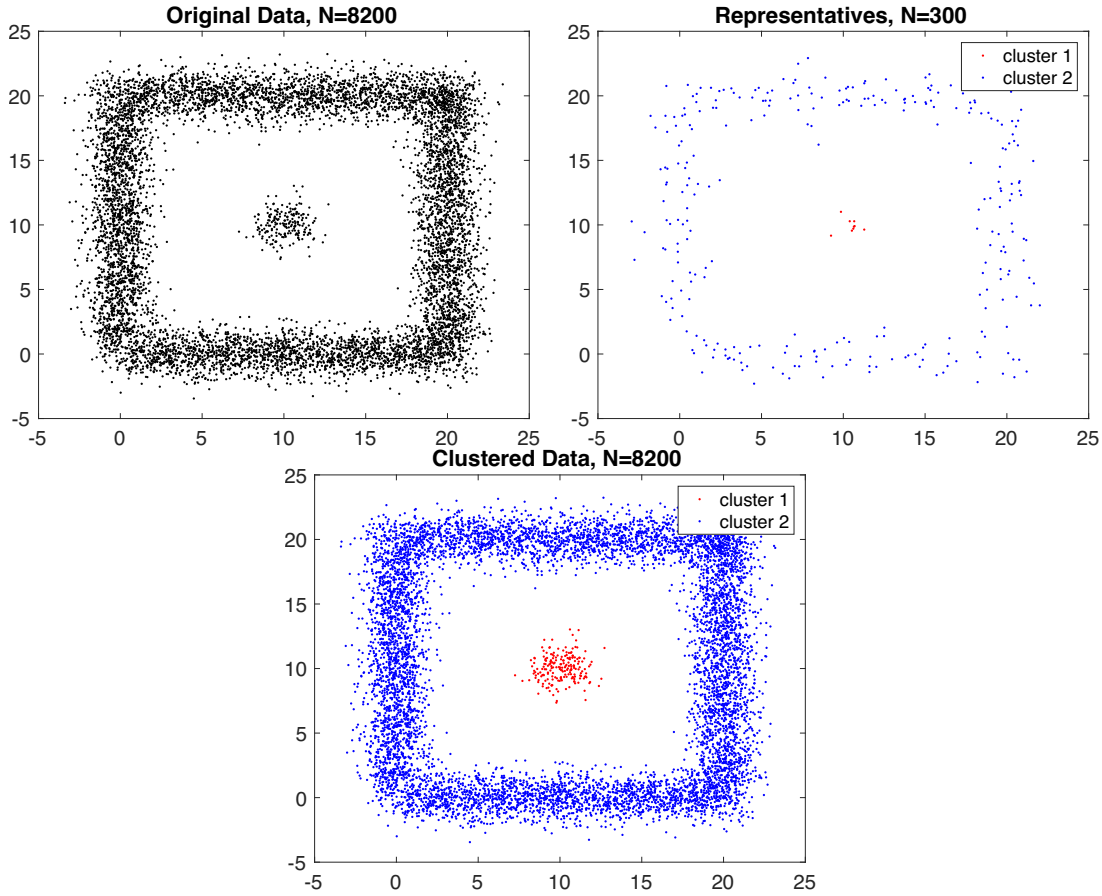


Figure 7.13: The CURE algorithm selection of representatives and final clustering

8. Classification: supervised learning

8.1 Motivation

We now move onto "supervised learning", or "classification" techniques. In all of these classification methods, we have some "training data" points that are divided into classes via a known "label". The objective is to learn how to assign such labels in order to classify new data points. The applications of these "machine learning" classification techniques are broad and continuing to expand, disrupting every data-driven industry. Some key examples of the applications of these algorithms include:

- **Recognition**: Optical character recognition, handwriting recognition, speech recognition, written language recognition (for example, in spam filtering)
- **Social media analysis**: Sentiment analysis of tweets (e.g., angry/sad/happy) in order to predict how similar individuals may vote.
- **Recommendations**: For example, product and book recommendations on Amazon and television recommendations on Netflix.
- **Financial Prediction**: Predicting individual credit scores and predicting loan defaulting based on an individual's financial profile.
- **Advertising**: Advertisers learn about your preferences through searches and use this information to present targeted advertisements.

Unlike the "unsupervised learning" techniques from the previous chapter – where the input data does not inform the clustering algorithm what the clusters should be – in "supervised learning" training data define the classes and allow us to learn the correct classification. From this learning, we can predict how to accurately classify future data. The *training data* contains pairs of data of the structure $(\mathbf{x}, y)$, where $\mathbf{x}$ contains information about a given data point (i.e. its coordinates – text of email, image of hand writing, area and age of a house), and y is the *label* (spam/not spam, which letters is drawn, or what price the house was sold at) of that data point for its given features. The labels may be a real numbers, binary (0 or 1) or part of some finite set of numbers. These known data are then divided into

"training data" used to deduce the algorithm and "validation data" used to test the results.

We begin by exploring perceptrons – a binary, linear classifying technique that later forms the basis of neural networks. Next, support vector machines demonstrate the use of a gradient-based optimization for the learning. Then, we move on to more more sophisticated techniques, such as feedforward neural networks, and finally, we look at k-nearest neighbor algorithms.

8.2 Perceptrons

A perceptron is a hyperplane of dimension $d-1$ that splits up d-dimensional data points into two groups, shown for example as a separating line in two dimensions in Fig. 8.1. Let the dimensionalizing vector, $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ represent a point in data space, where each such data point is assigned a label of either $y = +1$ or $y = -1$. The perceptron is defined by a vector of *weights*, $\mathbf{w} = [w_1, w_2, \ldots, w_d]$. Each perceptron also has a threshold, θ, which is used to determine the classification of the input vector, $\mathbf{x}$. If $\mathbf{w} \cdot \mathbf{x} > \theta$ then the output of the perceptron is $+1$, and if $\mathbf{w} \cdot \mathbf{x} < \theta$ the output is -1. Ideally, the perceptron defined by the weights vector, $\mathbf{w}$, applied to the training data leads to the correct classification of ± 1 for all points. Once a perceptron we are training has correctly classified all training data we stop updating the $\mathbf{w}$ vector, which reveals a potential shortcoming discussed later. The perceptron itself represents the line $\mathbf{w} \cdot \mathbf{x} = \theta$, where $\mathbf{x}$ here is the vector defining the dimensions of the hyperplane. The perceptron should be defined such that it correctly separates positive and negative labels.

8.2.1 Training perceptrons

In training a perceptron to correctly classify future data, we go over the training data vectors, $\mathbf{x}$, and use their respective, known classifications, y, to update the weight vector, $\mathbf{w}$ and threshold, θ, until all the training points are correctly classified. To simplify the training process initially, assume the threshold $\theta = 0$ and consider the following steps:

1. Initialize the weight vector, $\mathbf{w}$, with 0's or random numbers.
2. Initialize a learning rate parameter, η, which is a small, positive real number.
3. For each training data point, $(\mathbf{x}_i, y_i)$, we calculate the classification label $y_i' = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i)$. If the label y_i' is equal to the known, correct y_i, then the perceptron does not need to be adjusted. If, however, the labels are different, then the point is currently incorrectly classified. We update $\mathbf{w}$ by writing $\mathbf{w}_{new} = \mathbf{w}_{old} + \eta y_i \mathbf{x}$, adjusting $\mathbf{w}$ slightly in the direction of $\mathbf{x}$ as seen in Fig. 8.1.

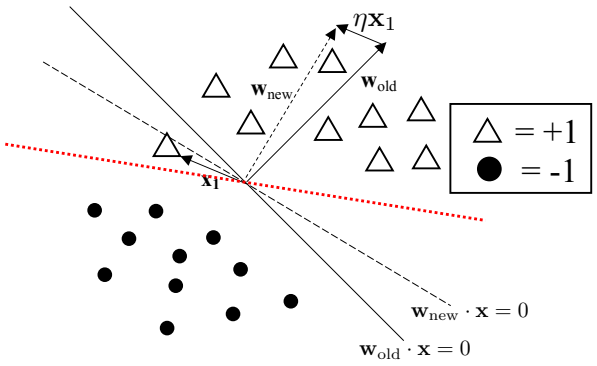


Figure 8.1: Updating a perceptron based on a misclassified point.

Following the usual machine learning paradigm, this procedure is repeated by going over all training data, and then (after possibly randomly shuffling them) going over all points several times until some convergence is reached. Each such pass over all data points is known as an "epoch". While assuming the threshold $\theta = 0$ is helpful for introducing how to update perceptrons, most classification problems require a non-zero, variable threshold $\theta \neq 0$. To generalize the above, we can allow θ to vary by adding θ as a component of the weight vector $\mathbf{w}$, and supplementing the data point vectors as follows,

1. Replace the weight vector, $\mathbf{w} = [w_1, w_2, \ldots, w_d]$ with $\mathbf{w}' = [w_1, w_2, .., w_d, \theta]$
2. Replace every vector $\mathbf{x} = [x_1, x_2, .., x_d]$ with $\mathbf{x}' = [x_1, x_2, .., x_d, -1]$

With these adjustments the perceptron condition with a threshold now becomes $\mathbf{w}' \cdot \mathbf{x}' \geq 0$ which appears to have the form of a perceptron with a zero threshold. However, if we compute the dot product we see this is in fact identical to the desired $\mathbf{w} \cdot \mathbf{x} \geq \theta$. We update the weight vector $\mathbf{w}'$ in the same way as described above for perceptrons with a zero threshold.

Carefully selecting the learning rate η is important: if η is too small the convergence to a correctly defined perceptron will take too many steps, and if η is too large, the $\mathbf{w}$ vector will jump around and converge slowly, if ever. In addition, it is advisable to make the learning rate smaller as the epochs proceed, in order to ensure convergence. One can also use an adjustable learning rate, η, that is, for example, proportional to the deviation of the current data point that is not classified correctly ($\eta = \eta_0 |\mathbf{x} \cdot \mathbf{w} - \theta|$), but bounded on both sides so that, say, $0.01 < \eta < 1$.

8.2.2 Example

■ **Example 8.1** Consider the following 2d training data, X, and the corresponding labels, y:

$$X = \begin{pmatrix} -0.81 & 0.45 & 0.65 & 0.11 & -0.12 \\ 0.87 & -0.87 & -0.05 & -0.76 & 0.43 \end{pmatrix}$$
$$y = \begin{pmatrix} -1 & 1 & 1 & 1 & -1 \end{pmatrix}$$

We start by visualizing the data in Fig. 8.2, where circles are classified $+1$ and crosses are classified -1, and also plot an initial guess for the perceptron line shown in dash green, $[\mathbf{w}, \theta] = [(-1, -1.8), -0.4]$. As a check, remember that a perceptron classifies a point as $+1$ if $\mathbf{w} \cdot \mathbf{x} > \theta$ and as -1 otherwise. The first point is above the line, and is classified correctly as,

$$\begin{aligned} \mathbf{x}_1 &= \begin{bmatrix} -0.81 & 0.87 \end{bmatrix}^T \\ \mathbf{w} &= \begin{bmatrix} -1 & 1 \end{bmatrix}^T \\ \theta &= -0.4 \\ \mathbf{w} \cdot \mathbf{x} - \theta &= -0.356 \\ -0.356 < 0. &\quad \text{Thus, this point is correctly classified as } -1. \end{aligned}$$

To be able to update θ as well, define the augmented $\mathbf{w}$ and X,

$$X = \begin{pmatrix} -0.81 & 0.45 & 0.65 & 0.11 & -0.12 \\ 0.87 & -0.87 & -0.05 & -0.76 & 0.43 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$
$$y = \begin{pmatrix} -1 & 1 & 1 & 1 & -1 \end{pmatrix}$$
$$\mathbf{w} = \begin{bmatrix} -1 & -1.8 & -0.4 \end{bmatrix}^T$$
$$\eta = 0.16$$

Consider point #1 again using these augmented $\mathbf{w}$ and X,

$$\mathbf{w}^T\mathbf{x}_1 = \begin{bmatrix}-1 & -1.8 & -0.4\end{bmatrix}\begin{bmatrix}-0.81\\ 0.87\\ -1\end{bmatrix} = -0.356.$$

< 0 so point 1 should have -1 label, which it does. Correct, no need to adjust.

