Math 5467: Introduction to the Mathematics of Image and Data Analysis

Jeff Calder

University of Minnesota
School of Mathematics
jwcalder@umn.edu

February 2, 2021

Contents

1 Introduction 3
  1.1 Course Information . . . . . . . . . . . . . . . . . . . . . . . 3
  1.2 Python . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.3 Background on audio, images, and data analysis . . . . . . . 5
    1.3.1 Audio . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    1.3.2 Images . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    1.3.3 Data science and machine learning . . . . . . . . . . . 9

2 Linear Algebra Review 10
  2.1 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.2 Projection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.3 Diagonalization of symmetric matrices . . . . . . . . . . . . . 14
  2.4 Vector calculus . . . . . . . . . . . . . . . . . . . . . . . . . 15

3 Principal Component Analysis 16
  3.1 Fitting the best linear subspace . . . . . . . . . . . . . . . . . 17
  3.2 How many principal directions? . . . . . . . . . . . . . . . . . 22
  3.3 Robust PCA . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  3.4 PCA-based Image Compression . . . . . . . . . . . . . . . . . 26
  3.5 PCA-based Handwritten Digit Recognition . . . . . . . . . . . 32
1 Introduction

1.1 Course Information

This course is a modern introduction to the mathematics of image and data analysis. The course will cover the discrete Fourier and Wavelet transforms, with applications to image and audio processing. We will also cover the mathematics of common data analysis algorithms, including principal component analysis (PCA), data ranking (e.g., Google’s PageRank for ranking webpages), and clustering algorithms such as k-means and spectral clustering. Time-permitting, we will give an introduction to machine learning (ML), and cover basic ML classifiers, neural networks (in particular, convolutional neural networks for image classification), and graph-based learning.

The course will cover both mathematical theory and practical applications. We will use Python for all computational work in this course. Students will get hands on experience working with real data through a series of computational projects that will be completed throughout the term, on topics such as audio or image compression, facial recognition, or image classification. We will start the course with a gentle introduction to Python; no prior knowledge is required. See the course website, below, for details on how to get access to Python:

http://www-users.math.umn.edu/~jwcalder/5467S21/index.html

1.2 Python

These lecture notes are accompanied by Python notebooks that can be found on the course website, and in links throughout these notes. Below are links to Google Colab Python notebooks with basic introductions to various aspects of Python programming.

Introduction to Python Notebooks:

1. Introduction to Python
2. Introduction to Numpy
3. Reading and writing images and audio in Python
4. Introduction to Pandas
Below are some exercises to be completed in Python (most are in the notebooks above as well).

**Exercise 1.1.** Write a Python function that approximates \( \sin(x) \) using the Taylor expansion \( \sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \). Test your function for simple known values of \( \sin(x) \), such as \( \sin(0) = 0 \), \( \sin(\pi/4) = \frac{1}{\sqrt{2}} \), \( \sin(\pi/2) = 1 \), and \( \sin(\pi) = 0 \), etc. How good is the approximation?

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

**Exercise 1.2.** Write a Python function that computes the square root of a positive number using the Babylonian method. The Babylonian method to compute \( \sqrt{S} \) for \( S > 0 \) constructs the sequence \( x_n \) by setting \( x_0 = S \) and iterating

\[
x_{n+1} = \frac{1}{2} \left( x_n + \frac{S}{x_n} \right).
\]

Your code can take as input a tolerance parameter \( \varepsilon > 0 \), and should iterate until \( |x_n^2 - S| \leq \varepsilon \), and then return \( x_n \). Test your square root function to make sure it works.

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

**Exercise 1.3.** Write a Python program that uses the Sieve of Eratosthenes to find all prime numbers between 2 and a given integer \( n \).

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

**Exercise 1.4.** Write a Python function that computes the largest magnitude eigenvalue of a square matrix with the power iteration. The power iteration is

\[
x_{n+1} = \frac{Ax_n}{\|Ax_n\|}.
\]

For a diagonalizable matrix, the iteration converges to the eigenvector of \( A \) with largest magnitude eigenvalue. The eigenvalue is

\[
\lambda = \lim_{n \to \infty} \|Ax_n\|.
\]

Compare your function to the true eigenvector and eigenvalue for small matrices where you can compute it by hand, to check that your function works.

In this exercise you may use Numpy. Try to write your code with only one loop, over the iterations in the power method.
Exercise 1.5. Write a Python function that numerically approximates $\pi$ via the integral expression

$$
\pi = 4 \int_0^1 \sqrt{1 - x^2} \, dx.
$$

Your function should not use any loops. Use a Numpy array and Numpy functions instead. The code cell below will get you started, you only have to replace the ?? by your formula for numerical integration. Then you can change $dx$ to be as small as you like to get higher accuracy. How many decimal places of $\pi$ can you accurately compute?

In this exercise you may use Numpy. Can you write the code without any loops? \(\triangle\)

The following exercise may be of interest, but does not involve Python.

Exercise 1.6. Prove that the iteration in the Babylonian method above converges quadratically to the square root of $x$. In particular, show that the error $\epsilon_n = \frac{x_n}{\sqrt{x}} - 1$ satisfies

$$
\epsilon_{n+1} = \frac{\epsilon_n^2}{2(\epsilon_n + 1)}.
$$

From this, we get that $\epsilon_n \geq 0$ for $n \geq 1$, and so

$$
\epsilon_{n+1} \leq \frac{1}{2} \min\{\epsilon_n^2, \epsilon_n\}.
$$

Why does the inequality above guarantee convergence (i.e., that $\epsilon_n \to 0$ as $n \to \infty$)? \(\triangle\)

1.3 Background on audio, images, and data analysis

A major theme in this course is that of finding a good basis to represent your data. Usually data (e.g., images, audio, video, etc.) are captured in formats that are convenient for acquiring the data, but very inconvenient for storage or other kinds of processing. Much of this course is concerned with finding new bases for image and audio data that are better for tasks like image compression or image classification, among others. A change of basis can be linear, which the reader should be familiar with from linear algebra, or nonlinear as in modern deep neural networks. The change of basis can also be either hand-crafted (like in Fourier or Wavelet analysis), or learned from the data, as in Principal Component Analysis (PCA) or deep learning. Before we proceed with the course, we review the formats in which images and audio are typically recorded and stored.
1.3.1 Audio

A digital audio signal is a series of discrete samples of a continuous audio signal generated by the movement of the vibrating membrane of a microphone (of course, the vibrations in the membrane are caused by the rarefaction and compression of air, e.g., sound waves). Audio signals can be captured at different sampling rates and bit-depths. Standard CD quality audio has two channels (left and right) sampled at 44,100 Hz with 16-bits per sample. This means each channel has 44,100 samples per second, each encoded as a (signed) 16-bit integer to represent the magnitude of the sound wave at that sample. By the Nyquist-Shannon sampling theorem, this sampling rate allows CD audio to represent frequencies up to 22,050 Hz, which is well above what most people can hear. Figure 1.1 shows stereo audio signals for a piece of classical music.

The bit-rate for CD quality audio is the number of bits used per second of audio, which we can compute as

\[
\text{Bit-rate} = \left( \frac{2 \text{ Channels}}{1 \text{ Second}} \right) \times 44,100 \times 16 = 1,411,200 \text{ bits/second}.
\]

Often this is written in terms of kilobits (kbit)—a kilobit is 1000 bits—so CD audio has a bitrate of 1,411 kbit/s (also denoted 1,411 kbps). A 4 minute long song would thus take

\[
4 \times \frac{60}{\text{Second}} \times 1,411 \text{ kbits/second} = 338,640 \text{ kbits},
\]

of space on disk. In terms of megabits (Mbit)—a megabit is 1000 kilobits—this is 339 Mbits. The reader may be more familiar with kilobytes (kB) and
megabytes (MB). A byte is 8 bits, and so 339 Mbits is actually around 42 MB.\footnote{Note we are using base 10 to define kilo and mega. It is also common to use base 2, so that a kilobit is 2^10 = 1024 bits, and a megabit is 1024 kilobits.}

The reader who is familiar with digital audio files will probably know that an mp3 audio file for a 4 minute song is usually around 5 MB, depending on the bitrate, which is much less than the 42 MB we computed for the raw data. In fact, a very high quality mp3 bitrate is 320 kbit/second\footnote{It is widely accepted, except by some audio enthusiasts, that 320 kbit/second mp3 audio is indistinguishable to the human ear from CD quality audio.}, which is less than one quarter of the bitrate of CD quality audio. Many mp3 audio files are compressed at much lower bitrates than this, with sometimes acceptable results. A main question we will address in this course is how to compress audio signals (and later images and video), without destroying important information. Other important tasks in audio processing are classifying audio signals (e.g., determining which song is playing automatically), and speech to text (e.g., determining what was said in an audio sample).

\section*{1.3.2 Images}

A digital image is a discrete sampling of a two dimensional continuous signal given by the light hitting a rectangular image sensor in a digital camera. Thus, we can think of an image as a two-dimensional array of pixels. An image can be grayscale, in which case each pixel has a single number associated with it, representing the brightness, or image intensity, at that location. Figure 1.2 shows an example of a grayscale image, the famous cameraman image. On the right in the figure, we show the image plotted as a function (e.g., a surface), over the two-dimensional plane, where the height of the surface is the intensity of the pixel at that location. In a color image, each pixel is associated with several numbers, representing the intensity of different colors at that point, which mix to produce the correct color. The most common colorspace is RGB (red–green–blue), in which each pixel has three numbers representing the amount of red, green, and blue present at each pixel in the image. Thus, a grayscale image can be thought of as a matrix, while a color image can be thought of as 3 (or possibly more) matrices, one for each color channel. There are other colorspaces that are frequently used (e.g., by printers) that have more color channels, or use different colors. For example, in hyperspectral satellite imagery, it is not uncommon to have over 200 color channels in an image, each corresponding to a different frequency band of the electromagnetic spectrum.
Figure 1.2: Example of a grayscale digital image. The left shows the image in its usual form, while the right side depicts the image intensity as a surface (i.e., a function of 2 variables).

The cameraman image is of size $512 \times 512$, and so it has 262,144 pixels. This is quite small by today’s standards. For example, a high-end modern smartphone has a camera with 12 million pixels (12 MP). For a color image, this requires storing 36 million numbers. Images usually have a bit depth of 8-bit, allowing $2^8 = 256$ different values for each color channel, or about 16 million different colors. Some newer smartphones use higher bit depths of 10-bits or 12-bits, yielding over 1 billion different colors. Let’s consider an 8-bit color image with 12 megapixels. The 36 million 8-bit numbers that are stored take up 36 MB of space. A 10-bit image would take 45 MB of space and a 12-bit image would take 54 MB of space. High end digital cameras can have even higher resolution, currently up to around 46 million pixels, yielding 8-bit images that take around 130 MB of space, and 14-bit images that take up to 240 MB.

However, images are rarely stored or transmitted in raw form, since it is a colossal waste of space. Common image compression algorithms, like jpeg, can compress images to roughly one tenth of their size, without noticeable loss in image quality. Building on our discussion of audio compression, we will learn in this course how to compress images without removing important information. Aside from image compression, there are many other important tasks in computer vision and image processing, such as image segmentation (determin-
ing the object of interest in an image), object recognition (determining what the object is), image restoration (removing noise or blur), image inpainting (recovering lost portions of images), and image classification. We will touch on some of these other applications in the course.

A few words about video compression. We will not touch on this in the course, but since video is simply a sequence of images, one could apply image compression in this setting as well. However, there is a lot of temporal information that is missed by such an approach, and modern video compression uses motion tracking and compresses only the differences between subsequent frames of video, which has significant advantages over image compression alone.

1.3.3 Data science and machine learning

Finally, this course is not only about image and audio processing. Part of the course will cover some fundamentals of data science and machine learning from a mathematical perspective. In data analysis, one typically has many datapoints, and the goal is to uncover structure in the data to perform tasks like classification, clustering, dimension reduction, etc. Many problems in machine learning are related to image analysis (e.g., image classification), but are substantially different in character than the image and audio compression problems described in the preceding sections. In machine learning, we are given many (possibly thousands or millions) of images and the task is to understand how the images naturally group together (e.g., clustering), and how to automatically distinguish images from different classes (e.g., image classification).

Figure 1.3 shows an example of some images from the MNIST dataset, which we will use for illustrating machine learning throughout the course. The MNIST dataset contains 70,000 grayscale images of handwritten digits 0 through 9. Each image is very small, containing only $28 \times 28$ pixels. We can still consider the image compression problem, but here the goal would be to compress the entire dataset, and not a single image. A more common task is to train a machine learning classifier to recognize the number written in each image. This is called optical character recognition, and is very commonly used for many tasks, including archiving old newspapers or books, and teaching self-driving cars to read street signs and house numbers. We will cover various methods for image classification in the course, starting with a method based on principal component analysis (PCA), and then moving on to more sophisticated methods based on Fourier and Wavelet features, and deep convolutional neural networks.
2 Linear Algebra Review

We recall some basic facts about linear algebra in this section.