Similarly, for point #2:

$$\mathbf{w}^T\mathbf{x}_2 = 1.516.$$

> 0 so point 2 should have $+1$ label, which it does. Correct, no need to adjust.

However, for point #3:

$$\mathbf{w}^T\mathbf{x}_3 = -0.16$$

< 0 so perceptron suggests point 3 should have -1 label, however it has a $+1$ label. The perceptron is thus incorrect and we need to adjust.

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta y_3 X_3 = \begin{bmatrix}-1\\ 1\\ -0.4\end{bmatrix} + 0.16 \times 1 \times \begin{bmatrix}0.65\\ -0.05\\ -1\end{bmatrix} = \begin{bmatrix}-0.43\\ -0.86\\ -0.27\end{bmatrix}$$

After this adjustment, point 3 is correctly classified in this case (check). Continuing with the new $\mathbf{w}$, we find that training points #4 and #5 are both correctly classified,

$$\mathbf{w}^T\mathbf{x}_4 = 0.877$$

> 0 so point 4 should have 1 label, which it does. Correct, no need to adjust.

$$\mathbf{w}^T\mathbf{x}_5 = -0.0525$$

< 0 so point 5 should have -1 label, which it does. Correct, no need to adjust.

We can continue this procedure over all points again to ensure this perceptron accurately classifies the training data to confirm our solution, but graphically we can already see the updated perceptron now correctly classifies all training points (blue line in Fig. 8.2). ∎

8.2.3 Problems with perceptrons

Linearly inseparable training data: The training data provided may not be separable with a hyperplane (e.g., blue circle in Fig. 8.3), in which case there will never be convergence, as misclassified points pull the perceptron in opposite directions. To address this, we can generalize the hyperplane by transforming the training data so that the hyperplane is able to separate the data but this may introduce over-fitting issues – we will have constructed a perceptron that is designed to effectively classify the training data, but may fail to correctly classify other input data.

Sub-optimal perceptrons: On the other hand, for two sets of data points that are quite spaced away from one another (e.g., Fig. 8.3 without the blue circle), we may construct several perceptrons that correctly classify the training data, shown in black. These may, however, be suboptimal and biased towards the positively or negatively classified data points. Ideally, the perceptron would be exactly in the center of the space between the two sets as it is then most likely to correctly classify new points that may be in this gap. That said, the perceptron training algorithm discussed

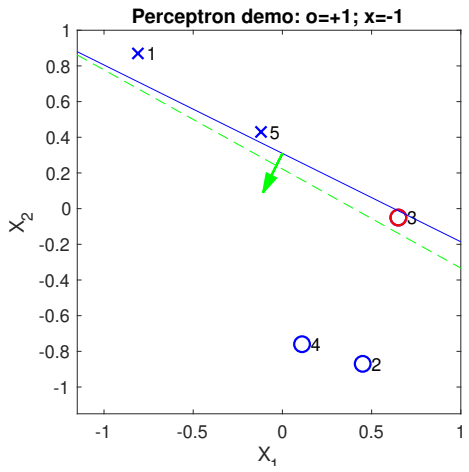


Figure 8.2: Perceptron example. Point 3 is initially misclassified (dash green line), and this is corrected after the update (blue line).

above stops once the perceptron accurately classifies all training data. Thus, we may derive perceptrons that correctly classify the training data but have a small margin for error and thus fail to classify new data in the gap between the two distinct clusters. We will see in the next chapter how support vector machines address this shortcoming of perceptrons.

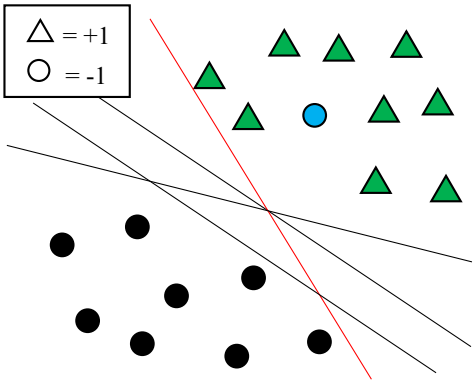


Figure 8.3: While the red perceptron correctly classifies the training data (in green), it fails to correctly classify additional data (in blue).

8.2.4 An extension to non-linear hyperplanes

Consider a 2d data set with the coordinates of each point given by $\mathbf{x}_i = (x_1, x_2)$. If a linear perceptron of the form $w_1x_1 + w_2x_2 - \theta = 0$ cannot classify the data points because they are not separable by a straight line or hyperplane, it is possible to construct a non-linear separating line that may be able to perform the classification. Suppose we are looking for a quadratic line that separates the two data sets, given by $x_2 = Ax_1 + Bx_1^2 + C$, or, more generally, by $w_1x_1 + w_2x_2 + w_3x_1^2 - \theta = 0$. Instead of looking for a weight vector with three elements in the linear two-dimensional case, we look for a weight vector

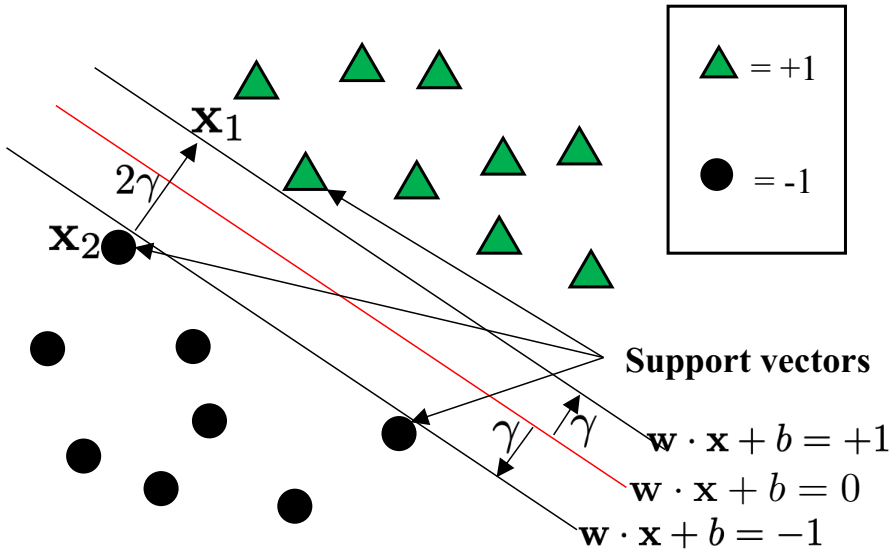


Figure 8.4: An SVM selects a feasible separating line (red line) at a distance γ from the distinct datasets, maximizing the likelihood of correctly classifying future data inputs.

with four elements. The training data need to be extended accordingly, such that each data vector is now $\mathbf{x}_i' = (x_1, x_2, x_1^2, 1)$ and $\mathbf{w}' = [w_1, w_2, w_3, \theta]$. We then proceed with the learning algorithm to find $\mathbf{w}'$ as before. It is straightforward to generalize this to more general or higher order non-linearities (e.g., $w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2 + w_6x_1^3 + w_7 = 0$); this extension to a non-linear hyperplane also works for support vector machines (SVMs) covered in the next section.

8.3 Support vector machines

To remedy the above issues with perceptrons, we next discuss support vector machines (SVMs). Unlike a perceptron, whose training halts as soon as a hyperplane that correctly classifies the training data is found, a support vector machine selects a specific hyperplane $\mathbf{w}\cdot\mathbf{x}+b=0$ that maximizes the distance γ between the hyperplane and the closest points of the training set, as shown in the Fig. 8.4.

Furthermore, we also see in Fig. 8.4 two parallel hyperplanes at distance γ from the central hyperplane (the SVM, in red), $\mathbf{w}\cdot\mathbf{x}+b=0$. Each of these parallel hyperplanes touches a few "support vectors" – the training points that actually constrain the dividing hyperplane by being at a distance γ away. A d-dimensional set of points generally has $d+1$ support vectors, as shown in Fig. 8.5.

As a preliminary attempt to construct a support vector machine, for each data point, $(\mathbf{x}_i, y_i)$, in our n-point training set, we would like to find the weights $\mathbf{w}$ that maximize the distance, γ, from the data to the SVM hyperplane, subject to the constraint that if $y=+1$, then $\mathbf{w}\cdot\mathbf{x}\geq\gamma$, and if $y=-1$, then $\mathbf{w}\cdot\mathbf{x}\leq-\gamma$. Equivalently, we may re-write this constraint as,

$$y_i(\mathbf{w}\cdot\mathbf{x}_i+b)\geq\gamma \qquad \text{for all } i=1,2,..,n. \tag{8.1}$$

However, this formulation does not lead to a unique solution for the weights, because given a set of weights $\mathbf{w}$ that satisfies (8.1), we can always multiply the weights by a factor of 2 and the distance γ is similarly scaled; we can just arbitrarily multiply the weights to maximize γ, which is trivial.

To address this shortcoming, we normalize the weight vector by requiring that the two planes on the two sides of the SVM plane separating the two data sets are represented by the lines $\mathbf{w}\cdot\mathbf{x}_i+b=\pm1$.

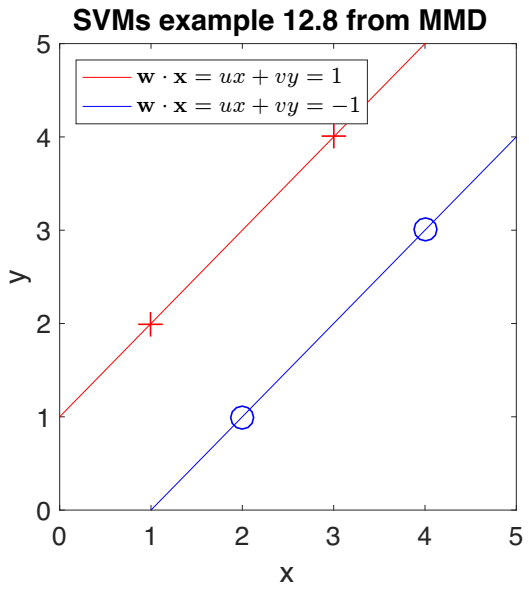


Figure 8.5: We can see there can be more than $d+1$ support vectors in d dimensions.

This means that we now cannot multiply $\mathbf{w}$ by some factor and can proceed to calculate γ in this case. Consider two points $\mathbf{x}_1$, $\mathbf{x}_2$ on the two planes above and below the SVM separating plane, just across from one another. The distance between them is 2γ, and the direction of the vector separating them is that of the unit vector $\mathbf{w}/\|\mathbf{w}\|$. We can therefore write, $\mathbf{x}_1 = \mathbf{x}_2 + 2\gamma\mathbf{w}/\|\mathbf{w}\|$. At the same time, because $\mathbf{x}_1$ is on the upper hyperplane, we have $\mathbf{w}\cdot\mathbf{x}_1 + b = +1$. Substituting this expression for $\mathbf{x}_1$ into the relation between $\mathbf{x}_1$, $\mathbf{x}_2$ we find, $\mathbf{w}\cdot(\mathbf{x}_2 + 2\gamma\mathbf{w}/\|\mathbf{w}\|) + b = +1$. Now use $\mathbf{w}\cdot\mathbf{x}_2 + b = -1$ to find, $\mathbf{w}\cdot(2\gamma\mathbf{w}/\|\mathbf{w}\|) = +2$, or $\gamma\mathbf{w}\cdot\mathbf{w}/\|\mathbf{w}\| = \gamma\|\mathbf{w}\|^2/\|\mathbf{w}\| = +1$, which leads to $\gamma = 1/\|\mathbf{w}\|$.