2.1 Notation

We will use capital letters like $A, B, C, \ldots$ for matrices, and lower case letters, such as $x, y, z, \ldots$ for vectors. Scalars will be denoted by lower case letters $a, b, c, \text{ or } a_1, a_2, \ldots$. When not specified, vectors are of length $n$ and matrices of size $m \times n$. Vectors are always treated as column vectors. If we have more than a handful of vectors, we will use subscripts to denote a collection of vectors, so $x_1, x_2, \ldots, x_p$ will always refer to a collection of $p$ vectors. For example, $e_1, e_2, \ldots, e_n$ will denote the $n$ standard basis vectors in $\mathbb{R}^n$. The reader should be careful not to confuse $x_i$ with the coordinates of the vector $x$. We will rarely need to notate the coordinates of vectors or matrices, and if we do, we will use the notation $x(i)$ for the $i^{\text{th}}$ coordinate of $x$, or $e_i^T x$. Similarly, we will denote the entries of a matrix $A$ as $A(i,j)$.

The dot product of vectors $x$ and $y$ is given by the product $x^T y$, treating $x$ and $y$ as $n \times 1$ column vectors. To write this out in coordinate notation we
have

\[ x^T y = \sum_{i=1}^{n} x(i) y(i). \]

The dot product induces a norm

\[ \|x\| = \sqrt{x^T x} = \sqrt{x(1)^2 + x(2)^2 + \cdots + x(n)^2} \]

on the Euclidean space \( \mathbb{R}^n \). The quantity \( \|x - y\| \) is exactly the Euclidean distance between \( x \) and \( y \). A commonly used algebraic expansion is

\[ \|x \pm y\|^2 = \|x\|^2 \pm 2x^T y + \|y\|^2. \]

This can be verified immediately by writing \( \|x \pm y\|^2 = (x \pm y)^T (x \pm y) \) and expanding.

We say two vectors \( x \) and \( y \) are \textit{orthogonal} if \( x^T y = 0 \). A vector \( x \) is a \textit{normal} vector if \( \|x\| = 1 \), and two orthogonal vectors \( x \) and \( y \) that are also normal vectors are called \textit{orthonormal}. A sequence of vectors \( v_1, v_2, \ldots, v_p \) is called orthogonal (resp. orthonormal) if each pair \( v_i, v_j \) is orthogonal (resp. orthonormal), provided \( i \neq j \). Consider a vectors \( x \) in the span of orthonormal vectors \( v_1, v_2, \ldots, v_p \), that is

\[ x = \sum_{i=1}^{p} a_i v_i. \]

Then the norm of \( x \) can be computed by

\[ \|x\|^2 = \sum_{i=1}^{p} a_i^2 v_i^T \sum_{j=1}^{p} a_j v_j = \sum_{i=1}^{p} \sum_{j=1}^{p} a_i a_j v_i v_j^T = \sum_{i=1}^{p} a_i^2. \]

**Exercise 2.1.** Given vectors \( x \) and \( y \), both of length \( n \), we will sometimes find it useful to construct the rank-one matrix \( xy^T \). The matrix is called “rank-one” since the range of \( xy^T \) is spanned by the vector \( x \), and is thus one-dimensional so the matrix has rank equal to one. Using the definition of matrix multiplication, \( xy^T \) is the \( n \times n \) matrix whose \((i, j)\) entry is \( x(i)y(j) \). Let \( x_1, x_2, x_3, \ldots, x_m \) be a collection of vectors of length \( n \). Show that

\[ \sum_{i=1}^{m} x_i x_i^T = X^T X, \]

where \( X \) is the \( m \times n \) matrix whose \( i^{th} \) row is \( x_i^T \), which can be written as

\[ X = [x_1 \ x_2 \ \cdots \ x_m]^T. \]

\[ \triangle \]
2.2 Projection

Let \( L \subset \mathbb{R}^n \) be a linear subspace spanned by the orthonormal vectors \( v_1, v_2, \ldots, v_p \), where \( p \leq n \). That is
\[
L = \left\{ \sum_{i=1}^{p} a_i v_i : a_i \in \mathbb{R} \right\}.
\]

In this case, \( L \) is \( p \)-dimensional. The projection of a point \( x \in \mathbb{R}^n \) onto \( L \), denoted \( \text{Proj}_L x \), is the closest point in the subspace \( L \) to \( x \). That is, \( \text{Proj}_L x \in L \) satisfies
\[
\|\text{Proj}_L x - x\| \leq \|y - x\| \text{ for all } y \in L.
\]

We recall here some basic properties of the projection. First, we claim that
\[
\text{(2.2)} \quad \text{Proj}_L x = \sum_{i=1}^{p} (x^T v_i) v_i
\]

To see this, let us write \( y \in L \) as
\[
y = \sum_{i=1}^{p} a_i v_i,
\]
and compute
\[
\|x - y\|^2 = \|x\|^2 - 2 x^T y + \|y\|^2 = \|x\|^2 - 2 \sum_{i=1}^{p} a_i x^T v_i + \sum_{i=1}^{p} a_i^2.
\]

Minimizing over \( a_i \) yields \( a_i = x^T v_i \), which establishes the claim.

Since the \( v_i \) are orthonormal, we have by (2.1) that
\[
\text{(2.3)} \quad \|\text{Proj}_L x\|^2 = \sum_{j=1}^{p} (x^T v_i)^2
\]

It can be useful to write (2.2) in matrix form. Let \( V \) be the \( n \times p \) matrix whose columns are \( v_1, v_2, \ldots, v_p \), that is
\[
V = [v_1 \ v_2 \ \cdots \ \ v_p] .
\]
Then we have
\[
\text{(2.4)} \quad \text{Proj}_L x = V V^T x.
\]
Note that since the columns of $V$ are orthonormal, we have $V^TV = I$, and so
\[(2.5) \quad (VV^T)^2 = VV^TVV^T = VV^T.\]

This is a natural property, and simply says that if we apply the projection twice, the second operation leaves the point unchanged, since it already lies in $L$. It is also clear that $VV^T$ is a symmetric matrix.

The residual is the difference $x - \text{Proj}_Lx$. The residual is orthogonal to $L$, and we thus say the projection is orthogonal projection. Indeed, we compute
\[
(x - \text{Proj}_Lx)^TV = (x - VV^Tx)^TV = x^TV - x^TVV^TV = 0.
\]

Note we used that $VV^T = I$ in the third equality above. Subsequently, we have
\[
\|x\|^2 = \|x - \text{Proj}_Lx + \text{Proj}_Lx\|^2 = \|x - \text{Proj}_Lx\|^2 + \|\text{Proj}_Lx\|^2,
\]
and so
\[(2.6) \quad \|x - \text{Proj}_Lx\|^2 = \|x\|^2 - \|\text{Proj}_Lx\|^2.
\]

The matrix corresponding to the residual mapping is $I - VV^T$, and, as with the projection, the residual matrix satisfies
\[(2.7) \quad (I - VV^T)^2 = I - 2VV^T + VV^TVV^T = I + VV^T.
\]

This is the analogous property to (2.5). The matrix $I - VV^T$ is also clearly a symmetric matrix.

It is sometimes useful to project onto affine spaces. An affine space has the form $x_0 + L$, where $L$ is a linear subspace. The key difference between affine and linear subspaces is that linear subspaces must contain the origin $0$, while an affine space translates the origin to a new point $x_0$. To project onto an affine subspace, we simply translate the affine space and the point $x$ to the origin, project onto $L$, and translate back. That is, the projection onto the affine space $A = x_0 + L$, denoted $\text{Proj}_A$, is given by
\[(2.8) \quad \text{Proj}_Ax = x_0 + \text{Proj}_L(x - x_0).
\]

We can also write this as
\[
\text{Proj}_Ax = x_0 - \text{Proj}_Lx_0 + \text{Proj}_Lx.
\]

---

3The notation means $x_0 + L = \{x_0 + y : y \in L\}$. 

13
Exercise 2.2. Let $L$ be a linear subspace of $\mathbb{R}^n$.

(i) Show that $\|\text{Proj}_L x\| \leq \|x\|$.

(ii) Show that $\text{Proj}_L x = x$ if and only if $x \in L$.

(iii) Show that if $\text{Proj}_L x = x$ for all $x \in \mathbb{R}^n$, then $L = \mathbb{R}^n$. \triangle

Exercise 2.3. Let $V$ and $W$ be orthogonal linear subspaces of $\mathbb{R}^n$. This means that for each $v \in V$ and $w \in W$ we have $w^T v = 0$. Define

$$V + W = \{v + w : v \in V \text{ and } w \in W\}.$$ 

Show that

$$\text{Proj}_{V+W} x = \text{Proj}_V x + \text{Proj}_W x.$$ \triangle

2.3 Diagonalization of symmetric matrices

Every symmetric matrix can be diagonalized. That is, for any symmetric matrix $A$, there exists an orthogonal matrix $Q$ and a diagonal matrix $D$ such that

$$A = QDQ^T.$$ 

An orthogonal matrix is a square matrix whose columns are orthonormal vectors. In this case, the columns of $Q$ are exactly the eigenvectors of the matrix $A$, which are orthogonal due to the symmetry of $A$. An orthogonal matrix also has the property that all rows are orthonormal and thus

$$Q^T Q = I = QQ^T.$$ 

Thus, the inverse of $Q$ is $Q^T$, and vice versa.

We generally arrange the eigenvalues of $A$ from smallest to largest

$$\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n.$$ 

The diagonal matrix $D$ has exactly the eigenvalues $\lambda_i$ on the diagonal, so $D(i, i) = \lambda_i$. Thus the decomposition $A = QDQ^T$ acting on a vector $x$ can be interpreted as changing basis into the coordinates of the eigenvectors by taking $Q^T x$, then multiplying by the diagonal matrix $D$, and then converting back to the standard coordinates.
Exercise 2.4. Let $Q$ be an orthogonal matrix. Show that $\|Qx\| = \|x\|$. △

Exercise 2.5. Let $A$ be a symmetric matrix, and consider the optimization problem

\[(2.9) \quad \min \{x^TAx : \|x\| = 1\}.\]

Since the set $\{x \in \mathbb{R}^n : \|x\| = 1\}$ is compact (closed and bounded), and the function $x \mapsto \|Ax\|$ is continuous, the optimization problem (2.9) admits a minimizer. Show that every minimizer $x^*$ is an eigenvector of $A$ with smallest eigenvalue. What happens if we switch the min to a max in (2.9)? [Hint: Diagonalize $A$ as $A = QDQ^T$ and write the optimization problem in terms of $y = Q^Tx$. Compute the optimal $y$ and convert back to the $x$ coordinates.] △

Exercise 2.6. We say a square matrix $A$ is positive semi-definite if $x^TAx \geq 0$ for all vectors $x$. Show that a symmetric matrix is positive semi-definite if and only if all its eigenvalues are nonnegative. [Hint: Diagonalize $A$.] △

2.4 Vector calculus

It will sometimes be useful to differentiate real-valued functions of a vector $x$, and we record some basic identities here. We recall that for a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient $\nabla f$ is defined by

$$\nabla f(x) = \left(\frac{\partial f}{\partial x(1)}, \frac{\partial f}{\partial x(2)}, \ldots, \frac{\partial f}{\partial x(n)}\right).$$

For example, for the function $f(x) = x(1)^2 - x(2)^2$ on $\mathbb{R}^2$, the gradient is $\nabla f(x) = (2x(1), -2x(2))$. The gradient is important since it characterizes minima and maxima of $f$, via the necessary condition $\nabla f = 0$.

For a linear function $f(x) = y^Tx$, we clearly have $\nabla f(x) = y$. For a quadratic function $f(x) = x^TAx$, where $A$ is an $n \times n$ matrix, we have

\[(2.10) \quad \nabla f(x) = (A + A^T)x.\]
Indeed, we compute

\[
\frac{\partial f}{\partial x(k)} = \frac{\partial}{\partial x(k)} \sum_{i=1}^{n} \sum_{j=1}^{n} A(i, j)x(i)x(j)
\]

\[
= \sum_{i=1}^{n} \sum_{j=1}^{n} A(i, j)(\delta(i, k)x(j) + \delta(j, k)x(i))
\]

\[
= \sum_{j=1}^{n} A(k, j)x(j) + \sum_{i=1}^{n} A(i, k)x(i)
\]

\[
= \sum_{i=1}^{n} (A(k, i) + A(i, k))x(i),
\]

which establishes the claim. Above, the notation \(\delta(i, j)\) is the Kronecker delta, which satisfies \(\delta(i, j) = 1\) when \(i = j\) and \(\delta(i, j) = 0\) when \(i \neq j\).

**Exercise 2.7.** Assume \(A\) is a symmetric matrix. Show that

\[
(2.11) \quad \nabla \|Ax\|^2 = 2A^2x,
\]

\[\triangle\]

**Exercise 2.8.** Let \(A\) be a symmetric matrix. Show that any minimizer \(x\) of (2.9) is an eigenvector of \(A\) with eigenvalue \(\lambda = x^T Ax\), without using that \(A\) is diagonalizable. [Hint: Any minimizer of (2.9) is also a minimizer of the Rayleigh quotient

\[
f(x) = \frac{x^T Ax}{x^Tx}.
\]

Compute \(\nabla f(x)\), set \(\nabla f(x) = 0\), and use that \(\|x\| = 1\).]

\[\triangle\]

## 3 Principal Component Analysis

One of the most important tasks in data analysis is that of finding simpler structures in data. One of the simplest mathematical structures is a linear subspace. In this lecture we will discuss how to find the best linear subspace approximating a collection of data points. This process is called principal component analysis (PCA). Projecting onto this best linear subspace effectively reduces the dimensionality of our data to the dimension of the subspace, yielding a useful dimension reduction algorithm. After developing the mathematical theory of PCA, we will explore applications to image compression and classification.
3.1 Fitting the best linear subspace

Let $x_1, x_2, \ldots, x_m$ be a collection of vectors in $\mathbb{R}^n$, which we treat as our data points. Figure 3.1 shows an example of a point cloud in $\mathbb{R}^2$ which is roughly one dimensional. Depending on the application, we may wish to approximate the data points by a linear or an affine subspace of $\mathbb{R}^n$. To find a line of best fit for the data in Figure 3.1, we would seek a one dimensional affine subspace that best approximates the data.

We proceed with the analysis for an affine space, and we will see that we can immediately reduce to the linear case. We seek an affine subspace $A = x_0 + L$, where $L$ is a $k$-dimensional linear subspace of $\mathbb{R}^n$ and $x_0 \in \mathbb{R}^n$, that best approximates our data in the mean squared sense. That is, we seek to minimize the mean squared error

$$E(x_0, L) = \sum_{i=1}^{m} \| x_i - \text{Proj}_A x_i \|^2$$

over all $k$-dimensional linear subspaces $L$ and translations $x_0 \in \mathbb{R}^n$. Let us write $L$ as the span of orthonormal vectors $v_1, v_2, \ldots, v_k$, and let $V$ be the matrix whose columns are the $v_i$.

We first consider the translation $x_0$. Recalling (2.8) we can write the energy $E$ as

$$E(x_0, L) = \sum_{i=1}^{m} \| x_i - x_0 - \text{Proj}_L (x_i - x_0) \|^2.$$
Then by (2.2) we have

\[ E(x_0, L) = \sum_{i=1}^{m} \| (I - VV^T)(x_i - x_0) \|^2. \]

Differentiating in \( x_0 \) using (2.11) we find that the optimal value for \( x_0 \) must satisfy

\[ 0 = \nabla_{x_0} E(x_0, L) = -2 \sum_{i=1}^{m} (I - VV^T)^2(x_i - x_0) = 0. \]

Let us define the mean, or centroid, of the data to be

\[ (3.2) \quad \bar{x} = \frac{1}{m} \sum_{i=1}^{m} x_i, \]

and recall that \((I - VV^T)^2 = (I - VV^T)\). Bringing the summation above inside we obtain

\[ 0 = (I - VV^T)(x_0 - \bar{x}) = x_0 - \bar{x} - \text{Proj}_L(x_0 - \bar{x}). \]

It follows that the optimal \( x_0 \) satisfies \( x_0 - \bar{x} \in L \). It is clear that the energy \( E \) is unchanged by adding an element of \( L \) to \( x_0 \), so we will take \( x_0 = \bar{x} \) for simplicity.

**Exercise 3.1.** Show that if \( v \in L \) then \( E(x_0 + v, L) = E(x_0, L) \). \( \triangle \)

The discussion above allows us to reduce to the case of fitting a linear subspace to our data. Indeed, if we wish to fit an affine subspace, the optimal offset is the centroid \( \bar{x} \), and we can simply center our data, by replacing \( x_i \) with \( x_i - \bar{x} \), and reduce to the problem of fitting a linear subspace to the centered data. Thus, without loss of generality we consider the problem of minimizing the energy

\[ (3.3) \quad E(L) = \sum_{i=1}^{m} \| x_i - \text{Proj}_L x_i \|^2 \]

over all \( k \)-dimensional linear subspaces \( L \) of \( \mathbb{R}^n \). Equivalently, we can view the problem as minimizing \( E \) over the orthonormal vectors \( v_1, v_2, \ldots, v_k \) that span \( L \). To proceed further, we rewrite the energy in a more convenient form.
Lemma 3.2. The energy $E(L)$ can be expressed as

$$E(L) = \text{Trace}(M) - \sum_{j=1}^{k} v_{j}^{T} M v_{j},$$

where $M$ is the covariance matrix of the data, given by

$$M = \sum_{i=1}^{m} x_{i} x_{i}^{T}.$$

Remark 3.3. By Exercise 2.1, the covariance matrix $M$ can also be written as $M = X^{T}X$, where $X$ is the $m \times n$ matrix whose $i$th row is $x_{i}^{T}$, which can be written as $X = [x_{1} \ x_{2} \ \cdots \ x_{m}]^{T}$. In practice, one is usually given the data matrix $X$ directly, and the formula $M = X^{T}X$ is both more convenient and more efficient in many programming languages.

Proof of Lemma 3.2. We recall (2.3) and (2.6), and compute

$$E(L) = \sum_{i=1}^{m} (\|x_{i}\|^{2} - \|\text{Proj}_{L}x_{i}\|^{2})$$

$$= \sum_{i=1}^{m} \|x_{i}\|^{2} - \sum_{i=1}^{m} \sum_{j=1}^{k} (v_{j}^{T} x_{i})^{2}$$

$$= \sum_{i=1}^{m} \|x_{i}\|^{2} - \sum_{j=1}^{k} v_{j}^{T} \left( \sum_{i=1}^{m} x_{i} x_{i}^{T} \right) v_{j}$$

$$= \sum_{i=1}^{m} \|x_{i}\|^{2} - \sum_{j=1}^{k} v_{j}^{T} M v_{j}.$$

To complete the proof, we note that

$$\text{Trace}(M) = \sum_{i=1}^{m} \text{Trace}(x_{i} x_{i}^{T}) = \sum_{i=1}^{m} \|x_{i}\|^{2}. \quad \square$$

The first term in (3.4) is independent of $L$, so we may focus on the second term. Due to the minus sign, Lemma 3.2 shows that minimizing $E$ is equivalent to maximizing the quantity

$$\sum_{j=1}^{k} v_{j}^{T} M v_{j}$$
over orthonormal vectors \(v_1, v_2, \ldots, v_k\). We note that the covariance matrix \(M\) is a symmetric matrix, thus it is diagonalizable. This means there exists an orthogonal matrix \(P\), whose columns are the orthonormal eigenvectors of \(M\), and a diagonal matrix \(D\), whose diagonal entries are the corresponding eigenvalues, such that \(M = PDP^T\). We arrange the eigenvalues from largest to smallest, so \(D_{ii} = \lambda_i\) and

\[
\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n. \tag{3.6}
\]

Let \(p_1, \ldots, p_n\) denote the corresponding orthonormal eigenvectors of \(M\), which are just the columns of \(P\).

We claim that \(M\) is positive semi-definite, that is, that \(\lambda_i \geq 0\) for all \(i\). To see this, simply note that

\[
\lambda_j = p_j^T M p_j = \sum_{i=1}^{m} p_j^T x_i x_i^T p_j = \sum_{i=1}^{m} (p_j^T x_i)^2 \geq 0. \tag{3.7}
\]

Since \(p_j^T x_i\) is the projection of \(x_i\) onto \(p_j\), we see that \(\lambda_j\) is simply the variance of the data in the direction \(p_j\) (up to a normalizing factor of \(1/m\)). We now compute

\[
\sum_{j=1}^{k} v_j^T M v_j = \sum_{j=1}^{k} v_j^T PDP^T v_j = \sum_{j=1}^{k} \| D^{1/2} P^T v_j \|^2 = \sum_{j=1}^{k} \sum_{i=1}^{n} \lambda_i (p_i^T v_j)^2 = \sum_{i=1}^{n} \sum_{j=1}^{k} \lambda_i (p_i^T v_j)^2 = \sum_{i=1}^{n} \lambda_i \| \text{Proj}_{LP_i} \|^2.
\]

Notice that

\[
\sum_{i=1}^{n} \| \text{Proj}_{LP_i} \|^2 = \sum_{j=1}^{k} \sum_{i=1}^{n} (p_j^T v_j)^2 = \sum_{j=1}^{k} 1 = k.
\]

Thus, we must choose the \(v_i\) so as to distribute the weights \(\| \text{Proj}_{LP_i} \|^2\) among the largest eigenvalues as much as possible. A natural choice is to set \(v_i = p_i\)
for \( i = 1, \ldots, k \). Then \( p_i \in L \) for \( i = 1, \ldots, k \), and the \( p_i \) are orthogonal to \( L \) for \( i \geq k + 1 \), yielding

\[
\| \text{Proj}_L p_i \|^2 = \begin{cases} 
1, & \text{if } 1 \leq i \leq k \\
0, & \text{otherwise.}
\end{cases}
\]

This choice of \( v_i \) then yields

\[
(3.8) \quad \sum_{j=1}^{k} v_j^T M v_j = \sum_{i=1}^{k} \lambda_i.
\]

The exercise below verifies that this choice of \( v_i = p_i \) is indeed optimal.

**Exercise 3.4.** Suppose that \( \lambda_i \) satisfy (3.6) and let \( a_1, \ldots, a_n \) satisfy \( 0 \leq a_i \leq 1 \) and \( \sum_{i=1}^{n} a_i = k \), where \( k \) is an integer \( 1 \leq k \leq n \). Show that

\[
\sum_{i=1}^{n} \lambda_i a_i \leq \sum_{i=1}^{k} \lambda_i.
\]

\( \Delta \)

We summarize our findings in a theorem.

**Theorem 3.5.** Let \( p_1, p_2, \ldots, p_n \) be the orthonormal eigenvectors of the covariance matrix \( M \), defined in (3.5), with corresponding eigenvalues \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \), given in decreasing order. The energy \( E(L) \), defined in (3.3), is minimized over \( k \)-dimensional linear subspaces \( L \subset \mathbb{R}^n \) by setting

\[
L = \text{span}\{p_1, p_2, \ldots, p_k\}
\]

and the optimal energy is given by

\[
E(L) = \sum_{i=k+1}^{n} \lambda_i.
\]

**Proof.** All that remains to prove is the formula for \( E(L) \). This follows from Eq. (3.8), Lemma 3.2, and the identity

\[
\text{Trace}(M) = \sum_{i=1}^{n} \lambda_i. \quad \square
\]

The vectors \( p_1, p_2, \ldots \) in Theorem 3.5 are called the principal components of the data, and the eigenvalues \( \lambda_1, \lambda_2, \ldots \) describe the amount of variation of
the data in the direction of each principal component, due to (3.7). In Figure 3.2 we plot the principal components of the point cloud from Figure 3.1, scaled to length $2\sqrt{\lambda_i}$ to match the variation in each direction of the data.

Let $P_k = [p_1 \ p_2 \ \cdots \ p_k]$. Then, recalling Section 2.2, the projection of a vector $x$ onto the PCA subspace $L$ is given by

$$\text{Proj}_L x = P_k P_k^T x.$$  

The coordinates of the point $x$ in the subspace $L$ are the contents of the length $k$ vector $P_k^T x$. In this sense, we can view PCA as dimension reduction, since the length $n$ vector $x$ is approximated by the length $k$ vector $P_k^T x$. Here, PCA gives a dimension reduction from $\mathbb{R}^n$ to $\mathbb{R}^k$.