Our corrected optimization problem therefore becomes minimizing $\|\mathbf{w}\|$, which is now the inverse of γ, by varying $\mathbf{w}$ and b subject to the constraint that,

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 \qquad \text{for all } i = 1, 2, \ldots, n. \tag{8.2}$$

8.3.1 Calcualting the SVM by gradient-based minimization

In applications, noise in the data may not allow us to find a plane that perfectly separates the two data sets, and therefore instead of minimizing $\|\mathbf{w}\|$ subject to the above constraint (8.2), we seek to minimize the following cost function,

$$f(\mathbf{w}, b) = \frac{1}{2}\sum_{j=1}^{d} w_i^2 + C\sum_{i=1}^{n} \max\{0, 1 - y_i(\mathbf{w}\cdot\mathbf{x}_i + b)\}. \tag{8.3}$$

The first term is minimized for small $\mathbf{w}$, and therefore large γ. The second term penalizes misclassified points for a given $\mathbf{w}$ and thus contains a sum over the penalty function $L(\mathbf{x}_i, y_i) = \max\{0, 1 - y_i(\mathbf{w}\cdot\mathbf{x}_i + b)\}$. If a point is correctly classified by the hyperplane defined by $\mathbf{w}$ and b, then $1 - y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \leq 0$, and thus $L(\mathbf{x}_i, y_i) = 0$ and no penalty is incurred. However, if a point $(\mathbf{x}_i, y_i)$ is too close to the hyperplane, or is misclassified by it, then $1 - y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 0$, $L(\mathbf{x}_i, y_i) \geq 0$, and the penalty function contributes to a larger $f(\mathbf{w}, b)$. Thus, the minimum of the penalty function (8.3) achieves our goal of

classifying all points as well as possible while making the margin γ as large as possible, thus placing the SVM plane as close as possible to the center of the gap between the two groups of points.

Note that the second term in the cost function is multiplied by a constant, C (the "regularization parameter" or "Penalty constant"). For large C, a misclassified point is more heavily penalized – thus, we select a large C if we want very few misclassified points, but a narrow margin, and conversely a small C if we accept a few misclassified points, but a larger margin (a hyperplane further away from the data boundaries of the two classes). In the case of data composed of two clusters that can be well separated, it is useful to choose a large $C = 1000$, or so, to find an appropriate solution.

As with perceptrons, we can transform to a problem without a threshold b by adding 1 as an extra dimension to each data point, and adding b to the weight vector,

$$\mathbf{w}' = [\mathbf{w}, b]; \quad \mathbf{x}'_i = [\mathbf{x}_i, 1].$$

(Note that we added -1 to each data point for perceptrons, because they were defined as $\mathbf{w}\cdot\mathbf{x} > \theta$, while we are adding 1 to the SVM which is defined as $\mathbf{w}\cdot\mathbf{x} + b > 0$).

The transformation to a threshold-less format allows us to write the penalty as,

$$f(\mathbf{w}') = \frac{1}{2}\sum_{j=1}^{d}(w'_i)^2 + C\sum_{i=1}^{n}\max\{0, 1 - y_i\mathbf{w}\cdot\mathbf{x}'_i\}. \tag{8.4}$$

Finally, we find the derivative of the penalty function by gradient-based minimization. Construct the gradient of the cost function with respect to the components of $\mathbf{w}$,

$$\frac{\partial f}{\partial w_j{}'} = w'_j + C\sum_{i=1}^{n}\left(\text{if } y_i\left(\sum_{j=1}^{d} w'_j x'_{ij}\right) \geq 1 \text{ then 0 else } -y_i x'_{ij}\right) \tag{8.5}$$

The gradient points in the direction of steepest increase of the penalty function as a function of the weights. To minimize $f(\mathbf{w}, b) = f(\mathbf{w}')$, we thus move the w'_j components in opposite directions with respect to the gradient. Using a constant learning rate η, we write the following iterative scheme for the weights,

$$w'^{k+1}_j = w'^{k}_j - \eta\frac{\partial f}{\partial w'_j} \qquad \text{for all } j = 1, 2, \ldots, d+1. \tag{8.6}$$

For sufficiently large k, this should approach the minimum of the penalty with respect to the weights $\mathbf{w}' = [\mathbf{w}, b]$.

8.3.2 Example

■ **Example 8.2** Consider the following example training data, $(X, \mathbf{y})$, and the initial guess for $\mathbf{w}, b$. The threshold b is found by adding an extra data point 1 to the original data points and making b the final element in a revised $\mathbf{w}$ vector, $\mathbf{w}' = [\mathbf{w}, b]$.

$$X = \begin{pmatrix} -0.8 & -0.2 & 0.65 & 0.11 & -0.12 \\ 0.87 & 0.5 & -0.05 & -0.76 & 0.43 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} 1 & -1 & -1 & -1 & 1 \end{pmatrix}$$

$$\mathbf{w}' = [\mathbf{w}, b] = \begin{bmatrix} 10 & 10 & 0 \end{bmatrix}^T$$

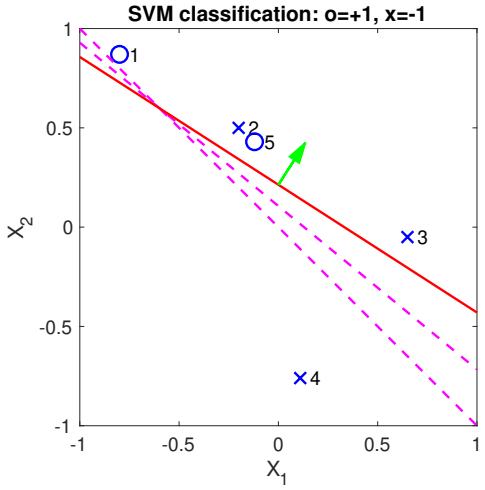


Figure 8.6: An example of two steepest decent SVM iterations, see text.

We can see from Fig. 8.6 below that it is not possible to partition the training data into two groups using a single straight line, but we can still look for the optimal SVM.

We carry out the first two steepest descent iterations, using a learning rate of $\eta = 0.1$, and a penalty constant of $C = 10$. With number of dimensions in the augmented data $d = 3$ and number of data points $n = 5$, the cost function to be minimized is given by,

$$f(\mathbf{w}') = \frac{1}{2}\sum_{i=1}^{3} w_i'^2 + C\sum_{i=1}^{5} \max\{0, 1 - y_i \mathbf{w}' \cdot \mathbf{x}_i'\}$$

The gradient with respect to $\mathbf{w}'$ is therefore given by,

$$\frac{\partial f(\mathbf{w}')}{\partial w_k'} = w_k' + C\sum_{i=1}^{n} \{\text{if } 1 - y_i \mathbf{w}' \cdot \mathbf{x}_i' < 0 \text{ then } 0, \text{ else } -y_i x_{ij}'\} = w_k' + C\sum_{i=1}^{n} G_i.$$

where we denote with G_i the contribution of data point i to the gradient of $f(\mathbf{w}')$. In the first iteration, we go over all data points and check the inequality $1 - y_i(\mathbf{w}' \cdot \mathbf{x}_i) < 0$ to determine the contribution G_i of each data point to the gradient updating. We find that the initial cost is 213, and,

$$\begin{aligned}
&i = 1, && C \times G_i = [8, -8.7, -10] \\
&i = 2, && C \times G_i = [-2, 5, 10] \\
&i = 3, && C \times G_i = [6.5, -0.5, 10] \\
&i = 4, && C \times G_i = [0, 0, 0] \\
&i = 5, && C \times G_i = [0, 0, 0]
\end{aligned}$$

Thus, at the end of this first iteration, $\partial f(\mathbf{w}')/\partial w_k' = [22.5, 5.8, 10]^T$, and the new estimate for the weights is,

$$\mathbf{w}' = \begin{bmatrix} 10 \\ 10 \\ 0 \end{bmatrix} - \eta \begin{bmatrix} 22.5 \\ 5.8 \\ 10 \end{bmatrix} = \begin{bmatrix} 7.75 \\ 9.42 \\ -1 \end{bmatrix},$$

and the cost is reduced to 152.21. Proceeding to the next steepest descent iteration,

$$\begin{aligned} i=1, &\qquad C\times G_i=[8,-8.7,-10]\\ i=2, &\qquad C\times G_i=[-2,5,10]\\ i=3, &\qquad C\times G_i=[6.5,-0.5,10]\\ i=4, &\qquad C\times G_i=[0,0,0]\\ i=5, &\qquad C\times G_i=[0,0,0] \end{aligned}$$

and now, $\partial f(\mathbf{w}')/\partial w'_k=[20.25,5.22,9]^T$ and the new estimate for the weights is

$$\mathbf{w}'=\begin{bmatrix}7.75\\9.42\\-1\end{bmatrix}-\eta\begin{bmatrix}20.25\\5.22\\9\end{bmatrix}=\begin{bmatrix}5.725\\8.898\\-1.9\end{bmatrix}.$$

The initial SVM and the two next iterations are seen in Fig. 8.6. We can continue with further iterations and will converge to an optimal SVM. An actual convergence requires making the learning rate smaller, in this case. ∎

8.4 Multi-Layer Artificial Neural Networks

8.4.1 Motivation

Neural networks are a machine learning classifier that have their foundations in the biological operation of neurons – cells in our body that transmit electric signals and in doing so determine human responses. Each neuron in the human body collects electrical and chemical inputs through branch-like structures known as dendrites. The neuron accumulates and analyzes these inputs from dendrites, and based on some criterion or threshold makes a decision to "fire", passing an output electrical response down its axon to structures at the end of the neuron known as boutons. These boutons are linked to thousands of other neurons via synapse connections. The human reaction, thus, can be explained by a series of electrochemical reactions and "decisions" made in neurons that ultimately lead to some physical response. This whole process happens in a matter of milliseconds.

Artificial neural networks draw on the concept of the human neuron to mimic its behavior: taking in inputs, processing these inputs, and determining an output using an activation function. The basic principles behind an artificial neural network link closely to our earlier discussion of perceptrons – linear binary classifiers that, using a weight vector, determine one of two classifications for a given input. Perceptrons take the dot product of an input vector, $\mathbf{x}$, and a trained weight vector, $\mathbf{w}$, and check to see if the result is above or below a calculated optimal threshold, b, to classify it with a "$+1$" or "-1" label. Thus, artificial neural networks may be taught of as a collection of perceptrons digesting information and deciding whether to fire. To see the similarities and the key differences between between neural networks and perceptrons, consider the following examples of two functions, the OR function, and the XOR function.

Fig. 8.7 shows these two functions. The OR function takes two inputs and outputs 1 if at least one of the input elements is 1, and 0 otherwise. The function can be reproduced by a simple perceptron with a weight vector $\mathbf{w}=[1,1]$ and a threshold $\theta=1$.