### 3.2 How many principal directions?

A basic question concerns how to chose the number of principal components $k$ to use in PCA. A standard way to do this is to instead specify how much of the variation in the data one wishes to capture with the subspace $L$. Let $0 < \alpha \leq 1$ describe this quantity. For example, $\alpha = 0.95$ is interpreted as requiring that the subspace $L$ capture 95% of the variation in the data. Since the eigenvalue $\lambda_i$ describes the amount of variation in the principal direction

---

Figure 3.2: A depiction of the principal directions obtained by running PCA on the point cloud from Figure 3.1. The principal directions are scaled by $2\sqrt{\lambda_i}$ for visualization.
pi, we can simply choose k as small as possible while ensuring that
\[ \sum_{i=1}^{k} \lambda_i \geq \alpha \sum_{i=1}^{n} \lambda_i. \]

As in the proof of Theorem 3.5, we have \( \sum_{i=1}^{n} \lambda_i = \text{Trace}(M) \) and so we may rewrite the condition as
\[ (3.9) \quad \sum_{i=1}^{k} \lambda_i \geq \alpha \text{Trace}(M). \]

The condition (3.9) can be checked without computing all of the eigenvalues of M, which may be computationally intensive in high dimensional applications, where n is very large. Instead, one can use an iterative eigenvalue solver, which finds the eigenvectors in order of decreasing eigenvalue, and stop the first time (3.9) holds.

### 3.3 Robust PCA

While PCA is simple to work with mathematically and computationally, it can be very sensitive to outliers in the data. This is due to its use of the mean squared error in (3.3), which strongly penalizes outliers. We show an example in Figure 3.3 of how a single outlier can lead to a large negative affect on the ability of PCA to accurately fit the main part of the point cloud.

Thus, it is important to remove outliers before applying PCA. Outlier detection can, however, be difficult, and so there has been substantial interest in more robust versions of PCA that are not sensitive to outliers. Many variants of robust PCA have been proposed in the literature. Some are based on minimizing a sum of distances energy of the form
\[ (3.10) \quad F(L) = \sum_{i=1}^{m} \| x_i - \text{Proj}_L x_i \|. \]

By omitting the square, we are placing a far lower penalty on severe outliers, and can achieve better performance. However, it is far more computationally challenging to minimize \( F(L) \), compared to the mean-squared error \( E(L) \), since there is no longer a simple relationship to the eigenvectors of the covariance matrix.
Figure 3.3: An illustration of how PCA is sensitive to outliers. Here, we used the same point cloud as in Figure 3.1, but added a single outlying point to the lower right of the point cloud (not depicted), whose distance from the point cloud is twice the length (longest dimension) of the point cloud. Since PCA minimize the sum of squared errors, it is overly concerned with approximating outliers, whose distance is far from the inlying point cloud.

**Exercise 3.6.** Consider the weighted PCA energy

\[ E_w(L) = \sum_{i=1}^{m} w_i \| x_i - \text{Proj}_L x_i \|^2, \]

where \( w_1, w_2, \ldots, w_m \) are nonnegative numbers (weights).

(i) Show that the weighted energy \( E_w \) is minimized over \( k \)-dimensional subspaces \( L \subset \mathbb{R}^n \) by setting

\[ L = \text{span}\{ p_1, p_2, \ldots, p_k \}, \]

where \( p_1, p_2, \ldots, p_n \) are the orthonormal eigenvectors of the covariance matrix

\[ M_w = \sum_{i=1}^{m} w_i x_i x_i^T, \]

with corresponding eigenvalues \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \), given in decreasing order.

(ii) Show that the weighted covariance matrix can also be expressed as

\[ M_w = X^T W X, \]

24
where $W$ is the $m \times m$ diagonal matrix with diagonal entries $w_1, w_2, \ldots, w_m$, and

$$X = [x_1 \ x_2 \ \cdots \ x_m]^T.$$  

(iii) Show that the optimal energy is given by

$$E_w(L) = \sum_{i=k+1}^{n} \lambda_i.$$  

(iv) Suppose we minimize $E_w$ over affine spaces $x_0 + L$, so

$$E_w(x_0, L) = \sum_{i=1}^{m} w_i \|x_i - x_0 - \text{Proj}_{L}(x_i - x_0)\|^2.$$  

Show that an optimal choice for $x_0$ is the weighted centroid

$$x_0 = \frac{\sum_{i=1}^{m} w_i x_i}{\sum_{i=1}^{m} w_i}.$$  

Project 3.1 (Robust PCA). Python Notebook: .ipynb

Robust PCA refers to a class of algorithms that minimize an energy of the form

$$E(x_0, L) = \sum_{i=1}^{m} \Phi(\|x_i - x_0 - \text{Proj}_{L}(x_i - x_0)\|^2).$$  

Choosing $\Phi(s) = s$ yields ordinary PCA, while $\Phi(s) = \sqrt{s}$ yields the robust sum of distances energy $F$ given in (3.10). In this project you will implement an iteratively re-weighted least squares (IRLS) algorithm for minimizing the robust PCA energy (3.11), and test it on some data with outliers. Please refer to the Python notebook linked above and complete the steps below.

1. Write a Python function to implement the weighted PCA from Exercise 3.6.

2. Write Python code to minimize (3.11) using the IRLS method, which solves a sequence of weighted PCA problems

$$(x_0^{k+1}, L^{k+1}) = \min_{(x_0, L)} \sum_{i=1}^{m} w_i^k \|x_i - x_0 - \text{Proj}_{L}(x_i - x_0)\|^2.$$  

25
where
\[ w_i^k = \frac{\Phi(\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2)}{\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2}. \]

Start with only a handful of iterations. You can then play with a stopping condition that checks how much the weights \( w_i^k \) change each iteration.

You may find that you divide by zero, or very small numbers, in the definition of the weights \( w_i^k \), which can cause problems. A common way to regularize is to define
\[ w_i^k = \frac{\Phi(\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2)}{\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2 + \varepsilon}, \]
where \( \varepsilon \) is a small number, say \( \varepsilon = 10^{-5} \).

3. Try your Robust PCA method on data with outliers. How many outliers do you need to add to break the algorithm?

\[ \triangle \]

### 3.4 PCA-based Image Compression

**Python Notebook:** .ipynb

We now give a first application of PCA to the problem of image compression. To apply PCA in this context, we must break an image up into pieces in such a way that the pieces are expected to be particularly simple and can be expressed well by PCA with a small number of principal components.

The simplest way to do this would be split the image up into its rows or columns. Let \( X \) be an \( m \times n \) matrix representing an \( m \times n \) grayscale image. If we split the image up by rows, then this amounts to applying PCA directly to the matrix \( X \), where each row is considered as a data point. In this case we do not center the data, so based on Remark 3.3, the covariance matrix is \( M = X^TX \). Let \( k \) denote the number of principal components to keep and let \( P_k = [p_1 \ p_2 \ \cdots \ p_k] \) be the matrix whose columns are the principal components, and let \( L_k \) denote the span of \( p_1, \ldots, p_k \). For a given vector \( x \), PCA approximates \( x \) by its projection onto \( L_k \), that is \( \text{Proj}_{L_k}x = P_kP_k^Tx \).

Since we work with row vectors in \( X \), let’s consider the transpose of this quantity, so that \( x_i^TP_kP_k^T \), is the PCA approximation of the \( i \)th row of \( X \), if \( X = [x_1 \ x_2 \ \cdots \ x_m]^T \). Thus, the PCA approximation of \( X \) is given by
\[ X \approx XP_kP_k^T. \]
Why does this amount to compression? To compute the right hand side above, we only need to store the matrix \( P_k \), which is \( n \times k \), and the matrix \( XP_k \), which is \( m \times k \). The latter matrix \( XP_k \) represents the coordinates of each row of \( X \) in the basis of \( L_k \). Thus, instead of storing the \( m \times n \) matrix \( X \), we store two matrices of size \( n \times k \) and \( m \times k \). Often \( m = n \) (the image is square), and so the compression ratio is \( n : k \).

While row-wise compression of images is an obvious first choice, the method misses important structure in the image, and the rows are not necessarily simple pieces of the image to compress. Since each row spans the entire image, the pixel values across the entire row may be entirely unrelated and have very little redundancy (which is useful for compression). Furthermore, why not use columns? Neighboring pixels vertically in the image should also have some correlation, but this is entirely ignored by row-wise image compression.

A better way to split up an image for compression is to use patches, or blocks, that are localized in space. The pixel intensities do not often vary rapidly in local areas of the image, and so small patches are expected to be well-approximated by a low dimensional PCA approximation. We work with the 512 \( \times \) 512 cameraman image shown in Figure 1.2. We use 8 \( \times \) 8 pixel patches in a regular grid, so the image contains 64 \( \times \) 64 = 4096 patches, each containing 8 \( \times \) 8 = 64 pixels. Figure 3.4 shows some of the patches of the cameraman image. Working with patches instead of rows requires a small amount of preprocessing of the image, to split it into patches. This produces
a matrix $X$ of size $4096 \times 64$, and we apply PCA to this matrix, instead of to the image itself, as we did in the row-wise compression example above. After this preprocessing, the compression proceeds exactly the same as in the row-wise compression example, and the decompressed image then needs to be reconstructed from its patches.

The reconstruction error in image compression is measured with the peak signal to noise ratio (PSNR). The PSNR computation is based on the mean squared error

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (I(i, j) - I_0(i, j))^2.$$ 

Here, $I_0$ is the original image and $I$ is the reconstructed image after compression. Both images have size $m \times n$. The PSNR computation also uses the peak signal value, $S_{\text{peak}}$, which is the largest possible value of the pixel intensity. For 8-bit images, the largest value is $S_{\text{peak}} = 2^8 - 1 = 255$. The PSNR is then given by

$$\text{PSNR} = 10 \log_{10} \left( \frac{S_{\text{peak}}^2}{\text{MSE}} \right).$$

The PSNR is measured in decibels dB. PSNR values of 20 dB to 30 dB are very low quality images you may see on wireless devices, while 30 dB to 50 dB are respectable, and above 50 dB are very good quality compressions. This discussion is for 8-bit images, and the values would change for higher bit-depth images.

Figure 3.5 shows the compressed and difference cameraman images at three different compression ratios, with PSNR ranging from 34 dB up to 51 dB. The reader should note the blocking-type artifacts at higher compression ratios. These are caused by the decomposition of the image into patches, which allows for the reconstructed patches to differ greatly near the patch boundary. In Figure 3.6 we plot the PSNR versus compression ratio for patch-based (or block-based) image compression and row-wise image compression. This clearly shows the advantage of using block-based compression instead of row-wise compression.

The same types of blocking artifacts are present in older jpeg compression algorithms, which are based on the same principle of splitting the image into patches. Instead of using PCA to find a basis for the patch space, the jpeg algorithm uses the Fourier series basis, which we will learn about later in the course. We show in Figure 3.7 the first 30 principal components obtained

\footnote{The more recent jpeg2000 algorithm uses wavelet-based compression that does not require splitting the image into patches, and does not have blocking artifacts.}
Figure 3.5: Examples of PCA-based image compression on the cameraman image at different compression ratios. Left is original, center is compressed, and right is the difference image.
Figure 3.6: PSNR vs Compression Ratio for block-wise and row-wise compression of the cameraman image.

Figure 3.7: The first 30 principal components extracted via PCA on $8 \times 8$ pixel patches of the cameraman image.

by applying PCA to the image patches. The principal components start of as low frequency, smooth, features, while the later components describe more high frequency content, like texture. When we study Fourier analysis later in the course, we will find that the Fourier basis for patch space looks strikingly similar to the principal components in Figure 3.7.

Let us also mention one aspect of compression we are omitting from this discussion. Normally the compressed data, which is $XP_k$ and $P_k$, would be
further compressed with a lossless compression algorithm (like zip compression) to save additional space. This adds further compression above the lossy compression we have described in this section. The mathematics of lossless compression are unfortunately beyond the scope of this course.

**Project 3.2 (Audio Compression). Python Notebook: .ipynb**

This project will use the same PCA-based compression algorithm for audio compression. There are two parts to the project, and the notebook above contains some code to get you started.

1. Use the same block-based image compression algorithm, described above, for audio compression. You can use any audio file you like; the python notebook linked above downloads a classical music sample from the course website that you can use. A stereo audio signal is an array of size $n \times 2$, where $n$ is the number of samples. Use blocks of size $N \times 2$ for compression.

2. When you play back the compressed audio file, you will likely hear some static noise artifacts, even at very low compression rates. These are caused by blocking artifacts, where the signals do not match up on the edges of the blocks, which introduces discontinuities into the signal (which are very high in frequency). This is similar to the blocking artifacts we observed in image compression (see Figure 3.5), however, the artifacts are more noticeable in audio than in images.