The XOR function outputs a label of 1 if the inputs are different, and 0 otherwise. Fig. 8.7 shows that these labels are not separate (cannot be classified) by a line, and thus a perceptron cannot be used to classify these data. This can be addressed by using a neural network with a single *hidden layer*,

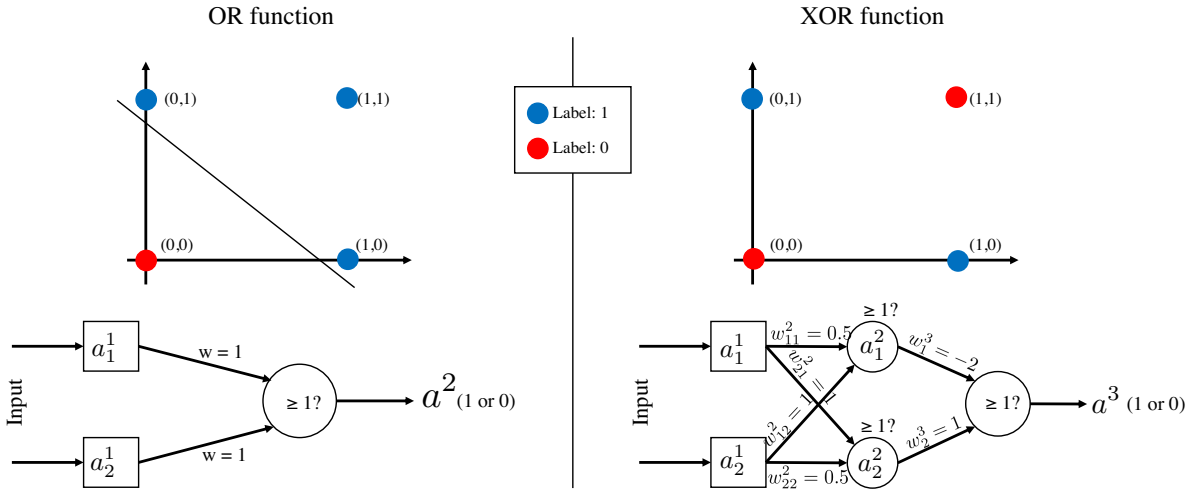


Figure 8.7: The XOR function cannot be classified by a linear perceptron, but can be well-classified with a neural network with a single hidden layer.

connecting the input and output layers, as shown in the right panel of Fig. 8.7. Consider, for example the input vector $(a_1^1, a_2^1) = (1,1)$. Using the specified weights, we calculate the hidden layer neuron values by first multiplying the inputs by the appropriate weights and testing against the threshold: $w_{11}^2 a_1^1 + w_{12}^2 a_2^1 = 1.5 > 1$ and therefore $a_1^2 = 1$. Similarly, $w_{12}^2 a_1^1 + w_{22}^2 a_2^1 = 1a_1^1 + 0.5a_2^1 = 1.5 > 1$, and therefore $a_1^2 = 1$. We use these to calculate the output neurons as follows $-2a_1^2 + 1a_2^2 = -1 < 1$ and therefore $a^3 = 0$, as it should be. Try this with other inputs to verify that XOR is indeed well-represented by this simple neural network.

The above calculation for a_1^2, for example, may be expressed as follows,

$$
\begin{aligned}
z_1^2 &= w_{11}^2 a_1^1 + w_{12}^2 a_2^1 - 1 \\
a_1^2 &= \sigma(z_1^2), \\
\sigma(z) &= \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}
\end{aligned}
$$

where σ is the step function (Heaviside function). In this formulation, the -1 added to z_1^2 is referred to as the "bias" and the step function is referred to as the "activation function", determining what the output of the neuron is as function of its input z_1^2.

The applications of feed-forward neural networks are broad-ranging, with examples including image and handwriting recognition, language translation, facial and voice recognition, and chess playing.

8.4.2 Evaluating a neural network (feedforward)

A neural network, following the above example, contains an input layer, several hidden layers, and an output layer. When designing a neural network, we must be careful to consider the design the correct architecture, with the optimal number of hidden layers for accuracy and efficient calculation. The value of the jth neuron in the ℓth layer is denoted a_j^ℓ. The relationship between each two layers is defined by

the elements of a weight matrix, where w^{ℓ}_{jk}, is the weight connecting the kth neuron in the $(\ell-1)$th layer to the jth neuron in the ℓth layer. Each neuron also has a "bias", denoted by b^{ℓ}_{j}, for the jth neuron in the ℓth layer, for the activation of the jth neuron in the ℓth layer. The activation function may be denoted $\sigma(x)$, and thus the output of the neural network is estimated from the input as follows,

$$\begin{aligned}
z^2_i &= \sum_j w^2_{ij} a^1_j + b_i \\
a^2_i &= \sigma(z^2_i) \\
&\cdots \\
z^{\ell}_i &= \sum_j w^{\ell}_{ij} a^{\ell-1}_j + b^{\ell}_i \\
a^{\ell}_i &= \sigma(z^{\ell}_i) \\
&\cdots \\
z^{L}_i &= \sum_j w^{L}_{ij} a^{L-1}_j + b^{L}_i \\
a^{L}_i &= \sigma(z^{L}_i).
\end{aligned}$$

Each step may also be written in matrix form as

$$\begin{aligned}
\mathbf{z}^{\ell} &= \mathsf{W}^{\ell}\mathbf{a}^{\ell-1} + \mathbf{b}^{\ell} \\
\mathbf{a}^{\ell} &= \sigma(\mathbf{z}^{\ell}),
\end{aligned}$$

where the activation function is applied separately to each element of the input vector. See an example architecture for a feedforward neural network below in Fig. 8.9.

There are several helpful activation functions, $\sigma(x)$, that we often use,

1. Tansig function: $\text{tansig}(x) = 2(1+\exp(-2x))^{-1} - 1$
2. Sigmoid function: $\text{sigmoid}(x) = (1+\exp(-x))^{-1}$
3. Softplus function: $\text{softplus}(x) = \log(1+\exp(x))$
4. Rectified linear: $\sigma(x) = 0$, if $x \leq 0$, $\sigma(x) = ax$ otherwise
5. Linear: $\sigma(x) = ax$.

Typically, the same activation function is used for all layers in a given network except for in the output layer, where the chosen activation fundction depends on the objective output of the network:

Binary outputs: For a binary output layer, deciding between two options represented by 1 and 0 (Yes and No), a sigmoid or similar output activation function is used.

Multiple discrete labels: Where a classification between multiple classes is needed, for example when deciding which handwritten letter is described by a given image, the softmax function can be used, generalizing the sigmoid function used for binary outputs. The softmax activation function is defined as,

$$\text{softmax}(\mathbf{x}) = \frac{\exp(\mathbf{x})}{\sum_i \exp(x_i)}$$

For example, if we input the vector,

$$\begin{aligned}
\mathbf{x} &= \begin{pmatrix} 1 & 2 & 3 & 4 & 1 & 2 & 3 \end{pmatrix}^T \\
\text{softmax}(\mathbf{x}) &= \begin{pmatrix} 0.024 & 0.064 & 0.175 & 0.475 & 0.024 & 0.064 & 0.175 \end{pmatrix}^T.
\end{aligned}$$

We can see that the softmax activation function effectively highlights the largest values and suppresses the values significantly below the maximum entry. In MATLAB, we apply a softmax output layer and use the vec2ind function to get our output.

Regression: When the network needs to calculate a number or several numbers within a continuous range (e.g., price of a house, value of CO_2 flux from a forest to the atmosphere), a linear or rectified linear output layer is often appropriate.

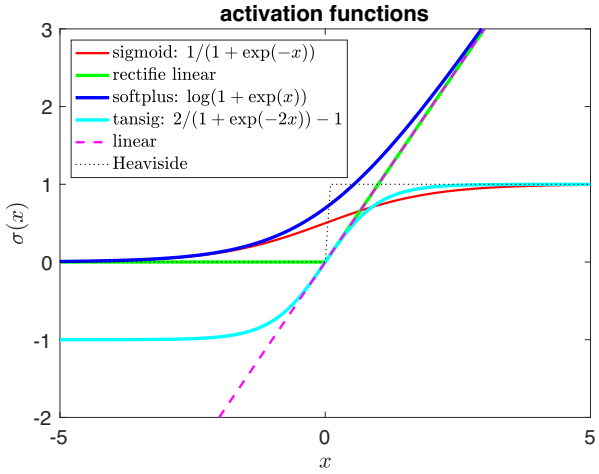


Figure 8.8: Example activation functions

8.4.2.1 Examples

Example – Using MATLAB

■ **Example 8.3** Consider the training data shown in the left panel of Fig. 8.9, where there are two input parameters (x,y) and three possible labels marked 1, 2, 3. The training data are used to train the network with two hidden layers to classify the training data into these three classes, such that each class is defined as one of the three vectors,

$$\mathbf{a} = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \mathbf{b} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \mathbf{c} = \begin{bmatrix}0\\0\\1\end{bmatrix}.$$

The training involves finding the weights and biases $\mathrm{W}^\ell, \mathbf{b}^\ell$ that lead to the best classification of the training data, the algorithm for which is described in section 8.4.3.

The trained network is given by the following weights and biases,

$$\mathrm{W}^2 = \begin{bmatrix}-1.55907 & 1.27589\\ -0.736139 & -0.589527\end{bmatrix}, \quad \mathbf{b}^2 = \begin{bmatrix}0.00936571\\ 7.94701\end{bmatrix}$$
$$\mathrm{W}^3 = \begin{bmatrix}-7.08115 & 5.63232\\ 5.1499 & -9.04534\end{bmatrix}, \quad \mathbf{b}^3 = \begin{bmatrix}-2.00214\\ -2.07198\end{bmatrix}$$
$$\mathrm{W}^4 = \begin{bmatrix}-3.29493 & 9.78708\\ -5.59981 & -6.17265\\ 9.01543 & -3.69417\end{bmatrix}, \quad \mathbf{b}^4 = \begin{bmatrix}1.62788\\ -1.04292\\ 1.20986\end{bmatrix}.$$

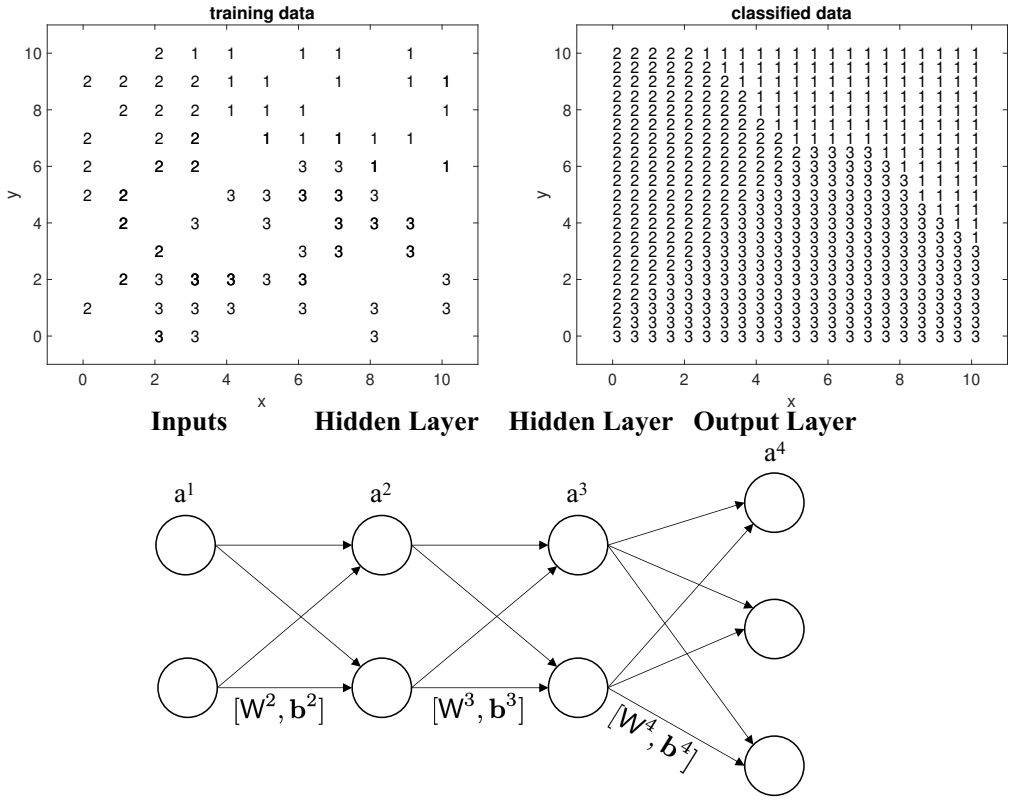


Figure 8.9: Training data and classification by a the neural network with 2 hidden layers