To fix this problem, audio compression algorithms use overlapping blocks, and apply a windowing function in order to smoothly patch together the audio in each block. The blocks are structured so that half of the first block overlaps with half of the second block, and so on. To implement this in python, just shift the signal by half of the block width, and apply the `images_to_patches` function on the original and shifted signals. Then compress and decompress both signals. After decompressing, and before converting back from the patch format to the audio signal, you’ll need to multiply by a windowing function to smooth the transition between patches. If the patch size is $2 \times N$, then each channel should be multiplied by a window function $w_i$, $i = 0, 1, \ldots, N - 1$. A common window function that is used, for example, in mp3 compression, is

$$w_i = \sin^2 \left( \frac{\pi}{N} \left( i + \frac{1}{2} \right) \right).$$

After you decompress and apply the window, undo the shift and add the signals together to get the decompressed audio. Does this improve the audio quality?
As a note, in order to make sure the shifted signals add up correctly, we need that

\[ w_i + w_{i+N/2} = 1. \]

As an exercise, the reader should check that the window function above satisfies this condition, which is called the Princen-Bradley condition.

\[ \triangle \]

### 3.5 PCA-based Handwritten Digit Recognition

**Python Notebook: .ipynb**

We now give a further application of PCA to the problem of handwritten digit recognition on the MNIST dataset, depicted in Figure 1.3. Recall MNIST is a dataset consisting of 70,000 images of handwritten digits 0 through 9. Each image is a 28 × 28 pixel grayscale image, which we can view as a vector in \( \mathbb{R}^{784} \) by flattening the image array, since \( 28^2 = 784 \).

PCA allows us to deduce a simple linear model for the images belonging to each digit. We choose a number of principal components \( k \), and use PCA to learn affine subspaces \( A_0, A_1, \ldots, A_9 \) for each of the 10 digits. A new image of a handwritten digit is classified by projecting the image onto each of the 10 affine spaces, and finding which it is closest to. In order have held-out testing images that are not used for constructing the affine subspaces, we use the first 60,000 images in the dataset to build the affine spaces, and the last 10,000 to test the accuracy of the digit recognition. This is a standard training/testing split for the MNIST dataset.

Mathematically, let \( P_0, P_1, \ldots, P_9 \) denote the 784×\( k \) matrices whose columns are the \( k \) principal components for each class. In this case, we center the data before applying PCA, so let \( \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_9 \) denote the mean images from each class. The mean MNIST images are shown in Figure 3.8, and the first 10 principal components per class are shown in Figure 3.9. To project a flattened MNIST image \( x \in \mathbb{R}^{784} \) onto the affine space \( A_i \), we recall from Section 2.2 that the formula for projection onto an affine space is

\[
\text{Proj}_{A_i} x = \overline{x}_i + P_i P_i^T (x - \overline{x}_i).
\]
To check how far $x$ is from the affine space $A_i$, we compute the norm of the difference between $x$ and its projection, that is

$$d_i(x) := \|x - \text{Proj}_{A_i}x\| = \|(I - P_iP_i^T)(x - x_i)\|.$$  

The quantity $d_i$ is the distance between $x$ and the affine space $A_i$. We can then classify the image $x$ by choosing the closest affine space, that is, the label $\ell_i(x)$ for image $x$ is given by

$$\ell_i(x) = \arg\min_{0 \leq i \leq 9} d_i(x).$$

Running this on the MNIST dataset produces a classifier that achieves 95.8% accuracy on the 10,000 held out testing images. This is a very good result for such a basic algorithm. State of the art deep neural networks can achieve
over 99% accuracy. Figure 3.10 shows some of the incorrectly labeled digits, along with the incorrectly predicted label. We can see many of the incorrectly classified digits are poorly written and thus difficult to classify.

Remark 3.7. The PCA-based classifier we have constructed is a simple affine mapping \((I - P_i P_i^T)(x - \mu_i)\), followed by a nonlinear operation—computing the norm of this quantity. We then check which of \(d_0(x), d_1(x), \ldots, d_9(x)\) is smallest to decide on the label for the image \(x\). This is similar to a single layer neural network (a neuron is a linear function composed with a nonlinear activation function). We will cover the basics of neural networks later in the course.

Remark 3.8. In Python, and other programming languages, it is inefficient to compute with single images, so it is useful to write the formulas above in terms of a matrix \(X\), whose rows are images from the MNIST dataset. Then we must work with the transposes of the quantities above, so

\[
(X - \mu_i^T)(I - P_i P_i^T)
\]

represents the difference between the projections of all rows of \(X\) onto the \(i^{th}\) affine space \(A_i\).

Let us mention briefly that the expression \(X - \mu_i^T\) is intended to mean that we subtract the row vector \(\mu_i^T\) from all rows of \(X\). This type of expression is allowed in Python and other languages like Matlab. To be absolutely correct mathematically, the expression should be written as

\[
(X - \textbf{1} \mu_i^T)(I - P_i P_i^T)
\]

where \(\textbf{1}\) is the all-ones vector whose length is the same as the number of rows in \(X\).

Project 3.3 (EigenFaces). Python Notebook: .ipynb

In this project you will explore a PCA-based facial recognition algorithm that is known as EigenFaces. The Python notebook above has code to get you started, including downloading and viewing a database of face images.
The dataset has 2414 images of faces, from 38 different subjects. Each person appears many times in the database under different lighting conditions, etc. The goal of face recognition is to match a new image of a face to an image in an existing database. Please refer to the Python notebook above while completing the project steps below.

1. To have held-out data for testing, we will first split the face dataset into training and testing sets. Use the `train_test_split` function in `sklearn.model_selection` to split the dataset randomly into training (70%) and testing (30%).

2. Run PCA on the training images to learn an affine space that well approximates the training set. You can leave the number of principal components, $k$, as a parameter; around $k = 100$ gives good results.

3. Project both the training images and testing images onto the affine space, using the lower dimensional coordinates of the affine space. If the testing (or training) images are stored in a matrix $X$, where each row is a flattened image, and $\bar{x}$ is the mean face image computed by PCA in Step 1 (as a row vector), then the PCA coordinates are given by

   $$(X - \bar{x})P,$$

   where $P$ is the matrix containing the principle components as columns. Then match each testing image to the training image that is closest in Euclidean distance in the PCA coordinates.

4. You may not get very good results with the method above (around 60% accuracy). Try using the Mahalanobis distance instead of Euclidean distance to match faces in the PCA coordinates. For two vectors $x$ and $y$ of length $k$, representing the PCA coordinates of two images, the Mahalanobis distance is given by

   $$d_M(x, y) = \sum_{i=1}^{k} \lambda_i^{-1} (x(i) - y(i))^2,$$

   where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$ are the eigenvalues of the covariance matrix (obtained by PCA in Step 1). The Mahalanobis distance rescales the norm along each coordinate to match the variation in the data in the corresponding principal direction. You should be able to get around 90% accuracy with the Mahalanobis distance, with some variation depending on the random training/testing split.

\[\triangle\]
4 Clustering

We now turn to the problem of clustering, or grouping, data. Figure 4.1 shows a sample dataset consisting of 500 points that appear to belong to three distinct clusters. Two of the clusters are very close together, compared to the third, which more isolated. The goal of clustering is to separate the points in Figure 4.1 into the three natural clusters. In general, when working with real data and not synthetic examples, it is difficult to visualize the “natural clusters” within data, and it can be difficult to define what constitutes a good or bad clustering (since there are various natural ways one can group data).

In this section we will study two popular and widely used algorithms for clustering: $k$-means clustering, and spectral clustering. We will study the mathematics behind each algorithm, to the extent that we can in this course, and highlight the advantages and drawbacks of each method. We will also give applications to real data, by considering the problem of clustering pairs of digits from the MNIST dataset.

4.1 $k$-Means Clustering

Python Notebook: .ipynb

The $k$-means algorithm aims to find a single good representative point from each of $k$ clusters. The dataset is then clustered into $k$ groups by assigning each datapoint to the cluster corresponding to the closest such representative point in the Euclidean distance. To describe the setting mathematically, let
Let \( x_1, x_2, \ldots, x_m \) be a dataset consisting of \( m \) points in \( \mathbb{R}^n \). Let \( c_1, c_2, \ldots, c_k \) be the cluster centers, which are vectors in \( \mathbb{R}^n \) and are yet to be determined. The \( k \)-means algorithm is guided by the task of minimizing the \( k \)-means clustering energy

\[
E(c_1, c_2, \ldots, c_k) = \sum_{i=1}^{m} \min_{1 \leq j \leq k} \| x_i - c_j \|^2.
\]

Minimizing \( E \) over the cluster centers (also called “means”) \( c_1, \ldots, c_k \) aims to find \( k \) points that well-represent the dataset, in the sense that all points are close to at least one \( c_j \) in the squared Euclidean distance. If we are able to minimize \( E \), then the \( j \)-th cluster in the dataset, denoted \( \Omega_j \), consists of all points \( x_i \) that are closer to \( c_j \) than they are to any other cluster center. That is

\[
\Omega_j = \left\{ x_i : \| x_i - c_j \|^2 = \min_{1 \leq \ell \leq k} \| x_i - c_\ell \|^2 \right\}.
\]

It turns out that minimizing the \( k \)-means clustering energy \( E \) is very hard computationally (it has been shown to be \( \text{NP-hard} \)). However, it is possible to construct a simple algorithm that descends on the energy \( E \), is provably convergent (to a local minimizer), and often gives good clustering results. This is called the \( k \)-means algorithm, and is outlined below.

**\( k \)-means algorithm:** We start with some randomized initial values for the means \( c_1^0, c_2^0, \ldots, c_k^0 \), and iterate the steps below until convergence.

1. Update the clusters

\[
\Omega_j^t = \left\{ x_i : \| x_i - c_j^t \|^2 = \min_{1 \leq \ell \leq k} \| x_i - c_\ell^t \|^2 \right\}.
\]

2. Update the cluster centers

\[
c_j^{t+1} = \frac{1}{\#\Omega_j^t} \sum_{x \in \Omega_j^t} x.
\]

Above, the notation \( \#\Omega_j^t \) denotes the number of points in the \( j \)-th cluster \( \Omega_j^t \) at the \( t \)-th step in the algorithm. Hence, \( c_j^{t+1} \) is exactly the mean of the \( j \)-th cluster \( \Omega_j^t \). The \( k \)-means algorithm generates a sequence of clusterings \( \Omega_j^0, \Omega_j^1, \Omega_j^2, \ldots \) and cluster means \( c_j^0, c_j^1, c_j^2, \ldots \), for \( j = 1, \ldots, k \), that get progressively better in the sense that the \( k \)-means clustering energy (4.1) is decreasing (which
Figure 4.2: An illustration of the intermediate steps in the $k$-means clustering algorithm. The red points are the cluster centroids. The algorithm converged in 8 steps, but steps 6, 7, 8 showed very little change in the clustering.

Figure 4.3: Two examples of poor clusterings obtained by the $k$-means algorithm. The final clustering obtained by $k$-means is not unique, and depends on the random initial condition.

we prove below). The algorithm converges when the clusters (and hence the cluster means) do not change from one iteration to the next, that is $\Omega_j^t = \Omega_j^{t+1}$ for all $j = 1, \ldots, k$. We also note that a point $x_i$ may be equally close to more than one cluster center, and in this case we can make any reasonable choice of which cluster to assign it to, such as the cluster whose index is smallest.
We show in Figure 4.2 an illustration of some of the intermediate steps in applying \( k \)-means clustering with \( k = 3 \) (i.e., the 3-means clustering algorithm) to the point cloud from Figure 4.1. The algorithm converged in 8 steps to a good clustering, although this depends on the randomized initial condition. For some initializations the algorithm converged in fewer iterations, sometimes as few as three, while for other initializations the algorithm took longer to converge. The clustering obtained can also depend on the initial condition. We show in Figure 4.3 two examples of poor clusterings obtained by 3-means clustering of the same point cloud.

The \( k \)-means algorithm often gives good results, but it does not find a global minimizer of the \( k \)-means clustering energy \( E \) defined in (4.1), and can converge to local minimizers that give poor clustering results. Nevertheless, we can prove that the \( k \)-means algorithm descends on the energy \( E \), and converges in a finite number of iterations. The proof requires a preliminary lemma, which shows that the centroid (or mean) minimizes the sum of squared distances to the cluster center.

**Lemma 4.1.** Let \( y_1, y_2, \ldots, y_m \) be points in \( \mathbb{R}^n \), and define the function \( f : \mathbb{R}^n \to \mathbb{R} \) by
\[
f(x) = \sum_{i=1}^{m} \left\| y_i - x \right\|^2.
\]
Then the unique minimizer of \( f \) is the centroid
\[
c = \frac{1}{m} \sum_{i=1}^{m} y_i.
\]

**Proof.** First, we claim that
\begin{equation}
(4.4) \quad f(c) = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2).
\end{equation}
To see this, we first compute
\[
\|c\|^2 = c^T c = \left( \frac{1}{m} \sum_{i=1}^{m} y_i \right)^T \left( \frac{1}{m} \sum_{j=1}^{m} y_j \right) = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} y_i^T y_j.
\]
From this, it follows that

\[ f(c) = \sum_{i=1}^{m} \|y_i - c\|^2 \]

\[ = \sum_{i=1}^{m} \left( \|y_i\|^2 - 2y_i^T \frac{1}{m} \sum_{j=1}^{m} y_j + \|c\|^2 \right) \]

\[ = \sum_{i=1}^{m} \|y_i\|^2 - 2m \sum_{i=1}^{m} \sum_{j=1}^{m} y_i^T y_j + m\|c\|^2 \]

\[ = \sum_{i=1}^{m} \|y_i\|^2 - 2m\|c\|^2 + m\|c\|^2 \]

\[ = \sum_{i=1}^{m} \|y_i\|^2 - m\|c\|^2 = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2) \]

which establishes the claim.