Given these weights, an input vector $\mathbf{a}^1 = (x_1, x_2)^T$ is classified as follows,

$$\begin{aligned}
\mathbf{a}^2 &= \text{tansig}(\mathsf{W}^2\mathbf{a}^1 + \mathbf{b}^2) \\
\mathbf{a}^3 &= \text{tansig}(\mathsf{W}^3\mathbf{a}^2 + \mathbf{b}^3) \\
\mathbf{a}^4 &= \text{softmax}(\mathsf{W}^4\mathbf{a}^3 + \mathbf{b}^4) \\
output &= \text{vec2ind}(\mathbf{a}^4).
\end{aligned}$$

The results of applying this network to a grid of points in the (x, y) plane are shown in the right panel of Fig. 8.9, and this simple network is clearly able to classify all the training data and come up with a reasonable classification of the entire two dimensional input plane. ∎

Example – Hand Calculation

∎ **Example 8.4** Consider the following step-by-step hand calculation for a similar but slightly different feedforward neural network which similarly classifies the input data into three classes, such that the class is defined as one of the three vectors,

$$\mathbf{a} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The network is given by the weights,

$$w_2 = \begin{bmatrix} -1.4862 & 1.2127 \\ -0.7293 & -0.5830 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 0.0105 \\ 7.8570 \end{bmatrix}$$

$$w_3 = \begin{bmatrix} -6.7012 & 5.2272 \\ 4.7825 & -8.4793 \end{bmatrix} \qquad b_3 = \begin{bmatrix} -1.9767 \\ -1.9729 \end{bmatrix}$$

$$w_4 = \begin{bmatrix} -3.1456 & 9.2062 \\ -5.1856 & -5.8752 \\ 8.4518 & -3.4107 \end{bmatrix} \qquad b_4 = \begin{bmatrix} 1.4622 \\ -0.9299 \\ 1.2625 \end{bmatrix}$$

and by the transfer functions

$$\begin{aligned}
\text{tansig}(\mathbf{x}) &= 2/(1 + \exp(-2\mathbf{x})) - 1 \\
\text{softmax}(\mathbf{x}) &= \exp(\mathbf{x}) / \sum_i \exp(x_i)
\end{aligned}$$

where these functions, when applied to vectors, are taken for each component separately. These activation functions may be written in MATLAB as "anonymous functions" using,

```
tansig= @(x) 2./(1+exp(-2.*x))-1;
softmax=@(x) exp(x)/sum(exp(x(:)));
```

Given the network, an input vector $\mathbf{a}^1 = (x_1, x_2)^T$ is classified as follows,

$$\begin{aligned}
\mathbf{a}^2 &= \text{tansig}(\mathsf{w}^2\mathbf{a}^1 + \mathbf{b}^2) \\
\mathbf{a}^3 &= \text{tansig}(\mathsf{w}^3\mathbf{a}^2 + \mathbf{b}^3) \\
\mathbf{y} = \mathbf{a}^4 &= \text{softmax}(\mathsf{w}^4\mathbf{a}^3 + \mathbf{b}^4);
\end{aligned}$$

where the matrix operations may be written explicitly, for example, as

$$\mathbf{z}^2 = \mathrm{w}^2\mathbf{a}^1 + \mathbf{b}^2, \quad z_i^2 = \sum_j w_{ij}^2 a_j^1 + b_i^2.$$

We would like to now apply this network to the following data point $x_2 = [2\ 8]^T$ to find its classification.

First, we calculate the output from the first layer into the second layer by finding the intermediate z_2 and input it into the tansig activation function to find a_2:

$$a_1 = \begin{bmatrix} 2 \\ 8 \end{bmatrix}$$

$$z_2 = (w_2 * a_1 + b_2) = \begin{bmatrix} -1.4862 & 1.2127 \\ -0.7293 & -0.5830 \end{bmatrix} \begin{bmatrix} 2 \\ 8 \end{bmatrix} + \begin{bmatrix} 0.0105 \\ 7.8570 \end{bmatrix}$$

$$z_2 = \begin{bmatrix} 6.7397 \\ 1.7344 \end{bmatrix}$$

$$a_2 = \text{tansig}(z_2) = \begin{bmatrix} 0.999997 \\ 0.939574 \end{bmatrix}$$

Continuing we find that,

$$z_3 = (w_3 * a_2 + b_3) = \begin{bmatrix} -6.7012 & 5.2272 \\ 4.7825 & -8.4793 \end{bmatrix} \begin{bmatrix} 0.999997 \\ 0.939574 \end{bmatrix} + \begin{bmatrix} -1.9767 \\ -1.9729 \end{bmatrix}$$

$$z_3 = \begin{bmatrix} -3.76654 \\ -5.15734 \end{bmatrix}$$

$$a_4 = \text{tansig}(z_3) = \begin{bmatrix} -0.99893 \\ -0.999934 \end{bmatrix}$$

$$z_4 = (w_4 * a_3 + b_4) = \begin{bmatrix} -3.1456 & 9.2062 \\ -5.1856 & -5.8752 \\ 8.4518 & -3.4107 \end{bmatrix} \begin{bmatrix} -0.99893 \\ -0.999934 \end{bmatrix} + \begin{bmatrix} 1.4622 \\ -0.9299 \\ 1.2625 \end{bmatrix}$$

$$z_4 = \begin{bmatrix} -4.60115 \\ 10.125 \\ -3.76979 \end{bmatrix}$$

$$a_4 = \text{softmax}(z_4) = \begin{bmatrix} 4.02279e^{-07} \\ 0.999999 \\ 9.23817e^{-07} \end{bmatrix} \approx \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Thus, the point $[2, 8]^T$ is classified as class $\mathbf{b}$. ∎

8.4.3 Back-propagation

Next, consider how the weights of a neural network are calculated using the training data. The training data are given by data vectors $\mathbf{x}_i$, $i = 1, \ldots, n$ and corresponding labels $y(\mathbf{x}_i)$. A classification/label can be a binary (0 or 1) label, a discrete label that can take multiple values, a continuous scalar value, or a vector of continuous values. Start by defining a *cost function* that measures the success of the network in classifying the training points,

$$C(\mathrm{W}, \mathbf{b}) = \frac{1}{2n} \sum_{i=1}^{n} \|y(\mathbf{x}_i) - \mathbf{a}^L\|^2 \tag{8.7}$$

where W denotes the collection of weights in the network, $\mathbf{b}$ all the biases in the network, $\mathbf{a}^L$ is the vector of output labels from the network for a given input vector $\mathbf{x}_i$, n is the number of training points, and $y(\mathbf{x}_i)$ is the training label which the network should match. Thus, the quadratic cost function is, in other words, the mean squared error of the neural network, and is constructed to be small when the output matches the training labels. The objective of the training is to adjust the weights of the neural network in order to minimize the cost function. To achieve this, we use a *gradient descent* algorithm, as done when updating SVMs, and compute the gradients with respect to the cost function via a process known as *back-propagation*.

Given the cost gradient, we update determine the weights and biases using a learning rate η,

$$w_{ij}^{\ell} \to w_{ij}^{\ell} - \eta \frac{\partial C}{\partial w_{ij}^{\ell}}$$
$$b_{i}^{\ell} \to b_{i}^{\ell} - \eta \frac{\partial C}{\partial b_{i}^{\ell}}.$$

By repeating the process, the cost is minimized and the weights and biases are able to accurately classify data points. A typical algorithm for such training involves calculating the gradient for a "batch" of, say m, input training data, calculates the averaged cost and averaged gradient for this batch, and uses it to updates the weights. Then, the next batch is processed and so on. Once all training data points have been processed (a single "epoch"), the training data are shuffled and used again and again, until convergence is achieved.

The back-propagation algorithm efficiently calculates the gradients used in the gradient-based optimization for training the network. To appreciate its efficiency, consider the brute-force approach for calculating the gradient. One could estimate the cost for a set of weights and biases, $C_1 = C(\mathsf{W}, \mathbf{b})$, then introduce a small perturbation into only one of the weights or the biases, say δw_{ij}^{ℓ}, recalculate the cost $C_2 = C(\mathsf{W} + \delta w_{ij}^{\ell}, \mathbf{b})$, and approximate the gradient using $(C_2 - C_1)/\delta w_{ij}^{\ell}$ (via a Taylor series approximation). This would need to be repeated for each weight denoted by i, j, for each layer ℓ, and a large network could have millions of such weights. Thus, this brute force approach may require evaluating the network millions of times and thus is computationally inefficient. Instead, the back-propagation algorithm evaluates the gradient with respect to all weights and biases using a computational cost equivalent to evaluating the network only once.

Before moving into the details of the back-propagation algorithm, we need to introduce a less-frequently used algebraic tool in linear algebra, the Hadamard product, $\mathbf{a} \odot \mathbf{b}$. For two vectors, the Hadamard product is the result of the multiplication of each component two vectors of equal dimension to produce a new vector. For example,

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \quad \mathbf{a} \odot \mathbf{b} = \begin{bmatrix} 1*4 \\ 2*5 \\ 3*6 \end{bmatrix} = \begin{bmatrix} 4 \\ 10 \\ 18 \end{bmatrix}$$

In order to determine the partial derivatives $\partial C/\partial w_{jk}^{\ell}$ and $\partial C/\partial b_{j}^{\ell}$, define an intermediate quantity, denoted δ_j^{ℓ} – the sensitivity of the cost to the input into the jth neuron of the ℓth layer,

$$\delta_j^{\ell} \equiv \frac{\partial C}{\partial z_j^{\ell}}. \tag{8.8}$$

Starting with the output layer and using the chain rule,

$$\delta_j^L = \frac{\partial C}{\partial z_j^\ell} = \sum_i \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_j^L} = \sum_i \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_j^L}\delta_{ij} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L),$$

or, in both scalar and vector forms,

$$\begin{aligned}\delta_j^L &= \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)\\ \boldsymbol{\delta}^L &= \nabla_{\mathbf{a}} C \odot \sigma'(\mathbf{z}^L).\end{aligned} \tag{8.9}$$

For a quadratic cost function, for example, $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$ then $\partial C/\partial a_j^L = (a_j^L - y_j)$. Next, calculate the sensitivity for the ℓth layer in terms of the sensitivity and weights of the $(\ell+1)$th layer,

$$\begin{aligned}\delta_j^\ell &= \frac{\partial C}{\partial z_j^\ell}\\ &= \sum_i \frac{\partial C}{\partial z_i^{\ell+1}}\frac{\partial z_i^{\ell+1}}{\partial z_j^\ell}\\ &= \sum_i \frac{\partial C}{\partial z_i^{\ell+1}}\frac{\partial}{\partial z_j^\ell}\Big(\sum_k w_{ik}^{\ell+1} a_k^\ell + b_i^{\ell+1}\Big)\\ &= \sum_i \frac{\partial C}{\partial z_i^{\ell+1}}\frac{\partial}{\partial z_j^\ell}\Big(\sum_k w_{ik}^{\ell+1}\sigma(z_k^\ell) + b_i^{\ell+1}\Big)\\ &= \sum_i \frac{\partial C}{\partial z_i^{\ell+1}}\sigma'(z_j^\ell) w_{ij}^{\ell+1}\\ &= \sum_i \delta_i^{\ell+1}\sigma'(z_j^\ell) w_{ij}^{\ell+1}\end{aligned}$$

so that, in both scalar and vector forms,

$$\delta_j^\ell = \sum_i w_{ij}^{\ell+1}\delta_i^{\ell+1}\sigma'(z_j^\ell) \tag{8.10}$$

$$\boldsymbol{\delta}^\ell = (\mathsf{W}^{\ell+1})^T\boldsymbol{\delta}^{\ell+1} \odot \sigma'(\mathbf{z}^\ell). \tag{8.11}$$