Now, using (4.4) we obtain

\[ f(z) = \sum_{i=1}^{m} \|y_i - z\|^2 \]

\[ = \sum_{i=1}^{m} \left( \|y_i\|^2 - 2z^T y_i + \|z\|^2 \right) \]

\[ = \sum_{i=1}^{m} \|y_i\|^2 - 2m\|c\|^2 + m\|z\|^2 \]

\[ = f(c) + m\|c\|^2 - 2mz^T c + m\|z\|^2 \]

\[ = f(c) + m\|c - z\|^2 \]

for any \( z \in \mathbb{R}^n \). Therefore \( f(c) \leq f(z) \) for all \( z \in \mathbb{R}^n \), with equality if and only if \( z = c \). This completes the proof. \( \square \)

We can now prove convergence of the \( k \)-means algorithm.

**Theorem 4.2.** The \( k \)-means algorithm descends on the energy (4.1), that is

\[ (4.5) \quad E(c_{t+1}^1, c_{t+1}^2, \ldots, c_{t+1}^k) \leq E(c_1^t, c_2^t, \ldots, c_k^t) \]

Furthermore, we have equality in (4.5) if and only if \( c_{j+1}^j = c_j^j \) for \( j = 1, \ldots, k \), and hence the \( k \)-means algorithm converges in a finite number of iterations.
Proof. The proof is based on re-writing the $k$-means energy in the following way:

$$E(c_1^t, c_2^t, \ldots, c_k^t) = \sum_{j=1}^k \sum_{x \in \Omega_j^t} \|x - c_j^t\|^2.$$ 

This follows from the definition of the clusters $\Omega_j^t$ defined in (4.2). Now, it follows from Lemma 4.1 that

$$\sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2 \leq \sum_{x \in \Omega_j^t} \|x - c_j^t\|^2$$

with equality if and only if $c_j^{t+1} = c_j^t$, due to the definition of $c_j^{t+1}$ in (4.3) as the mean of the $j$th cluster $\Omega_j^t$ at step $t$. Therefore

$$\sum_{j=1}^k \sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2 \leq E(c_1^t, c_2^t, \ldots, c_k^t)$$

with equality if and only if $c_j^{t+1} = c_j^t$ for $j = 1, \ldots, k$. Since the $k$-means energy uses the squared distance to the closest cluster center, we trivially have

$$E(c_1^{t+1}, c_2^{t+1}, \ldots, c_k^{t+1}) = \sum_{i=1}^m \min_{1 \leq j \leq k} \|x_i - c_j^{t+1}\|^2$$

$$\leq \sum_{j=1}^k \sum_{x \in \Omega_j^t} \|x - c_j^{t+1}\|^2$$

$$\leq E(c_1^t, c_2^t, \ldots, c_k^t),$$

again with equality if and only if $c_j^{t+1} = c_j^t$ for $j = 1, \ldots, k$.

We now show that this implies convergence of the $k$-means algorithm. Note that if $c_j^{t+1} \neq c_j^t$ for some $j$ (so the algorithm has not converged), then we proved above that the energy is strictly decreasing, so

$$E(c_1^{t+1}, c_2^{t+1}, \ldots, c_k^{t+1}) < E(c_1^t, c_2^t, \ldots, c_k^t).$$

Hence, prior to convergence, we can never revisit the same clustering

$$\Omega_1^t, \Omega_2^t, \ldots, \Omega_k^t$$

at any step in the $k$-means algorithm. Indeed, if we were to revisit the same clustering at some future step of the algorithm, then since the means
$c_1^{t+1}, c_2^{t+1}, \ldots, c_k^{t+1}$ depend only on the clusters $\Omega_1^t, \Omega_2^t, \ldots, \Omega_k^t$, we would revisit the same configuration of cluster centers, which is impossible since the $k$-means energy is strictly decreasing with each iteration prior to convergence. Since there are only a finite number of possible ways to cluster the dataset into $k$ groups, and the $k$-means algorithm cannot revisit any given clustering, the algorithm must eventually converge.

**Remark 4.3.** The proof of Theorem 4.2 uses strict energy monotonicity to prove convergence. The proof is non-quantitive, meaning it does not say anything about how many iterations the $k$-means algorithm may take to converge. In practice, the algorithm often converges very quickly, in only a handful of iterations, but it is possible for $k$-means to take substantially longer to converge. Indeed, in the worst case, the algorithm must visit every possible clustering before converging. Even for the 2-means problem with $n$ points, there $2^n$ possible ways to cluster the data, so the search space is exponentially large.

We also remark that convergence of the $k$-means algorithm simply means that the cluster centers stop changing from one iteration to the next. This does not mean the algorithm has converged to a minimizer of the $k$-means energy (4.1), and in general the algorithm does not find global minimizers. In fact, due to the random choice of initialization, the algorithm can find different clusterings every time it is executed.

**Exercise 4.4** (Robust $k$-means clustering).

**Exercise 4.5** (Fast 2-means clustering in 1D).

### 4.1.1 Clustering MNIST digits

**Python Notebook:** .ipynb

We now consider a brief application of $k$-means clustering to real data. Again, we use the MNIST dataset of handwritten digits, and we evaluate the 2-means algorithm for clustering pairs of MNIST digits. We consider all pairs of MNIST digits (there are around 7000 images per digit), yielding $\binom{10}{2}$ binary clustering problems, each with around 14000 datapoints in dimension $\mathbb{R}^{784} = \mathbb{R}^{28 \times 28}$. The $k$-means algorithm converged very quickly, in around 15 iterations taking around 1 second per clustering problem. Table 4.1.1 shows the clustering accuracy obtained by the 2-means algorithm for each pair of MNIST digits. The numbers can vary depending on the random choice of initial condition. We can see that many pairs of digits are very easy to cluster into the correct classes with the 2-means algorithm, while a handful of pairs of
Table 4.1: Accuracy for binary (2-means) clustering of pairs of MNIST digits. We see most pairs of digits are easy to separate, while a few pairs, such as (4,9), (5,3), (7,9), (5,8), and (8,9), are more difficult.

digits, such as (4,9), (5,3), (7,9), (5,8), and (8,9) are more difficult. It is also natural to run the 10-means algorithm on the whole MNIST dataset, but the algorithm takes a long time to converge and generally gives poor clustering results.

4.2 Spectral Clustering

Python Notebook: .ipynb

The $k$-means clustering algorithm works well for clusters that are roughly spherical (e.g., blob data). For clusters with more complicated geometries, a single cluster center may not be a good representative of the whole cluster, and Euclidean distance to cluster centers may not be a good indication of which cluster a datapoint belongs to. We show an example of this on the famous 2-moons dataset in Figure 4.4. This dataset has two clusters that have a nonlinear, and non-convex, shape, and are interleaved so that a single there are no good choice of Euclidean cluster centers for either cluster. In this case, 2-means clustering performs poorly. In this section we will study another algorithm, called spectral clustering, which can uncover more complicated cluster geometries.

Spectral clustering takes a different perspective on clustering, compared to $k$-means clustering. In $k$-means clustering we attempted to find $k$ cluster centers, and clustered the data by finding, for each datapoint, which cluster center is closest in Euclidean distance. In contrast, spectral clustering aims to
Figure 4.4: 2-means clustering on the two-moons dataset fails to uncover the true clusters.

ensure that points that are nearby spatially are assigned to the same cluster. It does not compute cluster centers, and makes no assumption on the shape or geometry of the clusters.

Spectral cluster requires the computation of an adjacency matrix, or weight matrix, for the dataset, which encodes the similarities between pairs of points. If our dataset consists of \(m\) points \(x_1, x_2, \ldots, x_m\) in \(\mathbb{R}^n\), the weight matrix \(W\) is an \(m \times m\) symmetric matrix, where the entry \(W(i, j)\) represents the similarity between datapoints \(x_i\) and \(x_j\). The similarity is always nonnegative (\(W(i, j) \geq 0\)), and should be large when \(x_i\) and \(x_j\) are close together spatially, and small (or zero), when \(x_i\) and \(x_j\) are far apart. A common choice for the weight matrix is Gaussian weights

\[
W(i, j) = \exp \left( -\frac{||x_i - x_j||^2}{2\sigma^2} \right),
\]

where the \(\sigma\) is a free parameter that controls the scale at which points are connected by strong similarities \(W(i, j)\) in the weight matrix. In fact, the weight matrix \(W\) endows the dataset with a graph structure, where each pair of points \((x_i, x_j)\) is connected by an edge with edge weight \(W(i, j)\). Other choices of weight matrix are possible, such as the \(k\)-nearest neighbor graph explored in Section 4.2.2, so we proceed with a general nonnegative and symmetric \(m \times m\) weight matrix \(W\) in what follows.

While spectral clustering can be applied to general problems with \(k\) clusters, it is simplest to explain for the binary clustering problem, where we seek
$k = 2$ clusters. Let $z \in \{0, 1\}^m$ be a vector giving the cluster labels for each datapoint, so $z(i) \in \{0, 1\}$ is the cluster label for $x_i$. Here, we have assigned one cluster to have the label 0 and the other cluster the label 1. Given a weight matrix $W$, we wish to find a binary clustering for which $x_i$ and $x_j$ are assigned to the same cluster when the weight $W(i, j)$ is large (since this means the points are nearby). It is thus natural to consider the graph cut energy

$$\begin{equation}
E(z) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j) |z(i) - z(j)|^2.
\end{equation}$$

The graph cut energy sums the edge weights $W(i, j)$ corresponding to pairs of points $(x_i, x_j)$ that are assigned to different clusters, so that $z(i) \neq z(j)$. A graph cut approach to clustering amounts to attempting to minimize $E$ over cluster labels $z$, which attempts to ensure that similar datapoints, where $W(i, j)$ is large, get assigned to the same cluster. While this approach gives good clusterings, it is a hard computational problem (in fact, it is also NP hard, similar to minimizing the $k$-means clustering energy).

Spectral clustering is a tractable relaxation of the graph cut problem described above. Instead of enforcing that the cluster labels $z(i)$ belong to the discrete set $\{0, 1\}$, we relax the problem and allow $z \in \mathbb{R}^m$ to be any real-valued vector. This introduces the problem that any constant vector $z = t\mathbf{1}$ is a minimizer, where $t \in \mathbb{R}$ and $\mathbf{1}$ is the all ones vector. Thus, we need to impose some additional conditions to rule out these trivial cases. Two natural conditions to impose are

$$\begin{equation}
\mathbf{1}^T z = \sum_{i=1}^{m} z_i = 0 \quad \text{and} \quad \|z\|^2 = \sum_{i=1}^{m} z(i)^2 = 1.
\end{equation}$$

The two constraints in (4.8), taken together, ensure that the any minimizer $z$ of $E(z)$ is not a trivial constant labeling $z = t\mathbf{1}$. This leads to the binary spectral clustering algorithm:

$$\begin{equation}
\text{Minimize } E(z) \text{ over } z \in \mathbb{R}^m \text{ subject to (4.8)}.
\end{equation}$$

After finding a minimizer $z^*$, the clustering is obtained by the sign of the minimizer, so $\{x_i : z^*(i) > 0\}$ is one cluster, while $\{x_i : z^*(i) \leq 0\}$ is the other cluster.

4.2.1 The graph Laplacian and Fiedler vector

We now show how to solve the binary spectral clustering problem (4.9), which will explain the spectral part of the name.
**Definition 4.6.** Let $W$ be a symmetric $m \times m$ matrix with nonnegative entries. The associated graph Laplacian matrix $L$ is the $m \times m$ matrix

\[(4.10)\]

\[L = D - W\]

where $D$ is the diagonal matrix with diagonal entries

\[D(i, i) = \sum_{j=1}^{m} W(i, j).\]

We now show that the graph cut energy can be rewritten in terms of the graph Laplacian matrix $L$.

**Lemma 4.7.** Let $W$ be a symmetric $m \times m$ matrix with nonnegative entries. Then the graph cut energy $E$ defined in (4.7) can be expressed as

\[(4.11)\]

\[E(z) = z^T L z,\]

where $L = D - W$ is the graph Laplacian matrix.

**Proof.** We simply compute

\[
E(z) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)|z_i - z_j|^2
\]

\[= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)(z_i^2 - 2z_i z_j + z_j^2)
\]

\[= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)z_i z_j + \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)z_j^2
\]

\[= \frac{1}{2} \sum_{i=1}^{m} D(i, i)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)z_i z_j + \frac{1}{2} \sum_{j=1}^{m} D(j, j)z_j^2
\]

\[= \sum_{i=1}^{m} D(i, i)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i, j)z_i z_j
\]

\[= z^T D z - z^T W z = z^T L z,
\]

which completes the proof. \(\square\)

We recall from Exercise 2.5 that minimizing the quantity $z^T L z$, subject to the constraint $\|z\| = 1$, is related to finding eigenvectors of $L$. Due to (4.8) we have an additional constraint $1^T z = 0$ that needs to be accounted for. Before proceeding, we record some basic properties of the graph Laplacian.
Lemma 4.8. Let $L = D - w$ be the graph Laplacian corresponding to a symmetric matrix $W$ with nonnegative entries. The following properties hold.

(i) $L$ is symmetric.

(ii) $L$ is positive semi-definite (i.e., $z^T L z \geq 0$ for all $z \in \mathbb{R}^m$).

(iii) All eigenvalues of $L$ are nonnegative, and the constant vector $z = 1$ is an eigenvector of $L$ with eigenvalue $\lambda = 0$.

Proof. (i) Both $D$ and $W$ are symmetric, so $L$ is as well.

(ii) By Lemma 4.7 we have $z^T L z = E(z) \geq 0$, for any $z \in \mathbb{R}^m$, thus $L$ is positive semi-definite.

(iii) By Exercise 2.6, all eigenvalues of $L$ are nonnegative, since $L$ is positive semi-definite. To check that $z = 1$ belongs to the kernel of $L$, let $y = L1 = D1 - W1$, and check that

$$y(i) = D(i, i) - \sum_{j=1}^{m} W(i, j) = 0$$

for all $i = 1, \ldots, m$. This completes the proof.

Since $L$ is a real symmetric matrix, it is diagonalizable, and there exists an orthonormal basis $v_1, v_2, \ldots, v_m$ of $\mathbb{R}^m$ consisting of eigenvectors of $L$, with corresponding eigenvalues

$$0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_m.$$ 

That is, $Lv_i = \lambda_i v_i$. Note that $\lambda_1 = 0$ by Lemma 4.8 (iii).

Definition 4.9. The second eigenvector $v_2$ of the graph Laplacian $L$ is called the Fiedler vector. The Fiedler vector is the first non-trivial eigenvector of $L$, when they are ordered by increasing eigenvalue. The first eigenvector is constant, $v_1 = \frac{1}{\sqrt{m}} 1$, and is considered trivial.

It turns out, due to the theorem below, that the Fiedler vector is exactly the solution of the binary spectral clustering problem (4.9). Thus, spectral clustering amounts to assigning points to one of two clusters depending on the sign of the Fiedler vector. This is also the reason for the name spectral clustering—we are using the spectrum (i.e., the eigenvectors) of the graph Laplacian matrix for clustering.
Theorem 4.10. The Fiedler vector $v_2$ solves the spectral clustering problem (4.9).

Proof. A minimizer of (4.9) exists, due to the fact that we are minimizing a continuous function $E(z)$ over a closed and bounded set

$$\{z \in \mathbb{R}^m : \|z\| = 1 \text{ and } 1^Tz = 0\}.$$ 

Let $z$ be a minimizer of (4.9), and write $z$ in the basis of eigenvectors of $L$ as

$$z = \sum_{i=1}^{m} a_i v_i.$$ 

Since $\|z\| = 1$ we have by (2.1) that

$$\sum_{i=1}^{m} a_i^2 = 1.$$ 

Since $v_1 = \frac{1}{\sqrt{m}}1$, and the $v_i$ are orthonormal vectors, the condition $1^Tz = 0$ is equivalent to

$$0 = 1^Tz = \sqrt{m}v_1^T\sum_{i=1}^{m} a_i v_i = \sqrt{m} \sum_{i=1}^{m} a_i v_1^T v_i = \sqrt{m}a_1.$$ 

Therefore $a_1 = 0$. We now compute, by Lemma 4.7, that

$$E(z) = z^T L z = z^T L \sum_{i=2}^{m} a_i v_i = \sum_{i=2}^{m} a_i z^T L v_i = \sum_{i=2}^{m} \lambda_i a_i z^T v_i = \sum_{i=2}^{m} \lambda_i a_i^2,$$

since $z^T v_i = a_i$. Since $\lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_m$, $a_1 = 0$, and (4.12) holds, we have

$$E(z) = \sum_{i=2}^{m} \lambda_i a_i^2 \geq \lambda_1 \sum_{i=2}^{m} a_i^2 = \lambda_1.$$ 

Therefore, setting $a_2 = 1$ and $a_i = 0$ for $i \neq 1$ minimizes $E(z)$. This amounts to $z = v_2$, which completes the proof. \qed

We return briefly to the two-moons clustering problem from Figure 4.4, which was not clustered correctly by $k$-means. We applied spectral clustering with Gaussian weights $W(i,j)$ (see (4.6)) with $\sigma = 0.15$. Figure 5(a) shows the Fiedler vector colored with lowest values dark blue, and highest values yellow. Figure 5(b) shows the result of spectral clustering on the two-moons dataset, where the clusters are based on the sign of the Fiedler vector. Spectral clustering finds the correct clustering for the two-moons dataset, which illustrates the ability of spectral clustering to handle more complicated nonlinear cluster geometry.
Remark 4.11. We briefly remark that spectral clustering is not only used for binary clustering, and there are versions of the algorithm that work for $k$ classes. The general idea is to compute the first $k$ eigenvectors $v_1, v_2, \ldots, v_k$ of the graph Laplacian $L = D - W$ and to perform a spectral embedding of the data into $\mathbb{R}^k$. The spectral embedding $\Phi_k : \mathbb{R}^n \rightarrow \mathbb{R}^k$ is

$$\Phi_k(x_i) = (v_1(i), v_2(i), \ldots, v_k(i)).$$

That is, at each datapoint $x_i$, we evaluate all $k$ eigenvectors of the graph Laplacian and those values are the coordinates of the points $x_i$ in the $k$-dimensional embedded space. This is a very useful method for dimension reduction. The data is then clustered in the spectral embedding space $\mathbb{R}^k$, often using the $k$-means clustering algorithm. There are several different variants of spectral clustering for more than two classes, based on using different normalizations for the graph Laplacian (two common normalizations are $D^{-1}L$ and $D^{-1/2}LD^{-1/2}$), and on using further normalizations of the points in the embedded space $\mathbb{R}^k$. We refer to [2] for more information on spectral clustering.

4.2.2 Clustering MNIST digits

Python Notebook: .ipynb

We now return to the problem of clustering MNIST digits from Section 4.1.1, to see if we can improve upon the results of $k$-means clustering. The choice of Gaussian weights (4.6) is not practical for real data for a couple of
Table 4.2: Accuracy for binary spectral clustering of pairs of MNIST digits. The results are generally an improvement over 2-means clustering (see Table 4.1.1), but we still see some pairs of digits, such as (4,9) and (7,9) are hard to separate.

reasons. First, it results in a dense matrix $W$, where all entries need to be stored, which is not tractable for large datasets. Second, there are difficulties using the same scale $\sigma$ over the whole graph, since some areas of the graph may be very dense and a smaller connectivity scale $\sigma$ would be appropriate, while some areas may be more sparse and require a larger $\sigma$ to ensure they are well-connected to neighboring points in the graph.

A better way to construct a graph over a dataset in practice is to build a $k$-nearest neighbor graph. To do this, we find for each $x_i$, the $k$ nearest points $x_j$ in the Euclidean distance, and we assign positive edge weights $W(i,j)$ only for these $k$-nearest neighbors. All other weights are zero, and the matrix is stored in a sparse matrix format so the zero entries do not need to be stored in memory, nor used in computations. The weights can be defined as $W(i,j) = 1$ when $j$ is a $k$-nearest neighbor of $i$, or we can use Gaussian weights with a length scale that adapts to the graph. In this experiment we used

$$W(i,j) = \exp \left( -\frac{4\|x_i - x_j\|^2}{d_k(x_i)^2} \right),$$

where $d_k(x_i)$ is the Euclidean distance from $x_i$ to its $k^{th}$ nearest neighbor.

A minor problem with $k$-nearest neighbor graphs is that they are not symmetric, that is $W(i,j) \neq W(j,i)$. This means the graph Laplacian matrix $L = D - W$ will not be symmetric, and hence it may not be diagonalizable, the spectrum may be complex-valued, and the Fiedler vector will have

<table>
<thead>
<tr>
<th>Digit</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99.8</td>
<td>98.9</td>
<td>99.5</td>
<td>99.8</td>
<td>99.5</td>
<td>98.7</td>
<td>99.7</td>
<td>99.2</td>
<td>99.3</td>
</tr>
<tr>
<td>1</td>
<td>97.0</td>
<td>99.3</td>
<td>99.1</td>
<td>99.4</td>
<td>99.7</td>
<td>98.8</td>
<td>99.1</td>
<td>99.6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>98.3</td>
<td>99.5</td>
<td>99.1</td>
<td>99.5</td>
<td>98.0</td>
<td>98.6</td>
<td>99.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>99.6</td>
<td>82.3</td>
<td>99.6</td>
<td>99.0</td>
<td>91.8</td>
<td>97.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>99.6</td>
<td>99.3</td>
<td>98.9</td>
<td>98.9</td>
<td>53.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>97.9</td>
<td>99.8</td>
<td>90.0</td>
<td>98.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>99.8</td>
<td>99.0</td>
<td>99.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>99.1</td>
<td>70.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>97.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
no meaning. Thus, we always symmetrize the weight matrix for a \( k \)-nearest neighbor graph in some reasonable way before applying spectral clustering. In this experiment, we replace \( W \) with \( \frac{1}{2}(W + W^T) \) to symmetrize. Other choices are possible, such as symmetrizing the weights directly

\[
W(i, j) = \exp \left( -\frac{4 \|x_i - x_j\|^2}{d_k(x_i)d_k(x_j)} \right).
\]

We show in Table 4.2.2 the results of binary spectral clustering of all pairs of MNIST digits using a \( k = 10 \) nearest neighbor graph. We generally see a good improvement over the corresponding results for the \( k \)-means algorithm given in Table 4.1.1. However, there are still some pairs of digits that are difficult to separate, such as \((4, 9)\) and \((7, 9)\). We will return to this example later in the lecture notes, and we will see how the results can be further improved by using neural network autoencoders.

## 5 PageRank

**Python Notebook:** .ipynb

Having introduced graph-based methods briefly in Section 4.2, we turn here to study the PageRank algorithm, which was used by Google to rank internet search results until around 2006. While PageRank originally gained popularity as Google’s search engine, it has found a wide range of applications in other fields, including biology (GeneRank), chemistry, ecology, neuroscience, physics, sports, and computer systems. We refer to [1] for a review of PageRank and a summary of the abundance of applications outside of search engines.

PageRank is a graph-based ranking algorithm that ranks websites based on their link structure. The idea behind the PageRank algorithm is to take a random walk on the internet for a long time (by randomly following links on each webpage visited) and record how often each website is visited along the way. The websites that are visited more often get higher ranks than those that are visited less often. While this is the core of the idea behind PageRank, there is a small problem with a simple random walk on a graph like the internet—the walker may get stuck in small disconnected components of the graph that have no outgoing links. The walker will then spend all of its time in a small part of the internet (which may be very sensitive to where the walk starts), and will be unable to rank the majority sites.

To resolve this issue, PageRank uses a *random surfer*, which is a random walker that at random times teleports to a completely random website, one
that is not necessarily connected to the current website by a link. This allows
the surfer to escape poorly connected pockets of the internet and produce a
ranking for all pages. Furthermore, specific choices of the type of teleportation
lead to localized or personalized versions of PageRank, which are discussed in
Section 5.2.

To describe the PageRank algorithm mathematically, the starting point is
an adjacency matrix $W$, which encodes the links between websites. Often the
adjacency matrix is a binary matrix with

$$W(i, j) = \begin{cases} 1, & \text{if site } i \text{ links to site } j \\ 0, & \text{otherwise}. \end{cases}$$

The adjacency matrix $W$ is an $n \times n$ matrix, where $n$ is the number of webpages
to rank. The matrix $W$ is a very large matrix that is usually very sparse. That
is, most of the entries of $W$ are zero, since most websites link to only a small
fraction of the internet. Matrix operations with $W$ are thus tractable if one
stores the matrix in a sparse format, so that the zero entries are not stored in
memory, and not used in matrix/vector multiplications.