Equations (8.9) and (8.11) allow us to calculate the sensitivities δ_j^ℓ for all layers. These can now be used to calculate the needed partial derivatives of the cost function. With respect to the biases this is given as,

$$\frac{\partial C}{\partial b_j^\ell} = \frac{\partial C}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial b_j^\ell} = \delta_j^\ell \tag{8.12}$$

while the cost derivative with respect to the weights is given by,

$$\frac{\partial C}{\partial w_{jk}^\ell} = \frac{\partial C}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial w_{jk}^\ell} = \delta_j^\ell a_k^{\ell-1}. \tag{8.13}$$

In summary, the back-propagation equations for calculating the cost derivative with respect to the weights and biases are,

$$\begin{aligned}
\boldsymbol{\delta}^L &= \nabla_{\mathbf{a}} C \odot \sigma'(\mathbf{z}^L) \\
\boldsymbol{\delta}^\ell &= ((\mathsf{W}^{\ell+1})^T \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell) \\
\frac{\partial C}{\partial b_j^\ell} &= \delta_j^\ell \\
\frac{\partial C}{\partial w_{jk}^\ell} &= a_k^{\ell-1} \delta_j^\ell .
\end{aligned}$$

The neural network learning algorithm is therefore,

1. Input a set of training data points and corresponding labels, $(\mathbf{x}, y)$.
2. For each training point, perform the following steps,
   - Evaluate the feed-forward network to calculate the label a^L.
   - Calculate the sensitivity, $\boldsymbol{\delta}^L$, for point $\mathbf{x}$,

$$\boldsymbol{\delta}^L = \nabla_{\mathbf{a}} C \odot \sigma'(\mathbf{z}^L)$$

   - Back-propagate the sensitivities – for each layer $\ell = L-1, L-2, ..., 2$ compute

$$\boldsymbol{\delta}^\ell = ((\mathsf{W}^{\ell+1})^T \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell)$$

3. Calculate the partial derivative of the cost with respect to the weights and biases; update the weights and the biases,

$$\begin{aligned}
w_{ij}^\ell &\to w_{ij}^\ell - \eta \frac{\partial C}{\partial w_{ij}^\ell} \\
b_i^\ell &\to b_i^\ell - \eta \frac{\partial C}{\partial b_i^\ell} .
\end{aligned}$$

8.4.4 Ways to improve neural networks

Beyond training the neural network via the back-propagation algorithm, we now explore the different ways to enhance the performance of a neural network to optimize for the trade off between efficiency and accuracy. We discuss several such improvements here:

Fine-tuning the architecture

Problems with neural networks may occur if too few or too many nodes are chosen. Consider a regression example first, where the problem is fitting data that are produced from the simple equation $y_i = 2X_i + \nu_i$ where ν_i is a random number (Fig. 8.10). The blue $\times$ marks show the training data. The red line shows the neural network evaluated at a higher resolution set of points X. Panels (a) of the figure show the results using a small network, with two hidden layers with 2 nodes each. The neural network is unable to fit all training data, yet this is not necessarily a problem, because these points contain noise and a perfect fit may not be needed. When making the network much larger (two hidden layers with 15 nodes each, panels (b) in Fig. 8.10), the neural network returns what are clearly unacceptable gaps between the training points, a clear sign of what we call "overfitting". The problem

occurs because we make the weights very large when many weights are available for the optimization so that we correctly fit the network to all training points. As a result, when these large weights are evaluated away from the training point they result in unreasonable values. The same problem occurs in trying to fit a polynomial to such data. A high-degree polynomial will provide a perfect fit to the training data, but an unreasonable behavior in between and as a result may not be useful in classifying new data near the training data. A smaller degree polynomial (such as a straight line or quadratic) will lead to an imperfect fit to the training data but a more stable overall performance – that is, no large fluctuations between training points. In the example, the average absolute value of weights and biases for the larger network are 2-4 times those of the small network.

One could note that there is a problem with the network by keeping some of the data for validation. In the example, consider the circles as these validation points. We evaluate the network using the training points as well as the validation points, by calculating the "performance" as the mean squared of the misfit between the training/validation labels and the labels calculated by the network. In the case of the small network (Fig. 8.10a) the fit to the validation points improves initially, but starts deteriorating after about 10 epochs. This indicates that even with this small network, there are not enough data points to well-constrain the weights. For the larger network (Fig. 8.10b), while the performance for the training points improves when training for additional epochs, the performance for the validation points does not improve, clearly demonstrating that there is a problem with the network.

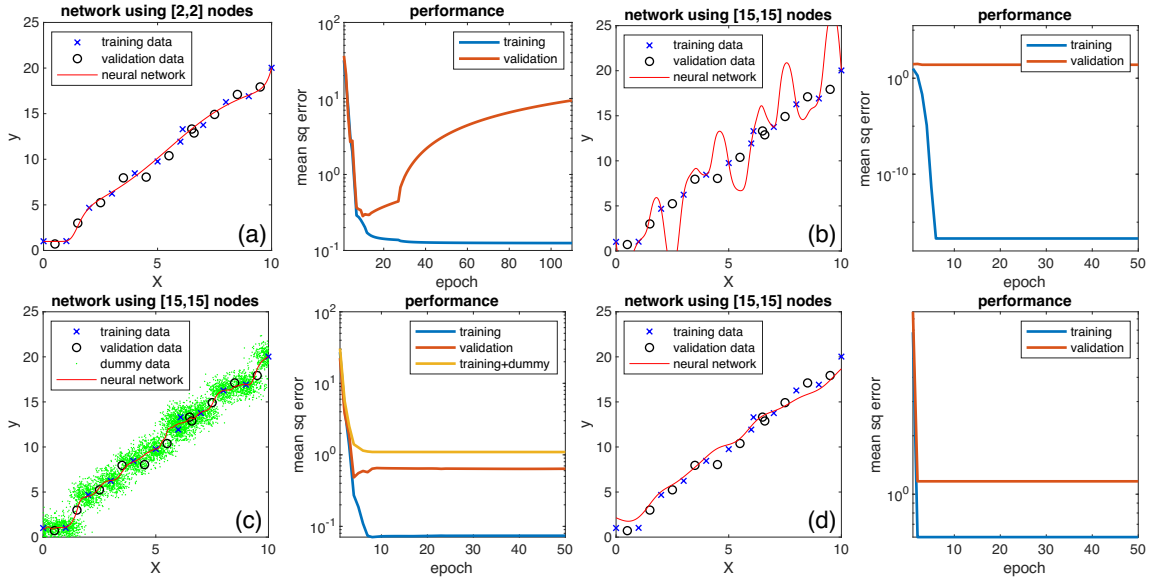


Figure 8.10: (a) A neural network with too few nodes. (b) Too many nodes, leading to over-fitting. (c) Adding dummy random data points to attempt to eliminate over-fitting. (d) Adding a regularization term to the cost function.

A rule of thumb for selecting the number of nodes is given by Haykin (2009) as $N = O(W/\varepsilon)$. Here N is the number of data points, W is the number of weights and biases to be calculated, and ε is the allowed misfit to the training data points. This relation allows us to guess how the required number of data points changes as we change the other two parameters. It does not tell us how many weights to use for a given problem, though, as it only provides an order-of-magnitude, or a scaling estimate, as indicated by the $O(\,)$. To reduce the effects of over-fitting, one can add dummy random data points by

adding noise to the original data points, as shown in Fig. 8.10c.

Regularization

Alternatively, one may "regularize" the problem by adding an extra term to the cost function, attempting to minimize the magnitude of the calculated weights,

$$C = \frac{1}{2}\sum_j (y_j - a_j^L)^2 + \lambda \sum_{i,j,\ell} (w_{ij}^\ell)^2$$

where λ is the *regularization parameter*. This again leads to more reasonable results (using $\lambda = 0.5$, Fig. 8.10d). A final alternative for preventing over-fitting to be considered is to stop the optimization early, so that the fit to the training data points is not as good, but the over-fitting issue is not as pronounced. We stop the optimization when the performance for the validation points stops improving and starts deteriorating, as seen around epoch number ten in Fig. 8.10a.

Similarly, in a classification problem, a network that it too small will not be able to fit all training labels (Fig. 8.11), and again one needs to decide if a perfect fit to the training data is needed, depending on the application context.

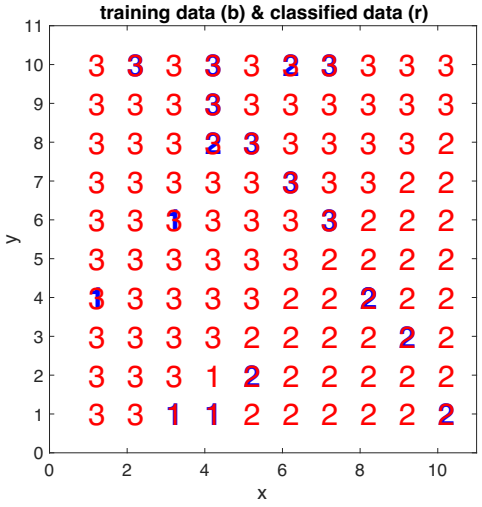


Figure 8.11: A classification neural network with too few nodes calculates labels (red) that cannot match all training data (blue).

8.5 k nearest neighbors (k-NN)

8.5.1 Classification of discrete labels

The k-nearest neighbor classification algorithm assigns a label to a given data point according to the point's proximity to known training data. The label may be discrete or continuous. In thinking about nearest-neighbor calculations we must be cognizant of three main considerations:

- The distance measure used to determine "proximity" to the nearest neighbors.
- The number, k, of nearest neighbors used to determine the label for the data point in question.
- The way the nearest neighbors are weighted: how should nearer neighbors be considered relative to ones that are further away?

In the case of discrete labels, given a choice for the number of neighbors to be considered, k, the label of the point to be classified can simply be chosen by a majority vote of these nearest neighbors.

■ **Example 8.5** Consider the training data in Fig. 8.12, and suppose we need to find an appropriate label for the point [5, 5.5]. The nearest neighbor training points are 15, 2, 9, 7 and 8, and the chosen label depends on the value of k as follows.
$k = 1$ nearest label is $+$, classification is "+"
$k = 2$ nearest labels are $++$, classification is "+"
$k = 3$ nearest labels are $++\times$, classification is "+"
$k = 4$ nearest labels are $++\times\circ$, dominant label and thus the classification is "+"
$k = 5$ nearest labels are $++\times\circ+$, dominant label and thus the classification is "+".

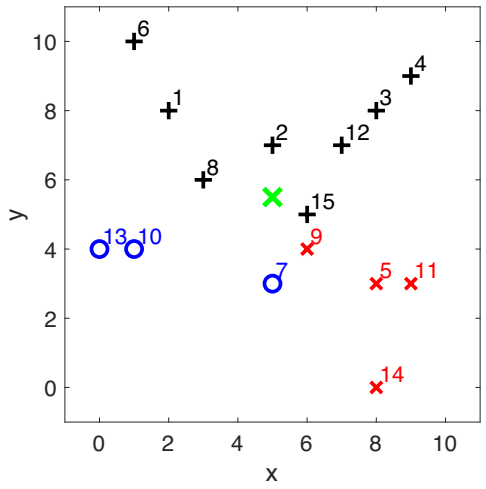


Figure 8.12: An example of k-NN with discrete label.

■

8.5.2 k-nn and Locally-weighted kernel regression

Next, consider the case of a non-discrete label. As an example, suppose we need to find the price $f(\mathbf{x})$ of a house that depends on a vector of parameters, e.g., $\mathbf{x} = (x_1, x_2)^T =$ (area, age), from k nearest neighbors of $\mathbf{x}$ for which we know the area and age $\mathbf{x}_n$, $n = 1, \ldots, k$ as well as the price $f(\mathbf{x}_n) = f_n$. One approach would be to use an average or weighted average of nearest neighbors. A more interesting one is to use a "local regression", where we write the quantity to be evaluated as a linear combination of the data coordinates (e.g., the house cost is a linear combination of the area and age),

$$f(\mathbf{x}) = \sum_{i=1}^{d} w_i x_i = \mathbf{w} \cdot \mathbf{x},$$

and find w_k using least squares fit to k nearest neighbors $\mathbf{x}_n$ by minimizing the weighted sum of squares for the training data.

Note that, as was the case for perceptrons, support vector machines and neural networks, if we would like the regression to include a free coefficient (bias, or non-zero threshold), we would add a row of ones to the data matrix such that each data vector is of the form $\hat{\mathbf{x}} = (x_1, x_2, 1)^T$, as well as

another element to $\mathbf{w}$ so it becomes of the form $\hat{\mathbf{w}} = (w_1, \ldots, w_d, w_{d+1})^T$ and of dimension $d+1$. The regression is of the form,

$$f(\mathbf{x}) = \sum_{i=1}^{d} w_i x_i + w_{d+1} = \mathbf{w}' \cdot \mathbf{x}',$$

and includes a free term as desired, while the solution procedure is identical for the cases with and without a free term, and is described as follows.

Define a coordinate matrix $\mathsf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$ such that the coordinates $\mathbf{x}_n$ of the nth nearest neighbor are given by the nth column of X. The function to be minimized in order to find the regression coefficients may then be written as,

$$\begin{aligned} J(\mathbf{w}) &= \sum_{n=1}^{k} \mathsf{K}(d(\mathbf{x}_n, \mathbf{x}))(f(\mathbf{x}_n) - \mathbf{w} \cdot \mathbf{x}_n)^2 \\ &= \sum_{n=1}^{k} K_{nn} \left(f_n - \sum_{i=1}^{N} w_i x_{in} \right)^2 \end{aligned}$$

where N is the dimension of $\mathbf{w}$ and $\mathbf{x}_n$ (or of $\mathbf{w}'$ and $\mathbf{x}'_n$ if a non-zero bias is needed), k is the number of near neighbors to be used, $d(\mathbf{x}_n, \mathbf{x})$ is the distance between the nth nearest neighbor training data point and the point being classified, $f_n = f(\mathbf{x}_n)$ is the known label (house price) of the nth near neighbor data point, K is a "kernel function" that weights the training data based on their distance from the point that is being estimated, $\mathbf{x}$. Two such examples of a kernal, K are a constant K and a K that decays as a Gaussian of the distance. K_{nn} is a diagonal matrix containing the distance-weights. At the minimum of $J(\mathbf{w})$, its derivative with respect to w_j is equal to zero, giving,

$$0 = \frac{dJ}{dw_j} = \sum_{n=1}^{k} 2 x_{jn} K_{nn} \left(f_n - \sum_{i=1}^{N} w_i x_{in} \right).$$

We can write this in matrix form as,

$$\mathsf{X}\mathsf{K}\mathsf{X}^T \mathbf{w} = \mathsf{X}\mathsf{K}\mathbf{f}$$

and the solution is,

$$\mathbf{w} = (\mathsf{X}\mathsf{K}\mathsf{X}^T)^{-1} (\mathsf{X}\mathsf{K})\mathbf{f}.$$

Note the dimensions of all matrices and vectors here,

$$\mathbf{w}_{N\times 1} = \left(\left(\mathsf{X}_{N\times k} \mathsf{K}_{k\times k} \mathsf{X}^T_{k\times N} \right)^{-1}_{N\times N} \left(\mathsf{X}_{N\times k} \mathsf{K}_{k\times k} \right)_{N\times k} \right)_{N\times k} \mathbf{f}_{k\times 1}.$$

We are assuming that there are more neighbors than weights to be calculated, $k > N$, so that the above matrix inverse exists. This is essentially an over-determined problem, where the only main difference from previous over-determined systems being the weight matrix, K. The predicted price for coordinates (house age and area $\mathbf{x}$) is now given by $\mathbf{w}^T\mathbf{x}$. Note that while the price may be a non-linear function of the (age, area) parameters, the regression provides a linear local approximation.

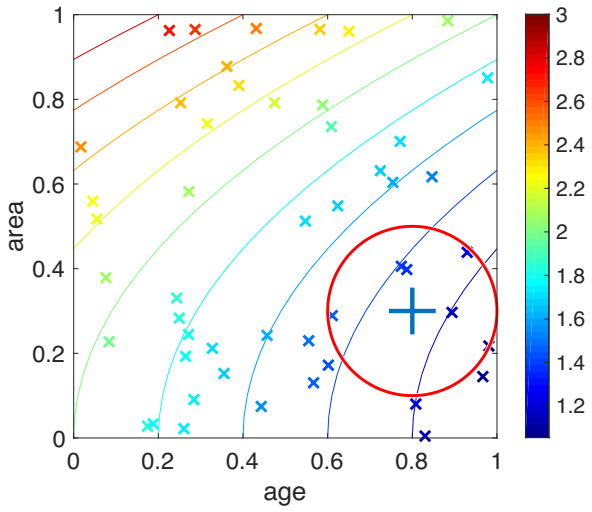
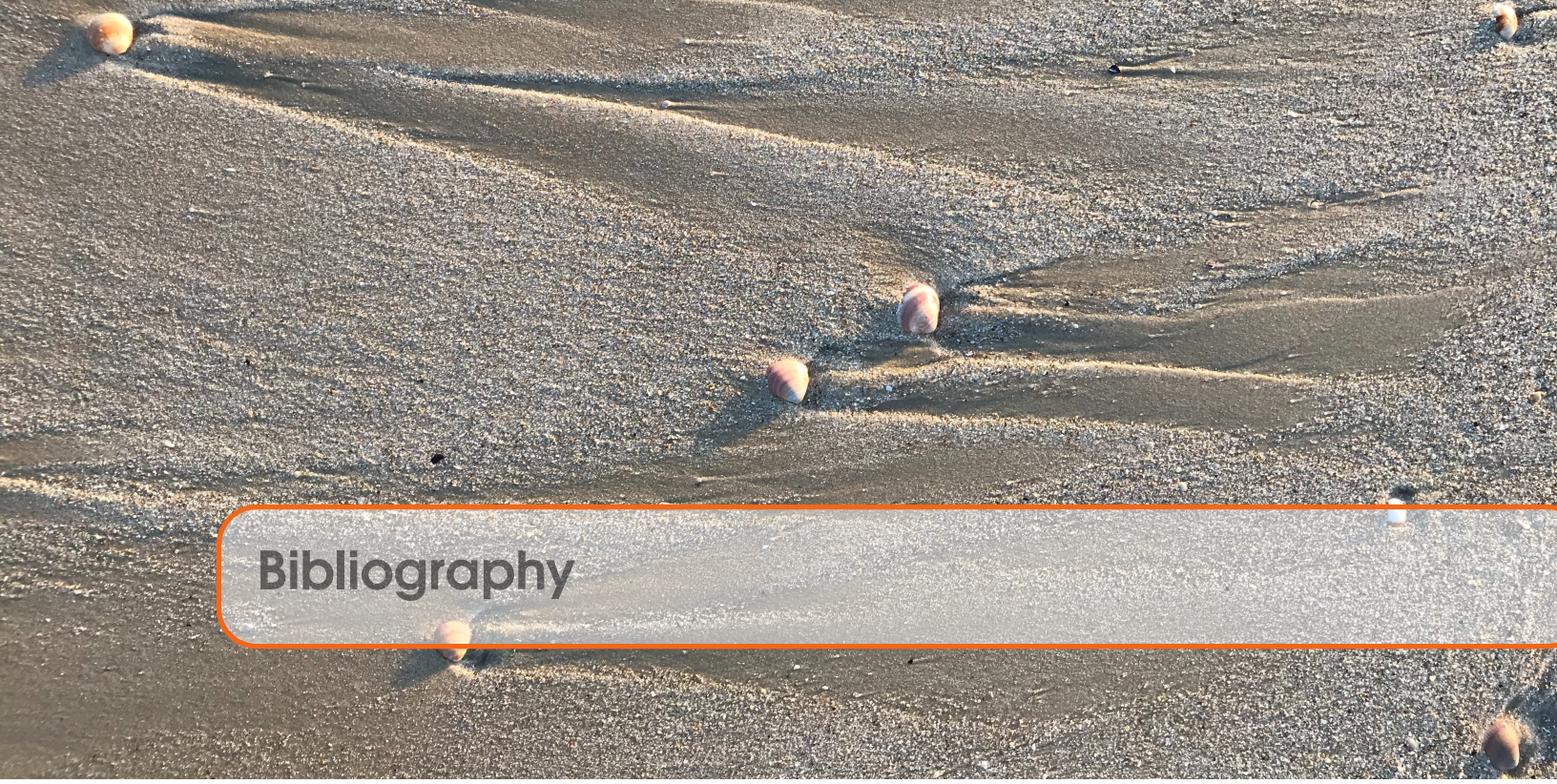


Figure 8.13: An example for house prices as function of age and area. Houses previously sold (the training data) are indicated by $\times$ symbols while the house whose price is to be estimated is indicated by the $+$. The contours are the actual cost as function of these two parameters prescribed for this example.



Bibliography

Agarwal, R., Imielinski, T., and Swami, A. (1993). Mining associations between sets of items in massive databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 207–216.

Agarwal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *Intl. Conf. on Very Large Databases*, pages 487–499.

Andersson, E. and Ekstrom, P.-A. (2004). Investigating google's pagerank algorithm. Student report in scientific computing.

Blum, A., Hopcroft, J., and Kannan, R. (2017). *Foundations of Data Science*.

Bradley, P. S., Fayyad, U., and Reina, C. (1998). Scaling clustering algorithms to large databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, pages 9–15. AAAI Press.

Compo, G. P. and Sardeshmukh, P. D. (2010). Removing ENSO-related variations from the climate record. *J. Climate*, 23(8):1957–1978.

Farrell, B. F. and Ioannou, P. J. (1996). Generalized stability theory part I: autonomous operators. *J. Atmos. Sci.*, 53:2025–2040.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Guha, R., Rastogi, R., and Shim, K. (1998). Cure: An efficient clustering algorithm for large databases. Proc. ACM SIGMOD Intl. Conf. on Management of Data.

Haykin, S. (2009). *Neural Networks and Learning Machines*. Pearson Prentice Hall.

Jacobson, L. (2013). Introduction to artificial neural networks. http://www.theprojectspot.com/.

Johnson, N. C., Feldstein, S. B., and Tremblay, B. (2008). The continuum of northern hemisphere teleconnection patterns and a description of the nao shift with the use of self-organizing maps. *J. Climate*, 21(23):6354–6371.

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Leskovec, J., Rajaraman, A., and Ullman, J. D. (2014). *Mining of massive datasets*. Cambridge University Press.

Mitchell, T. (1997). *Machine learning*. McGraw-Hill Science/ Engineering/ Math.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Peixoto, J. P. and Oort, A. H. (1992). *Physics of Climate*. American Institute of Physics.

Strang, G. (2006). *Linear algebra and its applications. 4th ed*. Cengage Learning.

Strogatz, S. (1994). *Nonlinear dynamics and chaos*. Westview Press.