To define a random walk on a graph, we need to define a probability tran-
sition matrix $P$, which is an $n \times n$ matrix whose $(i,j)$ entry is the one-step
transition probability

$$P(i, j) = \text{Probability of stepping from } j \text{ to } i.$$ 

Clicking on a link at random from webpage $j$ leads to the transition probabil-
ities

$$P(i, j) = \frac{W(j, i)}{\sum_{k=1}^{n} W(j, k)}.$$ 

The columns of $P$ are probability distributions, and hence for all $j$ we have

$$\sum_{i=1}^{n} P(i, j) = 1.$$ 

**Exercise 5.1.** Show that $P = W^T D^{-1}$, where $D$ is the diagonal matrix with
diagonal entries $D(i, i) = \sum_{j=1}^{n} W(i, j)$.

To model the random teleportation of the surfer, we flip a biased coin at
each step, and with probability $\alpha \in [0, 1)$ the surfer takes a random walk step,
and with probability $1 - \alpha$ the surfer teleports somewhere else at random in the
internet. The teleportation step is itself drawn at random, according to a tele-
portation probability distribution $v$, which satisfies $v(i) \geq 0$ and $\sum_{i=1}^{n} v(i) = 1$. 

52
The teleportation distribution can be uniform \(v(i) = 1/n\), in which case the random surfer jumps uniformly at random to another site on the internet. Another typical choice is a localized PageRank, obtained by setting \(v(i) = \delta_{ij}\), where \(\delta_{ij} = 1\) if \(i = j\) and \(\delta_{ij} = 0\) otherwise. In this case, the random surfer always jumps back to the same website \(j\) each time it teleports, which leads to a ranking of all pages relative to \(j\). This can be useful in retrieval problems. There are of course a continuum of intermediate teleportation distributions between the uniform random distribution and localized PageRank.

Let \(x_0(i)\) denote the probability that the random surfer is initially at site \(i\) at step \(k = 0\) of the walk. We can take \(x_0\) to be any probability vector (i.e., \(x_0(i) \geq 0\) and \(\sum_i x_0(i) = 1\)). For instance, setting \(x_0(i) = \delta_{ij}\) amounts to initially placing the random surfer at page \(j\). Any other distribution \(x_0\) amounts to placing the surfer at node \(j\) with probability \(x_0(j)\) at the initial time. For \(k \geq 1\) define

\[
x_k(i) = \text{Probability that the random surfer is at page } i \text{ on step } k.
\]

To see how \(x_k\) transitions to \(x_{k+1}\) requires some probability. We condition on the location of the surfer at step \(k\), and on the outcome of the coin flip, to obtain

\[
x_{k+1}(i) = (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j) x_k(j).
\]

We can write this in matrix/vector form as

\[
(5.2) \quad x_{k+1} = (1 - \alpha)v + \alpha Px_k.
\]

The PageRank vector is defined as the limit of \(x_k\) as \(k \to \infty\)—we will prove the limit exists below. Figure 5.1 shows an illustration of the PageRank vector (small values are blue and large are yellow) on a toy graph with \(\alpha = 2/3\). The black arrows indicate links between websites. Notice that the two lowest ranked sites (the dark blue ones) have no incoming edges, so they are not visited except for on teleportation steps of the random surfer, which is one of the reasons for their low ranking. On the other hand, the highest ranked yellow node has many incoming edges and is visited very often by the random surfer.

5.1 Convergence of the random surfer

If we send \(k \to \infty\) in the PageRank iteration (5.2), and if \(x_k\) converges to some vector \(x\), then clearly \(x\) should be a solution of the linear equation

\[
(5.3) \quad x = (1 - \alpha)v + \alpha Px.
\]
Figure 5.1: An illustration of the PageRank vector on a toy graph with $\alpha = 2/3$. The colors from blue to yellow indicate the value of the PageRank vector at each node, and the black arrows indicate directed links between webpages.

We call (5.3) the PageRank problem, and its solution $x$ is the PageRank vector. In this section, we will prove that the probability distribution $x_k$ of the random surfer converges as $k \to \infty$ to the solution of the PageRank problem (5.3).

It will be more convenient to work in the $\ell_1$-norm $\| \cdot \|_1$ defined by

$$
\| x \|_1 = \sum_{i=1}^{n} |x(i)|.
$$

In the $\ell_1$-norm, the transition matrix $P$ is non-expansive, according to the following proposition.

**Proposition 5.2.** We have $\| Px \|_1 \leq \| x \|_1$.

**Proof.** The proof is a straightforward computation

$$
\| Px \|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} P(i,j)x(j) \right| \leq \sum_{i=1}^{n} \sum_{j=1}^{n} |P(i,j)x(j)| = \sum_{j=1}^{n} |x(j)| = \| x \|_1.
$$
where we used (5.1) in the third step. This completes the proof.

We first establish that the PageRank problem (5.2) has a unique solution.

**Lemma 5.3.** Let \( v \in \mathbb{R}^n \) and \( 0 \leq \alpha < 1 \). Then there is a unique vector \( x \in \mathbb{R}^n \) solving the PageRank problem (5.3). Furthermore, the following hold.

(i) We have \( \sum_{i=1}^{n} x(i) = \sum_{i=1}^{n} v(i) \).

(ii) If \( v(i) \geq 0 \) for all \( i \), then \( x(i) \geq 0 \) for all \( i \).

**Remark 5.4.** Taking (i) and (ii) together shows that whenever \( v \) is a probability distribution (i.e., \( v(i) \geq 0 \) and \( \sum_i v(i) = 1 \)), the same is true of the unique solution \( x \) of the PageRank problem (5.3).

**Proof of Lemma 5.3.** Note that we can re-write (5.3) as

\[
Ax = v
\]

where

\[
A = (1 - \alpha)^{-1}(I - \alpha P).
\]

To prove there is a unique solution \( x \), we need only show that the kernel of \( A \) is trivial. Then the mapping \( A : \mathbb{R}^n \rightarrow \mathbb{R}^n \) is injective (one-to-one), and hence it is also surjective (onto), by the rank-nullity theorem.

To show that \( \ker(A) = \{0\} \), let \( z \in \ker(A) \). Then \( Az = 0 \) and so \( z = \alpha Pz \).

By Proposition 5.2 we have

\[
\|z\|_1 = \|\alpha Pz\|_1 = \alpha \|Pz\|_1 \leq \alpha \|z\|_1.
\]

Since \( \alpha < 1 \) we must have \( \|z\|_1 = 0 \), and so \( z = 0 \), which establishes that \( A \) is injective, and so \( Ax = v \) has a unique solution \( x \), for each \( v \).

We now prove (i). We use (5.1) to obtain

\[
\sum_{i=1}^{n} x(i) = \sum_{i=1}^{n} \left( (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x(j) \right)
\]

\[
= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} x(j) \sum_{i=1}^{n} P(i,j)
\]

\[
= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} x(j).
\]

Re-arranging we have

\[
(1 - \alpha) \sum_{i=1}^{n} x(i) = (1 - \alpha) \sum_{i=1}^{n} v(i).
\]
Since $\alpha < 1$, we can divide by $1 - \alpha$ to complete the proof of (i).

We finally prove (ii). Assume $v(i) \geq 0$ for all $i$. We first compute, using (5.3), that

$$|x(i)| = \left| (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x(j) \right|$$

$$\leq (1 - \alpha)|v(i)| + \alpha \sum_{j=1}^{n} P(i,j)|x(j)|$$

$$\leq (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)|x(j)|.$$ 

Summing both sides over $i = 1, \ldots, n$ and using (5.1) we find that

$$\sum_{i=1}^{n} |x(i)| \leq (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{i=1}^{n} \sum_{j=1}^{n} P(i,j)|x(j)|$$

$$= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} |x(j)|.$$ 

Re-arranging and using part (i) we have

$$(1 - \alpha) \sum_{i=1}^{n} |x(i)| \leq (1 - \alpha) \sum_{i=1}^{n} v(i) = (1 - \alpha) \sum_{i=1}^{n} x(i).$$

Since $\alpha < 1$ we can divide by $1 - \alpha$ to deduce that

$$\sum_{i=1}^{n} |x(i)| \leq \sum_{i=1}^{n} x(i).$$

It follows that $x(i) = |x(i)| \geq 0$ for all $i$, which completes the proof. □

**Remark 5.5.** It is common to re-write the PageRank problem (5.3) as an eigenvector problem. Let us assume we are in the setting where $v$ and $x$ are probability distributions, so $x(i) \geq 0$ and $\sum_{i=1}^{n} x(i) = 1$. Noting that $1^T x = \sum_{i=1}^{n} x(i) = 1$ we can re-write (5.3) as

$$x = (1 - \alpha)v1^T x + \alpha Px = ((1 - \alpha)v1^T + \alpha P) x.$$ 

The matrix

$$P_{\alpha} := (1 - \alpha)v1^T + \alpha P$$

(5.4)
is exactly the probability transition matrix for the random surfer, and the PageRank problem $P_x = x$ is an eigenvector problem, which merely states that the PageRank vector is the invariant (or stationary) distribution of the induced Markov chain. In this light, Lemma 5.3 (ii) is a direct consequence of the Perron-Frobenius Theorem.

We can now prove convergence, with a linear rate, of the random surfer’s distribution $x_k$ to the PageRank vector.

**Theorem 5.6.** Let $v \in \mathbb{R}^n$ and $0 \leq \alpha < 1$. Let $x_k$ satisfy the PageRank iteration (5.2), and let $x$ be the unique solution of the PageRank problem (5.3) (i.e., the PageRank vector). Then we have

$$\|x_k - x\|_1 \leq \alpha^k \|x_0 - x\|_1. \tag{5.5}$$

**Proof.** We simply subtract

$$x = (1 - \alpha)v + \alpha Px$$

from

$$x_k = (1 - \alpha)v + \alpha P x_{k-1}$$

to obtain

$$\|x_k - x\|_1 = \|\alpha P x_{k-1} - \alpha Px\| = \alpha \|P(x_{k-1} - x)\|_1.$$

Using Proposition 5.2 we obtain

$$\|x_k - x\|_1 \leq \alpha \|x_{k-1} - x\|_1.$$

The proof is completed by induction. □

**Remark 5.7.** Since $0 \leq \alpha < 1$, Theorem 5.6 shows that the PageRank iterations $x_k$ converge to the PageRank vector $x$ as $k \to \infty$ at the linear rate of $\alpha$ in the $\ell_1$-norm. In particular, the convergence can be very slow if $\alpha$ is close to one. This shows another advantage to introducing the teleportation step; not only does it guarantee convergence of the PageRank iterations, but it gives the user control over the convergence rate.

In practice, the standard way to compute the solution of the PageRank problem (5.3) is via the PageRank iteration (5.2), since it is simple to compute (especially with large sparse matrices), and can be terminated early to obtain an approximate solution. The PageRank iteration (5.2) can be interpreted as the power iteration for computing the largest eigenvector of a matrix. Indeed, following Remark 5.5, we can rewrite the PageRank iteration as $x_{k+1} = P_\alpha x_k$, where $P_\alpha$ is defined in (5.4). This is exactly the power iteration, except the normalization step appears to be omitted (normally it would be $x_{k+1} = P_\alpha x_k/\|P_\alpha x\|$). The normalization is not needed since $P$ is non-expansive (see Proposition 5.2).
5.2 Personalized PageRank for image retrieval

Python Notebook: .ipynb

We give here an example of using personalized, or localized, PageRank for image retrieval. Image retrieval takes a query image and tries to find similar images in a dataset. We consider the MNIST dataset of handwritten digits, and construct a $k$-nearest neighbor graph over the dataset as described in Section 4.2.2. To retrieve images similar to image $j$, we set the teleportation distribution to be $v(i) = \delta_{ij}$, in order to rank all images based on their similarity to image $j$. We then solve the PageRank problem (5.3) with the iteration (5.2), and return the top ranked images. We took one image from each of the 10 MNIST digits and ran personalized PageRank to retrieve the top 14 similar images. Figure 5.2 shows the results. The image on the left is the query image, and the following 14 images are the retrieved images. We note that most digits are from the same class, and are in fact written in a very similar way to the query digit. There are a handful retrieved digits that are
from different classes than the query; for example, a 5 in the 8 row and some 4's in the 9 row. There is also what appears to be an 8 in the 7 row, but this is hard to distinguish by eye.
6 The Discrete Fourier Transform

7 The Discrete Wavelet Transform

8 Variational Methods in Image Processing

9 Machine Learning

9.1 Introduction and basic algorithms

9.2 Neural Networks

9.3 Convolutional Neural Networks

References