Trefethen, L. N. and Bau III, D. (1997). *Numerical linear algebra*, volume 50. Siam.

Von-Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416.

Vozalis, M. G. and Margaritis, K. G. (2006). Applying SVD on generalized item-based filtering. *IJCSA*, 3(3):27–51.

Index

D

E

F

G

H

I

J

K

L

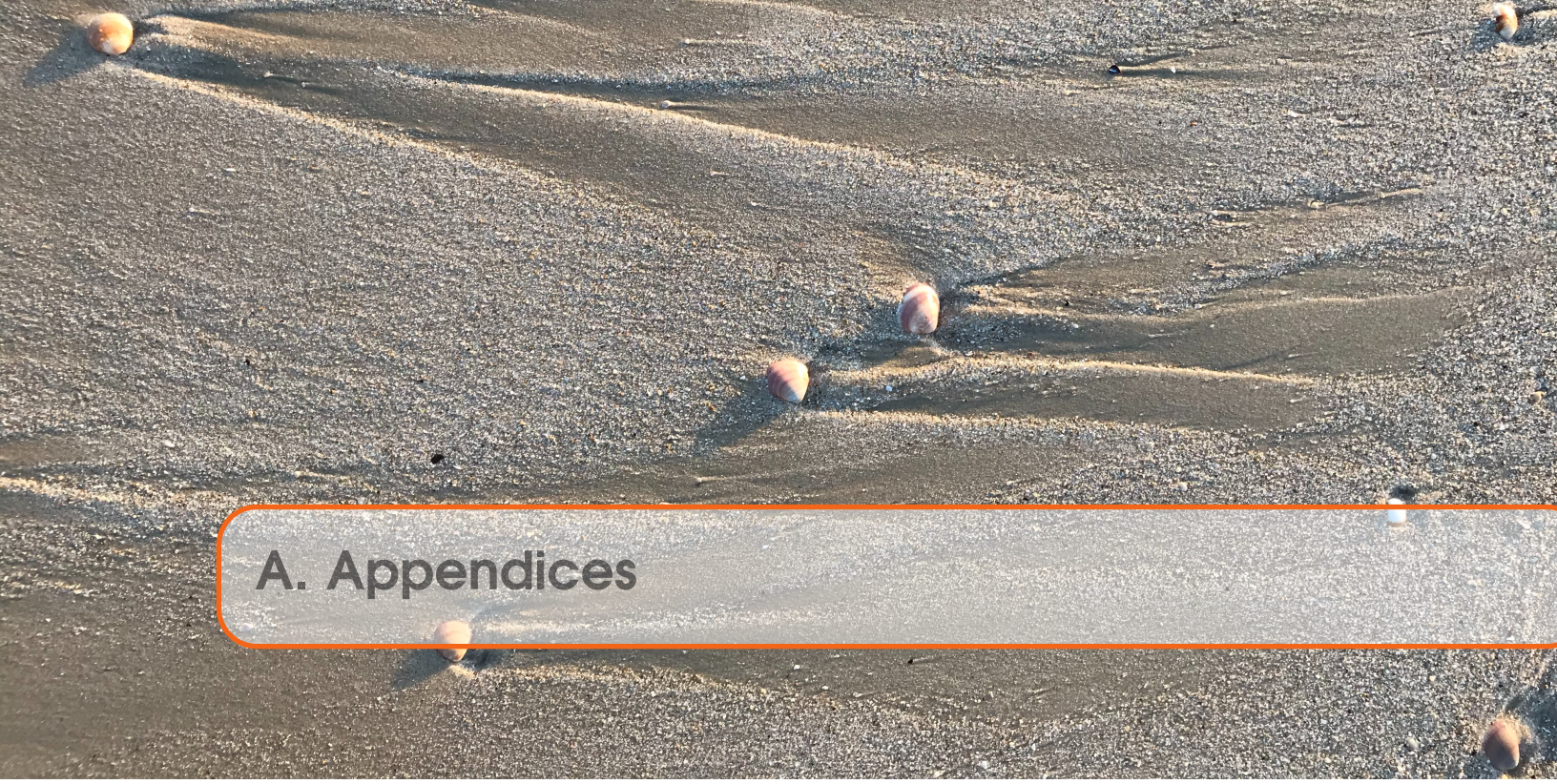
M

N

O

W

X



A. Appendices

A.1 Proof that eigenvectors are orthogonal $\iff$ the matrix is normal

Consider the following proof that the eigenvectors of a matrix are orthogonal if and only if the matrix is normal, that is, $\mathsf{A}^T\mathsf{A} = \mathsf{A}\mathsf{A}^T$, see http://fourier.eng.hmc.edu/e161/lectures/algebra/node8.html.

Theorem: The product of two unitary matrices is unitary.

Proof: Let U and V be unitary, i.e., $\mathsf{U}^* = \mathsf{U}^{-1}$ and $\mathsf{V}^* = \mathsf{V}^{-1}$, then UV is unitary:

$$(\mathsf{UV})^* = \mathsf{V}^*\mathsf{U}^* = \mathsf{V}^{-1}\mathsf{U}^{-1} = (\mathsf{UV})^{-1}$$

Theorem: Two square matrices A and B are simultaneously diagonalizable if and only if they commute.

Proof: Let A and B be simultaneously diagonalizable by R (this means that the two have the same eigenvectors that are in R),

$$\mathsf{R}^{-1}\mathsf{A}\mathsf{R} = \Lambda_A, \qquad \mathsf{R}^{-1}\mathsf{B}\mathsf{R} = \Lambda_B$$

then

$$\mathsf{AB} = (\mathsf{R}\Lambda_A\mathsf{R}^{-1})(\mathsf{R}\Lambda_B\mathsf{R}^{-1}) = (\mathsf{R}\Lambda_A\Lambda_B\mathsf{R}^{-1}) = (\mathsf{R}\Lambda_B\Lambda_A\mathsf{R}^{-1}) = (\mathsf{R}\Lambda_B\mathsf{R}^{-1})(\mathsf{R}\Lambda_A\mathsf{R}^{-1}) = \mathsf{BA}$$

Let A and B commute, i.e., $\mathsf{AB} = \mathsf{BA}$. Assuming $\mathbf{u}$ is an eigenvector of A corresponding to eigenvalue λ, i.e., $\mathsf{A}\mathbf{u} = \lambda\mathbf{u}$, then

$$\mathsf{AB}\mathbf{u} = \mathsf{B}(\mathsf{A}\mathbf{u}) = \lambda\mathsf{B}\mathbf{u}.$$

We see that $\mathsf{B}\mathbf{u}$ is also an eigenvector of A corresponding to the same eigenvalue λ, i.e., $\mathsf{B}\mathbf{u}$ must be a scaled version of $\mathbf{u}$ (in the same 1-D space): $\mathsf{B}\mathbf{u} = \gamma\mathbf{u}$, i.e., $\mathbf{u}$ is also an eigenvector of B.

Theorem: A matrix is normal if and only if it is unitarily diagonalizable.

Proof: If A is unitarily diagonalizable,

$$\mathsf{AU} = \mathsf{U}\Lambda, \qquad \mathsf{U}^{-1}\mathsf{AU} = \mathsf{U}^*\mathsf{AU} = \mathsf{A}, \qquad \mathsf{A} = \mathsf{U}\Lambda\mathsf{U}^*$$

where $\mathsf{U}^* = \mathsf{U}^{-1}$ is unitary and Λ is a diagonal matrix satisfying $\Lambda^*\Lambda = \Lambda\Lambda^*$, then A is normal:

$$\begin{aligned}\mathsf{A}\mathsf{A}^* &= (\mathsf{U}\Lambda\mathsf{U}^*)(\mathsf{U}\Lambda\mathsf{U}^*)^* = (\mathsf{U}\Lambda\mathsf{U}^*)(\mathsf{U}\Lambda^*\mathsf{U}^*) \\ &= \mathsf{U}\Lambda\Lambda^*\mathsf{U}^* = \mathsf{U}\Lambda^*\Lambda\mathsf{U}^* = (\mathsf{U}\Lambda^*\mathsf{U}^*)(\mathsf{U}\Lambda\mathsf{U}^*) = \mathsf{A}^*\mathsf{A}\end{aligned}$$

Note that "diagonalizable by a unitary matrix", implies immediately that the matrix has orthonormal eigenvectors and vice versa.

If A is normal, then it is diagonalizable by a unitary matrix. First, we show any matrix A can be written as

$$\mathsf{A} = \mathsf{B} + i\mathsf{C}$$

where

$$\mathsf{B} = \frac{1}{2}(\mathsf{A} + \mathsf{A}^*) = \mathsf{B}^*, \quad \mathsf{C} = -\frac{1}{2}i(\mathsf{A} - \mathsf{A}^*) = \mathsf{C}^*,$$

are both Hermitian, and diagonalizable by a unitary matrix. As A is normal, we have

$$0 = \mathsf{A}\mathsf{A}^* - \mathsf{A}^*\mathsf{A} = (\mathsf{B} + i\mathsf{C})(\mathsf{B} - i\mathsf{C}) - (\mathsf{B} - i\mathsf{C})(\mathsf{B} + i\mathsf{C}) = 2i(\mathsf{C}\mathsf{B} - \mathsf{B}\mathsf{C})$$

We see that $\mathsf{C}\mathsf{B} = \mathsf{B}\mathsf{C}$, i.e., B and C commute, and they can be simultaneously diagonalized by some unitary matrix U:

$$\mathsf{U}^*\mathsf{B}\mathsf{U} = \Lambda_B, \qquad \mathsf{U}^*\mathsf{C}\mathsf{U} = \Lambda_C,$$

and so can $\mathsf{A} = \mathsf{B} + i\mathsf{C}$:

$$\mathsf{U}^*\mathsf{A}\mathsf{U} = \mathsf{U}^*(\mathsf{B} + i\mathsf{C})\mathsf{U} = \Lambda_B + i\Lambda_C = \Lambda_A$$

A.2 Proof that Frobenius norm is equal to sum of singular values squared

Consider one row of A, $\mathbf{a}_j$. Because the $\mathbf{v}_i$ vectors are orthogonal and normalized, we can use them to expand $\mathbf{a}_j = \sum_{i=1}^{N}(\mathbf{a}_j \cdot \mathbf{v}_i)\mathbf{v}_i$. The squared magnitude of a_j, which is the sum of squares of the elements of a_j, is therefore,

$$|\mathbf{a}_j|^2 = \sum_{i=1}^{N}(\mathbf{a}_j \cdot \mathbf{v}_i)\mathbf{v}_i \sum_{k=1}^{N}(\mathbf{a}_j \cdot \mathbf{v}_k)\mathbf{v}_k = \sum_{i=1}^{N}(\mathbf{a}_j \cdot \mathbf{v}_i)^2$$

where in the last equality we used the fact that the $\mathbf{v}_i$ vectors are orthonormal. According to this, the Frobenius norm is,

$$\begin{aligned}\text{norm}_F(\mathsf{A}) = \|\mathsf{A}\|_F &= \sum_{i,j} a_{ij}^2 = \sum_{j=1}^{M}|\mathbf{a}_j|^2 \\ &= \sum_{j=1}^{M}\sum_{i=1}^{N}(\mathbf{a}_j \cdot \mathbf{v}_i)^2 = \sum_{i=1}^{N}(A\mathbf{v}_i)^2 \\ &= \sum_{i=1}^{N}(A\mathbf{v}_i)^T(A\mathbf{v}_i) = \sum_{i=1}^{N}\mathbf{v}_i^T(\mathsf{A}^T\mathsf{A})\mathbf{v}_i = \sum_{i=1}^{N}\mathbf{v}_i^T\sigma_i^2\mathbf{v}_i \\ &= \sum_{i=1}^{N}\sigma_i^2.\end{aligned}$$